Using a Database

15.1 Database Tables and SQL Queries
15.2 Connecting to a Database
15.3 Retrieving Information
15.4 Database Information and Aggregate Functions
15.5 Stored Procedures and Transactions
15.6 A GUI for Database Queries

We can use files to store data for small applications, but as the amount of data that we need to save gets larger, the services of a database system become invaluable. A database system allows us to model the information we need while it handles the details of inserting, removing, and retrieving data from individual files in response to our requests.

Of course, each database vendor provides its own procedures for performing database operations. The .NET Framework hides the details of different databases; our programs can work with many different databases on many different platforms. They can be used as part of large-scale enterprise applications. In this chapter we cover the concepts using a small example that allows many extensions, some of which we pursue in the exercises.

The example programs illustrate database access using console applications to avoid obscuring the programs with the details involved in building a GUI. In the last section, our extended case study develops a graphical user interface to a database.

OBJECTIVES

- Introduce relational database tables
- Introduce SQL (Structured Query Language)
- Connect to a database from C#
- Build a database using SQL
- Use C# to query a database
- Obtain the properties of a database

- Introduce selected aggregate functions
- Use stored procedures for efficiency
- Process database transactions
- Provide a GUI for the user to query a database

15.1 DATABASE TABLES AND SQL QUERIES

Database design is best left to other texts and courses. We introduce a few database concepts here to provide an example with which to illustrate the techniques for working with databases using C#. Relational databases provide an implementation-independent way for users to view data. The Structured Query Language (SQL) lets us create, update, and query a database using standard commands that hide the details of any particular vendor's database system.

Relational Database Tables

When designing a database we need to identify the entities in our system. For example, a company might use a database to keep track of its sales and associated information. In our example company, an order includes one customer who can order several items. A salesperson may take several orders from the same customer, but each order is taken by exactly one salesperson.

Using a relational database, we keep our data in tables. In our example, we might have a `Customer` table with fields for the customer id, name, address, and balance due, as shown in Figure 15.1.

Each row of the table represents the information needed for one customer. We assign each customer a unique customer ID number. Customer names are not unique; moreover, they may change. `CustomerID` is a key that identifies the data in the row. Knowing the `CustomerID`, we can retrieve the other information about that customer.

Do not embed spaces in field names. Use `CustomerID` rather than `Customer ID`.

Figures 15.2 and 15.3 show the `Salesperson` and `Item` tables, which we define in a similar manner. A more realistic example would have additional fields.

FIGURE 15.1 The `Customer` table

CustomerID	*CustomerName*	*Address*	*BalanceDue*
1234	Fred Flynn	22 First St.	1667.00
5678	Darnell Davis	33 Second St.	130.95
4321	Marla Martinez	44 Third St.	0
8765	Carla Kahn	55 Fourth St.	0

FIGURE 15.2 The Salesperson table

SalespersonID	SalespersonName	Address
12	Peter Patterson	66 Fifth St.
98	Donna Dubarian	77 Sixth St.

FIGURE 15.3 The Item table

ItemNumber	Description	Quantity
222222	radio	32
333333	television	14
444444	computer	9

FIGURE 15.4 The Orders table

OrderNumber	CustomerID	SalespersonID	OrderDate
1	1234	12	4/3/99
2	5678	12	3/22/99
3	8765	98	2/19/99
4	1234	12	4/5/99
5	8765	98	2/28/99

The `SalespersonID` serves as the key for the `Salesperson` table, and we use the `ItemNumber` to identify an item in the `Item` table. We have to be more careful in designing the `Orders` table, because an order can have multiple items. We use a second table, the `OrderItem` table, to list the items in each order. Figure 15.4 shows the `Orders` table with the fields `OrderNumber`, `CustomerID`, `SalespersonID`, and `OrderDate`. The `OrderNumber` is the key. `CustomerID` and `SalespersonID` are foreign keys that allow us to avoid redundancy by referring to data in other tables. For example, including the `CustomerID` lets us find the customer's name and address from the `Customer` table rather than repeating it in the `Orders` table.

When choosing field names, avoid names like `Number`, `Value`, `Order`, `Name`, or `Date` that might conflict with reserved names in the database system.

The `OrderItem` table uses a compound key consisting of both the `OrderNumber` and the `ItemNumber` to identify a specific item that is part of an order. Figure 15.5 shows that each pair (`OrderNumber`, `ItemNumber`) occurs only

FIGURE 15.5 The `OrderItem` table

OrderNumber	ItemNumber	Quantity	UnitPrice
1	222222	4	27.00
1	333333	2	210.50
1	444444	1	569.00
2	333333	2	230.95
3	222222	3	27.00
3	333333	1	230.95
4	444444	1	569.00
5	222222	2	27.00
5	444444	1	725.00

once, identifying a row containing the data for a specific item in a particular order. For example, the first row shows that for order number one, and item 222222, four units were ordered at a price of $27 each.

Now that we have defined our `Sales` database, we want to see how to get information from it and how to make changes as needed.

Structured Query Language (SQL)

The Structured Query Language (SQL) is a standard language with which to get information from or make changes to a database. We can execute SQL statements from within C#. The SQL statements we will use are `CREATE`, `SELECT`, `INSERT`, `DELETE`, and `UPDATE`. We illustrate these statements using the `Sales` database defined in the previous section. The names for the data types may depend on the actual database system used. Our examples work with Microsoft Access.

We could use the `CREATE` statement

```
CREATE TABLE Customer (CustomerID CHAR(4), CustomerName
       VARCHAR(25), Address VARCHAR(25), BalanceDue DECIMAL)
```

to create the `Customer` table, the statement

```
CREATE TABLE Orders (OrderNumber VARCHAR(4), CustomerID
       CHAR(4), SalespersonID CHAR(2), OrderDate DATE)
```

to create the `Orders` table, and the statement

```
CREATE TABLE OrderItem (OrderNumber VARCHAR(4), ItemNumber
       CHAR(6), Quantity INTEGER, UnitPrice DECIMAL)
```

to create the `OrderItem` table. We use character fields for `CustomerID`, `OrderNumber`, `SalespersonID`, and `ItemNumber`, even though they use numerical characters, because we have no need to do arithmetic using these values. By contrast, we use the type `INTEGER` for the `Quantity` field because we may wish to compute with it.

FIGURE 15.6 **SQL data types**

Type	*Standard SQL Description*
CHAR(N)	Fixed size string of length N
VARCHAR(N)	Variable size string up to length N
INTEGER	32-bit integer
DATE	Year, month, and day
DECIMAL	Used for dollars and cents

Standard SQL uses various types, which are not all supported in every database system. Figure 15.6 shows the SQL types we use in this text.

The type `DECIMAL(M,N)`, where `M` is the maximum number of digits and `N` is the maximum number of digits after the decimal point, is standard SQL, but is not supported in Access.

To insert the first row in the `Customer` table, we could use the `INSERT` statement

```
INSERT INTO Customer
VALUES (1234, 'Fred Flynn', '22 First St.', 1667.00)
```

Use the single quote, `'`, to enclose strings within an SQL statement.

The statement

```
INSERT INTO Orders VALUES (1,1234,12,'Apr 3, 1999')
```

inserts the first row into the `Order` table. We write dates in the form

```
Month Day, Year
```

to avoid confusion among date formats used in various locales and to indicate the century explicitly. The database system translates this form to its internal representation and can present dates in various formats in its tables.

The `DELETE` statement

```
DELETE FROM OrderItem WHERE OrderNumber = '1'
```

will delete the first three rows of the `OrderItem` table in Figure 15.5. These rows contain the data for the three items comprising the order with an `OrderNumber` of 1.

Use the single equality sign, `=`, in the equality test, OrderNumber = 1, instead of the C# equality symbol, `==`.

To delete just the televisions from that order and leave the order for radios and a computer, we could use the statement

```
DELETE FROM OrderItem
WHERE OrderNumber = '1' AND ItemNumber = '333333'
```

To update an existing row we use the UPDATE statement. For example, to reduce the number of radios in order number 1 to 3, we can use the statement

```
UPDATE OrderItem SET Quantity = 3
WHERE OrderNumber = '1' AND ItemNumber = '222222'
```

When we change an order we will also want to change the balance due in the Customer table, which we can do using

```
UPDATE Customer SET BalanceDue = 1640.00
WHERE CustomerID = '1234'
```

Because the OrderItem table uses a compound key

```
(OrderNumber, ItemNumber)
```

to identify a row, we need to specify values for both in the WHERE clause. In updating the Customer table we only need to specify the value of the single CustomerID key to identify a row.

The CREATE statement creates a table, and the INSERT, DELETE, and UPDATE statements make changes in a table. In many applications, we retrieve information from the database more frequently than we create a table or make changes to a table. To retrieve information we use the SELECT statement.

The simplest query we can make is to retrieve the entire table. For example, the statement

```
SELECT * FROM Customer
```

retrieves the entire Customer table. We use the star symbol, *, which matches every row. To retrieve the names and addresses of the customers we use the statement

```
SELECT CustomerName, Address FROM Customer
```

If we do not want data from the entire table, we can use a WHERE clause to specify a condition that the data of interest satisfy. For example, to retrieve all orders for radios we could use the statement

```
SELECT * FROM OrderItems
WHERE ItemNumber = '222222'
```

The power of database systems becomes evident when we use SQL to get information combined from several tables. For example, suppose we would like to know the names of all customers who placed orders on March 22, 1999. We can find that information using the statement

```
SELECT CustomerName FROM Customer, Orders
WHERE Customer.CustomerID = Orders.CustomerID
AND  OrderDate = #3/22/99#
```

where #3/22/99# is the Microsoft Access format for a date.

When a field such as `Address` occurs in more than one table, prefix the field name with the table name, as in `Customer.Address`, to state precisely which `Address` field you desire. Similarly, use the prefixes `Customer` and `Orders` to refer to the `CustomerID` fields in each of these tables.

In finding the names of customers who placed orders on March 22, 1999, the database joins two tables. Customer names occur in the `Customer` table, but we find order dates in the `Orders` table, so we list both the `Customer` and the `Orders` tables in the `FROM` part of the query. We want to find which orders each customer placed. `CustomerID`, the primary key of the `Customer` table, is also a foreign key of the `Orders` table. For each `CustomerID` in the `Customer` table we want to inspect only the rows of the `Orders` table that have the same `CustomerID`, so we include the condition

```
Customer.CustomerID = Orders.CustomerID
```

in our query.

The first row of the `Customer` table has a `CustomerID` of `1234`. The first and fourth rows of the `Orders` table have the same `CustomerID` of `1234`, but neither of the `OrderDate` fields equals `3/22/99`. The second row of the `Customer` table has `CustomerID 5678`, as does the second row of the `Orders` table, and the `OrderDate` is `3/22/99`, so the system adds '`Darnell Davis`' to the result set of customers placing orders on March 22, 1999. Continuing the search turns up no further matches. A three-line SQL statement can cause many steps to occur in the process of retrieving the requested information. The database handles all the details. We will use other interesting examples of `SELECT` statements when we develop our C# programs later in this chapter.

Figure 15.7 shows the general pattern for the `SQL` statements we have introduced so far.

```
CREATE TABLE tablename
     (fieldname1 TYPE1, fieldname2 TYPE2, ... , fieldnameN TYPEn)

INSERT INTO tablename
VALUES (field1value,field2value, ..., fieldNvalue)

DELETE FROM tablename
WHERE fieldname1 = value1 ... AND fieldnameN = valueN

UPDATE tablename SET fieldnameToSet = newValue
WHERE fieldname1ToCheck = value1ToCheck

SELECT fieldname1, ..., fieldnameN FROM table1, ..., tableM
WHERE condition1 ... AND conditionN
```

FIGURE 15.7 Some patterns for SQL statements

The Big Picture

In a relational database we keep our data in tables, making sure not to enter information redundantly. Using SQL, we can write statements to create a table, insert, delete, and update elements, and query the database. Generally, SQL is standardized so queries do not reflect implementation details of specific database systems.

✓ Test Your Understanding

1. Why is it a good idea to use `SalespersonID` as the key in the `Salesperson` table rather than the salesperson's name?
2. Write an SQL statement to create the `Salesperson` table with the fields shown in Figure 15.2.
3. Write SQL statements to insert the data shown in Figure 15.2 into the `Salesperson` table.
4. Write an SQL statement to add a new salesman, Paul Sanchez, who lives at 88 Seventh St., and has an ID of 54, to the `Salesperson` table of Figure 15.2.
5. Write an SQL statement to delete Carla Kahn's order of a computer from the `Sales` database.
6. Write an SQL statement to find the names of all salespersons in the `Sales` database.
7. Write an SQL statement to find the order numbers of all orders taken by Peter Patterson.

15.2 CONNECTING TO A DATABASE

After an overview contrasting two-tiered with three-tiered architectures for software systems, we show how to connect to a database using C#.

Database and Application Servers

In building large systems, a database server may reside on one machine to which various clients connect when they need to access the stored data (Figure 15.8).

In a three-tiered design, business logic resides in a middle machine, sometimes called an application server, which acts as a server to various application clients (Figure 15.9). These clients provide user interfaces to the business applications on the middle machine, which is itself a client of the database server.

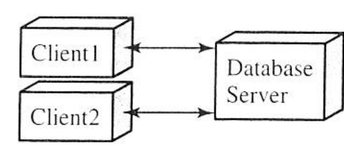


FIGURE 15.8 Client–server database access

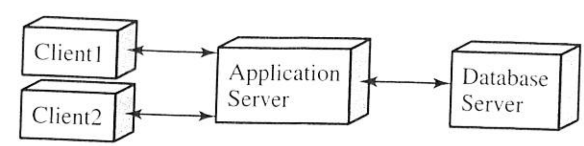


FIGURE 15.9 A three-tiered system architecture

For example, a business may have an accounting department that runs a payroll client providing a user interface to the payroll application on the middle machine which itself is a client of the database server. The marketing department might have several client programs running in their sales offices enabling salespersons to get necessary information. Rather than configuring each salesperson's machine to process all the details of the application, the company simply allows the sales staff to interact with the sales application on the middle machine. This sales program gets data from the database server as needed.

The .NET Framework allows us to write C# programs that will work no matter which database system we use. We can work entirely on one machine or use a two-tier, three-tier, or even more complex architecture for our system. Microsoft Access is suitable for work on a single machine, but we can still illustrate a three-tiered architecture because the tiers are logical, not necessarily physical, separations.

Creating the Database

We open Microsoft Access and create a blank database, give it a file name, and save it. We will refer to the database in our program by the file name we use to create it. In this chapter we use the name `Sales` and the file `Sales.mdb` to hold our database of five tables.

Connecting from C#

We want our `Sales` database to contain the five tables with the data shown in Figures 15.1 through 15.5. We could create these tables and populate them within Access, but prefer to show how to do this using C#.

The .NET Framework provides an `SqlConnection` class in the `System.Data.SqlClient` namespace to connect to the Microsoft SqlServer database and an `OleDbConnection` class in the `System.Data.OleDb` namespace to connect to Microsoft Access and other database systems. Example 15.1 will connect to the `Sales` database. The code we use will occur at the beginning of all of our examples in this chapter.

EXAMPLE 15.1 ■ Connect.cs

```
/* Connects to a Microsoft Access database.
 */

using System;
using System.Data;
```

```
using System.Data.OleDb;
public class Connect {
 public static void Main () {
   String connect = "Provider=Microsoft.JET.OLEDB.4.0;"
     + @"data source=c:\booksharp\gittleman\ch15\Sales.mdb";
                                                     // Note 1
   OleDbConnection con = new OleDbConnection(connect); // Note 2
   con.Open();                                         // Note 3
   Console.WriteLine
       ("Made the connection to the Sales database");
   con.Close();                                        // Note 4
 }
}
```

```
Made the connection to the Sales database
```

Note 1:
```
String connect = "Provider=Microsoft.JET.OLEDB.4.0;"
    + @"data source=c:\booksharp\gittleman\ch15\Sales.mdb";
```

We use an OLE DB connection string that includes the provider and the data source, which is the file containing the database we created. The @ character in front of the string signifies that every character is a literal and not a special character such as an escape character. C# calls such a string a *verbatim* string. Normally the backslash is an escape character that combines with the next character to form a special character. For example, '\b' represents a backspace. Had we omitted the @ character we would have to escape the backslashes by writing the string as

```
data source=c:\\booksharp\\gittleman\\ch15\\Sales.mdb
```

Note 2:
```
OleDbConnection con = new OleDbConnection(connect);
```

We pass the connection string to the `OleDbConnection` constructor to connect to the `Sales` database.

Note 3:
```
con.Open();
```

We open the connection to the database.

Note 4:
```
con.Close();
```

We close the database to release any resources used and do any necessary cleanup.

Building the Database

Once we have a connection to the database we can execute SQL statements to create our database. The `CreateCommand` method returns an `OleDbCommand` object that we use to send SQL statements to the database.

Some SQL statements, such as those used to create tables and insert values in a table, change the database but do not return any values to the program. To execute SQL `CREATE` and `INSERT` statements we use the `ExecuteNonQuery` method. We set the `CommandText` property of an `OleDbCommand` with the SQL statement and then invoke `ExecuteNonQuery`. For example,

```
cmd.CommandText =
    "INSERT INTO Item VALUES ('55555', 'CD player', 10)";
cmd.ExecuteNonQuery();
```

would insert a fourth row into the `Item` table.

Example 15.2 uses C# to create and populate the `Sales` database. We create the five tables shown in Figures 15.1 through 15.5, using a `CREATE` statement to create each table and `INSERT` statements to add the rows. Figure 15.10 shows the resulting Access `Sales` database and Figure 15.11 shows the `Customer` table that results from executing Example 15.2.

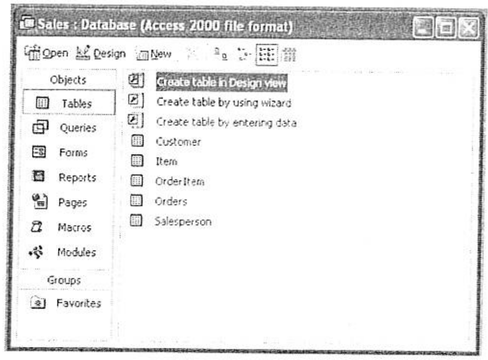
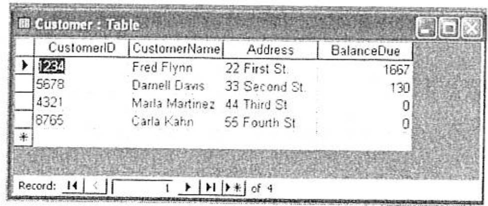


FIGURE 15.10 The Access `Sales` database created by Example 15.2



CustomerID	CustomerName	Address	BalanceDue
1234	Fred Flynn	22 First St.	1667
5678	Darnell Davis	33 Second St.	130
4321	Marla Martinez	44 Third St.	0
8765	Carla Kahn	55 Fourth St.	0

FIGURE 15.11 The `Customer` table created by Example 15.2

After running this program, the database contains the five tables. Running the program again will cause an error unless the tables are first deleted from the database.

EXAMPLE 15.2 Create.cs

```
/* Creates and populates the Sales database
 */

using System;
using System.Data;
using System.Data.OleDb;
public class Create {
 public static void Main () {
   String connect = "Provider=Microsoft.JET.OLEDB.4.0;"
    + @"data source=c:\booksharp\gittleman\ch15\Sales.mdb";
   OleDbConnection con = new OleDbConnection(connect);
   con.Open();
   Console.WriteLine
        ("Made the connection to the Sales database");
   OleDbCommand cmd = con.CreateCommand();

   cmd.CommandText = "CREATE TABLE Customer (CustomerID "
          + "CHAR(4), CustomerName VARCHAR(25), Address "
          + "VARCHAR(25), BalanceDue DECIMAL)";        // Note 1
   cmd.ExecuteNonQuery();
   cmd.CommandText = "INSERT INTO Customer VALUES (1234,'Fred "
                 + "Flynn','22 First St.',1667.00)";
   cmd.ExecuteNonQuery();
   cmd.CommandText = "INSERT INTO Customer VALUES "    // Note 2
            "(5678,'Darnell Davis','33 Second St.',130.95)";
   cmd.ExecuteNonQuery();
   cmd.CommandText = "INSERT INTO Customer VALUES (4321,'Marla "
                   + "Martinez','44 Third St.',0)";
   cmd.ExecuteNonQuery();
   cmd.CommandText = "INSERT INTO Customer VALUES (8765,'Carla "
                   + "Kahn','55 Fourth St.', 0)";
   cmd.ExecuteNonQuery();

   cmd.CommandText = "CREATE TABLE Salesperson (SalespersonID "
                 + "CHAR(2), SalespersonName VARCHAR(25), "
                 + "Address VARCHAR(25))";
   cmd.ExecuteNonQuery();
   cmd.CommandText = "INSERT INTO Salesperson VALUES "
                 + "(12,'Peter Patterson','66 Fifth St.')";
   cmd.ExecuteNonQuery();
```

```
cmd.CommandText = "INSERT INTO Salesperson VALUES "
                  + "(98,'Donna Dubarian','77 Sixth St.')";
cmd.ExecuteNonQuery();

cmd.CommandText = "CREATE TABLE Item (ItemNumber CHAR(6),"
          + "Description VARCHAR(20), Quantity INTEGER)";
cmd.ExecuteNonQuery();
cmd.CommandText =
          "INSERT INTO Item VALUES (222222,'radio',32)";
cmd.ExecuteNonQuery();
cmd.CommandText =
       "INSERT INTO Item VALUES (333333,'television',14)";
cmd.ExecuteNonQuery();
cmd.CommandText =
         "INSERT INTO Item VALUES (444444,'computer',9)";
cmd.ExecuteNonQuery();

cmd.CommandText = "CREATE TABLE Orders (OrderNumber "
         + "VARCHAR(4), CustomerID CHAR(4), SalespersonID "
         + " CHAR(2), OrderDate DATE)";
cmd.ExecuteNonQuery();
cmd.CommandText =
     "INSERT INTO Orders VALUES (1,1234,12,'Apr 3, 1999')";
cmd.ExecuteNonQuery();
cmd.CommandText =
    "INSERT INTO Orders VALUES (2,5678,12,'Mar 22, 1999')";
cmd.ExecuteNonQuery();
cmd.CommandText =
    "INSERT INTO Orders VALUES (3,8765,98,'Feb 19, 1999')";
cmd.ExecuteNonQuery();
cmd.CommandText =
    "INSERT INTO Orders VALUES (4,1234,12,'Apr 5, 1999')";
cmd.ExecuteNonQuery();
cmd.CommandText =
    "INSERT INTO Orders VALUES (5,8765,98,'Feb 28, 1999')";
cmd.ExecuteNonQuery();

cmd.CommandText = "CREATE TABLE OrderItem (OrderNumber "
               + "CHAR(4), ItemNumber CHAR(6), Quantity "
               + "INTEGER, UnitPrice DECIMAL)";
cmd.ExecuteNonQuery();
cmd.CommandText =
  "INSERT INTO OrderItem VALUES (1,222222,4,27.00)";// Note 3
cmd.ExecuteNonQuery();
cmd.CommandText =
       "INSERT INTO OrderItem VALUES (1,333333,2,210.50)";
cmd.ExecuteNonQuery();
```

```
    cmd.CommandText =
        "INSERT INTO OrderItem VALUES (1,444444,1,569.00)";
    cmd.ExecuteNonQuery();
    cmd.CommandText =
        "INSERT INTO OrderItem VALUES (2,333333,2,230.95)";
    cmd.ExecuteNonQuery();
    cmd.CommandText =
        "INSERT INTO OrderItem VALUES (3,222222,3,27.00)";
    cmd.ExecuteNonQuery();
    cmd.CommandText =
        "INSERT INTO OrderItem VALUES (3,333333,1,230.95)";
    cmd.ExecuteNonQuery();
    cmd.CommandText =
        "INSERT INTO OrderItem VALUES (4,444444,1,569.00)";
    cmd.ExecuteNonQuery();
    cmd.CommandText =
        "INSERT INTO OrderItem VALUES (5,222222,2,27.00)";
    cmd.ExecuteNonQuery();
    cmd.CommandText =
        "INSERT INTO OrderItem VALUES (5,444444,1,725.00)";
    cmd.ExecuteNonQuery();
    con.Close();
  }
}
```

Note 1:

```
cmd.CommandText = "CREATE TABLE Customer (CustomerID "
  + "CHAR(4), CustomerName VARCHAR(25), Address "
  + "VARCHAR(25), BalanceDue DECIMAL)";
```

Just as with any string, we need to split the SQL statement over multiple lines using the concatenation operator so that each string constant fits on one line.

Note 2:

```
cmd.CommandText = "INSERT INTO Customer VALUES "
      "(5678,'Darnell Davis','33 Second St.',130.95)";
```

When splitting the SQL statement over multiple lines we must be sure to add spaces to separate identifiers. Without the spaces either after `Customer` or before `VALUES`, the juxtaposition of `CustomerVALUES` would cause an error.

Note 3:

```
cmd.CommandText =
    "INSERT INTO OrderItem VALUES (1,222222,4,27.00)";
```

Using nine statements to insert the nine rows into the `OrderItem` table is cumbersome and would be more so if the table were larger. A better

Try

Try

Try

Viewi

method is to read the data to enter from a file. We leave this improvement to the exercises.

The BIG Picture

We use a connection string that specifies the provider and the data source to connect to a database. Once connected to the database, we can create tables and insert data into them from a C# program. Optionally, we could have created the tables outside of C#.

✓Test Your Understanding

Try It Yourself ➤ 8. Rewrite Example 15.1 to omit the @ character in the connection string. What other changes to that string do you need to make?

Try It Yourself ➤ 9. What happens when you try to run Example 15.2 twice in succession?

Try It Yourself ➤ 10. Modify Example 15.2, as described in Note 2, to omit the spaces after `Customer` and before `VALUES`. What is the effect of this change?

15.3 RETRIEVING INFORMATION

Now that we have created the `Sales` database we can extract information from it. When executing an SQL statement that returns results, we use the `ExecuteReader` method, which returns an `OleDbDataReader` that allows us to read a stream of data from a data source containing the data that satisfies the query. Executing

```
cmd.CommandText =
      "SELECT CustomerName, Address FROM Customer";
OleDbDataReader reader = cmd.ExecuteReader();
```

provides a reader to obtain the `CustomerName` and `Address` columns from the `Customer` table.

Viewing Query Results

To view the results, the `OleDbDataReader` has `getXXX` methods where `XXX` is the C# type corresponding to the SQL type of the data field we are retrieving. Because `CustomerName` and `Address` both have the `VARCHAR` SQL type, we use the `GetString` method to retrieve these fields. We retrieve fields by field number, starting with zero, as with arrays indexing. The loop

FIGURE 15.12
C# methods for SQL types

C# method	SQL type
GetInt32	INTEGER
GetString	VARCHAR
GetDecimal	DECIMAL
GetDate	DATE

```
while(reader.Read())
  Console.WriteLine("{0}\t{1}",
                  reader.GetString(0), reader.GetString(1));
```

will list the rows of names and addresses from the `Customer` table. We retrieve the `CustomerName` field using its column number 0 and the `Address` field using its column number 1. The `Read()` method returns `true` when another row is available and `false` otherwise. It advances the `OleDbReader` to the next record. Figure 15.12 shows the C# methods corresponding to the SQL types we use.

SELECT Statement Options

The `SELECT` statement has additional options. The `ORDER` clause allows us to display the results sorted with respect to one or more columns. The query

```
SELECT CustomerName, Address FROM Customer
ORDER BY CustomerName
```

returns the result set by name in alphabetical order. We could use

```
SELECT CustomerName, Address FROM Customer
ORDER BY 1
```

to achieve the same result using the column number in the `ORDER` clause.

Sometimes a query may return duplicate rows. For example, in selecting customers who ordered computers we would get the result

```
Fred Flynn
Fred Flynn
Carla Kahn
```

because Fred Flynn bought computers in orders 1 and 4. We can remove duplicates by using the `SELECT DISTINCT` variant of the `SELECT` statement. This query,

```
SELECT DISTINCT CustomerName
FROM Customer, Item, Orders, OrderItem
WHERE Customer.CustomerID = Orders.CustomerID
AND Orders.OrderNumber = OrderItem.OrderNumber
AND OrderItem.ItemNumber = Item.ItemNumber
AND Description = 'computer'
```

joins rows from four tables to produce the result.

The `UPDATE` and `DELETE` statements change the database, but do not return results, so we use the `ExecuteNonQuery` method to execute them.

EXAMPLE 15.3 ■ ExtractInfo.cs

```
/* Demonstrates the use of SQL queries from
 * a C# program.
 */

using System;
using System.Data;
using System.Data.OleDb;
public class ExtractInfo {
 public static void Main () {
   String connect = "Provider=Microsoft.JET.OLEDB.4.0;"
    + @"data source=c:\booksharp\gittleman\ch15\Sales.mdb";
   OleDbConnection con = new OleDbConnection(connect);
   con.Open();
   Console.WriteLine
     ("Made the connection to the Sales database");

   OleDbCommand cmd = con.CreateCommand();
   cmd.CommandText = "SELECT CustomerName, Address "
                   + "FROM Customer ORDER BY CustomerName";
   OleDbDataReader reader = cmd.ExecuteReader();
   Console.WriteLine("   Names and Addresses of Customers");
   Console.WriteLine("Name\t\tAddress");                 // Note 1
   while(reader.Read())
     Console.WriteLine("{0}\t{1}",
           reader.GetString(0), reader.GetString(1));
   reader.Close();                                       // Note 2

   cmd.CommandText = "SELECT * FROM OrderItem "
             + "WHERE ItemNumber = '222222'";
   reader = cmd.ExecuteReader();
   Console.WriteLine();
   Console.WriteLine("   Order items for radios");
   Console.WriteLine("OrderNumber\tQuantity\tUnitPrice");
   while (reader.Read())
     Console.WriteLine("{0}\t\t{1}\t\t{2}",reader.GetString(0),
          reader.GetInt32(2), reader.GetDecimal(3));  //Note 3
   reader.Close();

   cmd.CommandText = "SELECT CustomerName FROM Customer,Orders "
           + "WHERE Customer.CustomerID = Orders.CustomerID "
           + "AND  OrderDate = #3/22/99#";
   reader = cmd.ExecuteReader();
```

```
Console.WriteLine();
Console.WriteLine
  ("   Customer placing orders on Mar 22, 1999");
while(reader.Read())
  Console.WriteLine(reader.GetString(0));
reader.Close();

cmd.CommandText = "SELECT DISTINCT CustomerName "
         + "FROM Customer, Item, Orders, OrderItem "
         + "WHERE Customer.CustomerID = Orders.CustomerID "
         + "AND Orders.OrderNumber = OrderItem.OrderNumber "
         + "AND OrderItem.ItemNumber = Item.ItemNumber "
         + "AND Description = 'computer'";
reader = cmd.ExecuteReader();
Console.WriteLine();
Console.WriteLine("   Customers ordering computers");
while(reader.Read())
  Console.WriteLine(reader.GetString(0));
reader.Close();

cmd.CommandText =  "SELECT OrderNumber FROM Orders "
          + "WHERE OrderDate BETWEEN #4/1/99# AND #4/30/99#";
reader = cmd.ExecuteReader();
Console.WriteLine();
Console.WriteLine
   ("   Order numbers of orders from 4/1/99 to 4/30/99");
while(reader.Read())
  Console.WriteLine(reader.GetString(0));
reader.Close();

cmd.CommandText = "INSERT INTO Item VALUES ('555555','CD "
                           + "player',10)";      // Note 4
cmd.ExecuteNonQuery();
cmd.CommandText = "UPDATE Item SET Quantity = 12 "
        + "WHERE Description = 'CD player'";
cmd.ExecuteNonQuery();
Console.WriteLine();
Console.WriteLine("   Added and updated a new item");
Console.WriteLine("Description");
cmd.CommandText = "SELECT Description FROM Item";
reader = cmd.ExecuteReader();
while(reader.Read())
  Console.WriteLine(reader.GetString(0));
reader.Close();
cmd.CommandText =
  "DELETE FROM Item WHERE Description = 'CD player'";
cmd.ExecuteNonQuery();
cmd.CommandText = "SELECT Description FROM Item";
```

```
    reader = cmd.ExecuteReader();
    Console.WriteLine();
    Console.WriteLine("   Deleted the new item");
    Console.WriteLine("Description");
    while(reader.Read())
      Console.WriteLine(reader.GetString(0));
    reader.Close();
    con.Close();
  }
}
```

```
Made the connection to the Sales database
   Names and Addresses of Customers
Name                Address
Carla Kahn          55 Fourth St.
Darnell Davis       33 Second St.
Fred Flynn          22 First St.
Marla Martinez      44 Third St.

   Order items for radios
OrderNumber         Quantity          UnitPrice
1                   4                 27
3                   3                 27
5                   2                 27

   Customer placing orders on Mar 22, 1999
Darnell Davis

   Customers ordering computers
Carla Kahn
Fred Flynn

   Order numbers of orders from 4/1/99 to 4/30/99
1
4

   Added and updated a new item
Description
radio
television
computer
CD player

   Deleted the new item
```

```
Description
radio
television
computer
```

Note 1:

```
Console.WriteLine("Name\t\tAddress");
```

We embed tab characters, `\t`, in the string to space the data horizontally.

Note 2:

```
reader.Close();
```

We close the reader after each query to continue using the connection.

Note 3:

```
while (reader.Read())
Console.WriteLine("{0}\t\t{1}\t\t{2}",
    reader.GetString(0),
    reader.GetInt32(2), reader.GetDecimal(3));
```

We omitted field 1, `ItemNumber`, from the display because we selected all results to have `ItemNumber` equal to 222222. We use the `GetInt32` method because field 3, `Quantity`, has SQL type `INTEGER`.

Note 4:

```
cmd.CommandText =
 "INSERT INTO Item VALUES ('555555','CD player',10)";
```

We add a new row to illustrate the `UPDATE` and `DELETE` statements which change the database. We update the new row and then delete it, leaving the database unchanged when we exit the program. This is nice while learning because we can try various `SELECT` statements repeatedly, running the same program without changing the data.

The BIG Picture

When querying the database, a result of the query contains the selected rows. We use methods such as `GetString` to display a value from a resulting row. The SQL types have corresponding C# methods, so the C# `GetInt32` method retrieves `INTEGER` values, for example. We can write our SQL queries to order the results or to eliminate duplicate rows. A query may have to join several tables on common fields to obtain the desired information.

✓Test Your Understanding

11. Write an SQL statement to find names of salespersons and the customers that have placed orders with them. Be sure to eliminate duplicates.

Try It Yourself ➤

12. Modify Example 15.3 to list `CustomerID` in addition to `CustomerName` and `Address`. Arrange the output rows so the `CustomerID` numbers appear in numerical order.
13. Write a `SELECT` statement to find the names and addresses of customers who placed orders with Peter Patterson. Be sure to eliminate duplicates.

15.4 DATABASE INFORMATION AND AGGREGATE FUNCTIONS

.NET allows us to get information about the database with which we are working and about the results of queries. We can use SQL functions to compute with the data.

Database Information

The `GetOleDbSchemaTable` method returns information about the data set. This method has two parameters

```
Guid schema
Object[] restrictions
```

The `Guid` is a 128-bit identifier that is unique across all computers and networks. For this method we use `OleDbSchemaGuid` values. The two we illustrate are `Tables` and `Columns`. Using `Tables` will return a table that describes each table in the data set. Using `Columns` will return a table that describes each column.

The second argument places restrictions on the source data set. Each `OleDbSchemaGuid` has four available restrictions. For `Tables` and `Columns` these restrictions are:

Tables	Columns
TABLE_CATALOG	TABLE_CATALOG
TABLE_SCHEMA	TABLE_SCHEMA
TABLE_NAME	TABLE_NAME
TABLE_TYPE	COLUMN_NAME

Microsoft Access does not use a catalog or a schema, so we pass `null` for these restrictions. Example 15.4 shows how to set the last two restrictions.

Creating a Data Set

In this chapter we present a simple introduction to databases using C#. Our examples connect to the `Sales` data, do some processing, and close the connection. More generally, we can create a `DataSet` and process the data offline. We illustrate that in the next example by creating a `DataSet` to represent the results of a query. We can then get information about the data set of query results.

Aggregate Functions

Aggregate functions compute values from the table data, using all the rows to produce the result. For example, the query

```
SELECT SUM(BalanceDue), AVG(BalanceDue),  MAX(BalanceDue)
FROM Customer
```

returns the sum, average, and maximum of all the balances due in the customer table. These functions operate on the `BalanceDue` column for all rows in the `Customer` table. Using a `WHERE` clause, as in

```
SELECT COUNT(*), MIN(Quantity) FROM OrderItem
WHERE ItemNumber = '222222'
```

will limit the computation to the rows of the `OrderItem` table that correspond to orders for radios. The function `COUNT(*)` will return the total number of rows satisfying this condition. `MIN(Quantity)` returns the minimum quantity of radios ordered in one of the three rows of the `OrderItem` table that represent orders for radios (item number 222222).

EXAMPLE 15.4 ■ DatabaseInfo.cs

```
/* Illustrates methods for getting information about
 * the data and SQL aggregate functions.
 */

using System;
using System.Data;
using System.Data.OleDb;
public class DatabaseInfo {
 public static void Main () {
   String connect = "Provider=Microsoft.JET.OLEDB.4.0;"
    + @"data source=c:\booksharp\gittleman\ch15\Sales.mdb";
   OleDbConnection con = new OleDbConnection(connect);
   con.Open();
   Console.WriteLine
    ("Made the connection to the Sales database");

   Console.WriteLine
     ("Information for each Sales table contains:");
   DataTable tables = con.GetOleDbSchemaTable
        (OleDbSchemaGuid.Tables,
             new object[]{null,null,null,"TABLE"});   // Note 1
   foreach(DataColumn col in tables.Columns)
     Console.WriteLine
        ("  {0}\t{1}", col.ColumnName, col.DataType);  // Note 2

   Console.WriteLine("The Sales tables are:");
   foreach(DataRow row in tables.Rows)
     Console.Write("  {0}", row[2]);                   // Note 3
   Console.WriteLine();

   DataTable cols = con.GetOleDbSchemaTable
                         (OleDbSchemaGuid.Columns,
```

```
        new object[]{null,null,"Customer",null});      // Note 4
  Console.WriteLine
    ("The columns describing the Customer table are:");
  foreach(DataColumn col in cols.Columns)
    Console.WriteLine
       ("  {0}\t{1}", col.ColumnName, col.DataType);  // Note 5
  Console.WriteLine("The columns in the Customer table are:");
  foreach(DataRow row in cols.Rows)
    Console.WriteLine("  {0}\t{1}", row[3],
                           (OleDbType)row[11]);     // Note 6
  Console.WriteLine();

  String cmd = "SELECT * FROM Item";                   // Note 7
  OleDbDataAdapter adapter = new OleDbDataAdapter();  // Note 8
  adapter.SelectCommand = new OleDbCommand(cmd, con); // Note 9
  DataSet ds = new DataSet();
  adapter.Fill(ds, "Item");                           // Note 10
  DataTable item = ds.Tables[0];                      // Note 11
  Console.WriteLine("Table name: {0}", item.TableName);
  Console.WriteLine("Its columns are:");
  foreach (DataColumn col in item.Columns)
    Console.WriteLine
       ("  {0}\t{1}", col.ColumnName, col.DataType); // Note 12

  OleDbCommand command = con.CreateCommand();
  command.CommandText =
           "SELECT SUM(BalanceDue) FROM Customer";// Note 13
  Console.WriteLine("Sum of balances: {0:C}",
               (decimal)command.ExecuteScalar());// Note 14
  command.CommandText =
           "SELECT AVG(BalanceDue) FROM Customer";
  Console.WriteLine("Average of balances: {0:C}",
                   (decimal)command.ExecuteScalar());
  command.CommandText =
           "SELECT MAX(BalanceDue) FROM Customer";
  Console.WriteLine("Max of balances: {0:C}",
                   (decimal)command.ExecuteScalar());
  command.CommandText =
           "SELECT COUNT(*) FROM Customer";
  Console.WriteLine("Number of Customer rows: {0}",
                   (int)command.ExecuteScalar());
  con.Close();
 }
}
```

```
Made the connection to the Sales database
Information for each Sales table contains:
  TABLE_CATALOG System.String
  TABLE_SCHEMA  System.String
  TABLE_NAME    System.String
  TABLE_TYPE    System.String
  TABLE_GUID    System.Guid
  DESCRIPTION   System.String
  TABLE_PROPID  System.Int64
  DATE_CREATED  System.DateTime
  DATE_MODIFIED System.DateTime
The Sales tables are:
  Customer  Item  OrderItem  Orders  Salesperson
The columns describing the Customer table are:
  TABLE_CATALOG System.String
  TABLE_SCHEMA  System.String
  TABLE_NAME    System.String
  COLUMN_NAME   System.String
  COLUMN_GUID   System.Guid
  COLUMN_PROPID System.Int64
  ORDINAL_POSITION       System.Int64
  COLUMN_HASDEFAULT      System.Boolean
  COLUMN_DEFAULT         System.String
  COLUMN_FLAGS  System.Int64
  IS_NULLABLE   System.Boolean
  DATA_TYPE     System.Int32
  TYPE_GUID     System.Guid
  CHARACTER_MAXIMUM_LENGTH       System.Int64
  CHARACTER_OCTET_LENGTH         System.Int64
  NUMERIC_PRECISION      System.Int32
  NUMERIC_SCALE System.Int16
  DATETIME_PRECISION     System.Int64
  CHARACTER_SET_CATALOG System.String
  CHARACTER_SET_SCHEMA  System.String
  CHARACTER_SET_NAME    System.String
  COLLATION_CATALOG     System.String
  COLLATION_SCHEMA      System.String
  COLLATION_NAME        System.String
  DOMAIN_CATALOG        System.String
  DOMAIN_SCHEMA System.String
  DOMAIN_NAME   System.String
  DESCRIPTION   System.String
The columns in the Customer table are:
  Address       WChar
  BalanceDue    Numeric
  CustomerID    WChar
```

```
  CustomerName  WChar

Table name: Item
Its columns are:
  ItemNumber    System.String
  Description   System.String
  Quantity      System.Int32
Sum of balances: $1,797.00
Average of balances: $449.25
Max of balances: $1,667.00
Number of Customer rows: 4
```

Note 1:

```
DataTable tables = con.GetOleDbSchemaTable
(OleDbSchemaGuid.Tables,
      new object[]{null,null,null,"TABLE"});
```

The `GetOleDbSchemaTable` method, with `Tables` as the first argument, returns a `DataTable` containing a row describing each table. We pass null as the third element of the second argument to place no restriction on the names of the tables returned. We restrict the table type to `"TABLE"` in the fourth element so we get only the user-defined tables. The documentation shows others choices of restrictions for specialized uses. The `DataTable` returned has five rows, one for each of the five `Sales` tables.

Note 2:

```
foreach(DataColumn col in tables.Columns)
    Console.WriteLine
      ("  {0}\t{1}", col.ColumnName, col.DataType);
```

The `Columns` property returns a `DataColumnCollection` with one `DataColumn` for each column of the table. The `ColumnName` property gives us the name of the column and the `DataType` property gives us the type. This is a table with five rows, one for each `Sales` table. The nine column names in the output show the nine pieces of information we can get about each `Sales` table if we look at the data in each row.

Note 3:

```
foreach(DataRow row in tables.Rows)
    Console.Write("  {0}", row[2]);
```

The `Rows` property returns a `DataRowCollection` containing the data for each row of the table. From the previous output we see that the third column, with index 2, gives the table name, so we output the third column of the table to get the names of the `Sales` tables.

Note 4:

```
DataTable cols = con.GetOleDbSchemaTable
        (OleDbSchemaGuid.Columns,
           new object[]{null,null,"Customer",null});
```

The GetOleDbSchemaTable method, with Columns as the first argument, returns a DataTable containing a row describing each column. Passing Customer as the third element of the second argument restricts the tables to the Customer table, so we will get columns of only the Customer table. By passing null as the fourth element, we place no restrictions on the column names to be returned.

Note 5:
```
foreach(DataColumn col in cols.Columns)
    Console.WriteLine
        ("  {0}\t{1}", col.ColumnName, col.DataType);
```

We use the ColumnName and DataType properties to display the name and type of each column of this table that has a row for each column of the Customer table.

Note 6:
```
Console.WriteLine("  {0}\t{1}", row[3],
                         (OleDbType)row[11]);
```

From the previous output, we know that the column with index 3 contains the column name, and the column with index 11 contains the column type. The type is an integer which we cast to OleDbType. The WChar type represents a stream of Unicode characters. It maps to the String type. The Numeric type represents an exact numeric value with fixed precision. It maps to type Decimal.

Note 7:
```
String cmd = "SELECT * FROM Item";
```

We just found the names of the five Sales tables and the names and types of the four Customer columns. We connected to the Sales database to process the Sales data set. We use this query to illustrate how to build our own DataSet and get information about it. The DataSet will contain the results of the query.

Note 8:
```
OleDbDataAdapter adapter = new OleDbDataAdapter();
```

The OleDbDataAdapter serves as a bridge between the DataSet and the data source. We use its Fill method to populate the DataSet from the data source and the Update method to send changes back to the data source.

Note 9:
```
adapter.SelectCommand = new OleDbCommand(cmd, con);
```

We use the SelectCommand property to set the SQL statement used to retrieve data from the data source. The first argument to the OleDbCommand constructor is a String representing the text of the query. The second argument is an OleDbConnection.

Note 10:
```
adapter.Fill(ds, "Item");
```

The Fill method adds the data resulting from the SELECT query to the DataSet ds. The second argument is the table name.

Note 11:
```
DataTable item = ds.Tables[0];
```

This data set has only one table. We assign it to a DataTable variable.

Note 12:
```
Console.WriteLine
       ("   {0}\t{1}", col.ColumnName, col.DataType);
```

We output the name and type of each column of this table, which because of our query is the same as the `Sales Item` table. By using a `DataSet` we can process data offline and update the data source with the results. Processing offline reduces the load on the connection to the data source.

Note 13:
```
"SELECT SUM(BalanceDue) FROM Customer";
```

The rest of this program illustrates aggregate functions including `SUM`, `AVG`, `MAX`, and `COUNT`.

Note 14:
```
Console.WriteLine("Sum of balances: {0:C}",
           (decimal)command.ExecuteScalar());
```

We use the `ExecuteScalar` method to return a single value. It is easier than using an `OleDbDataReader`. The return type is `Object`, which we cast to `decimal` because currency represents exact values.

The BIG Picture

We use the `GetOleDbSchemaTable` method to obtain properties of the database such as the names of its tables and the names and types of the columns in a table. Aggregate functions compute values from the rows of a table.

✓Test Your Understanding

Try It Yourself ➤ **14.** Modify Example 15.4 to pass `null` as the fourth restriction to the `GetOleDbSchemaTable` method with the `TABLES` as first argument, instead of `"TABLE"` array. This will list all tables in the database, including the system tables.

Try It Yourself ➤ **15.** Modify Example 15.4 to find the names of the columns of the `Orders` tables in the `Sales` database.

15.5 STORED PROCEDURES AND TRANSACTIONS

A stored procedure lets us translate a statement to low-level database commands once and execute it many times, thus avoiding the inefficient repetition of the translation process. Using Microsoft Access we illustrate a simple form of stored procedure without named parameters.

When making changes to a database we must be very careful that we complete all steps of the transaction. It would not do to withdraw funds from one account but not have them deposited in another. Transaction processing allows us to explicitly control when changes become final, so we commit changes only when all those desired have completed correctly.

Using Stored Procedures

Often we may wish to execute a query repeatedly, using different conditions each time. The query

```
SELECT * FROM OrderItem
WHERE ItemNumber = '222222'
```

selects all order items with number 222222. The `Sales` database has three types of items. We could repeat the query to find orders for other items. For example, executing the query

```
SELECT * FROM OrderItem
WHERE ItemNumber = '333333'
```

produces the order items for televisions.

We have only three products in our database, but we might have had many more. For each product, the database system must process the SQL query, analyzing how to find the requested data from the database in the most efficient way possible. Our query is quite simple, but it could have been much more complex. Each time we execute the query with a different item number we have to process it, spending the time over and over again to find the best way to find the results that satisfy it.

The stored procedure allows the database system to process an SQL query once, determining the best way to get the results. We can then use this stored procedure over and over again with different data but without the overhead of translating it again.

We use the question mark, ?, to denote the arguments to query that we wish to change from one execution to the next. To make a stored procedure from our previous query, we write it as

```
SELECT * FROM OrderItem WHERE ItemNumber = ?
```

where the question mark stands for the item number that we will pass in. Example 15.5 illustrates how to call this stored procedure with different values. To pass multiple parameters we use additional question marks in the query. In the query

```
SELECT OrderNumber FROM Orders
WHERE OrderDate BETWEEN ? AND ?
```

the parameters represent the starting and ending dates of orders.

EXAMPLE 15.5 ■ Prepare.cs

```
/* Illustrates simple stored procedures with
 * unnamed parameters in the query.
 */

using System;
using System.Data;
using System.Data.OleDb;
public class Prepare {
```

```
public static void Main () {
  String connect = "Provider=Microsoft.JET.OLEDB.4.0;"
   + @"data source=c:\booksharp\gittleman\ch15\Sales.mdb";
  OleDbConnection con = new OleDbConnection(connect);
  con.Open();
  Console.WriteLine
    ("Made the connection to the Sales database");

  OleDbCommand cmd = con.CreateCommand();
  cmd.CommandText =
   "SELECT Quantity FROM Item WHERE Description = ?";
  OleDbParameter param = new OleDbParameter();     // Note 1
  cmd.Parameters.Add(param);                       // Note 2
  param.Value = "radio";                           // Note 3
  OleDbDataReader reader = cmd.ExecuteReader();
  Console.WriteLine("    Using a stored procedure
                      to find quantity of radios");
  while(reader.Read())
    Console.WriteLine("{0}", reader.GetInt32(0)); // Note 4
  reader.Close();

  param.Value = "computer";                        // Note 5
  reader = cmd.ExecuteReader();
  Console.WriteLine
    ("    Using a stored procedure to find "
      + "quantity of computers");
  while(reader.Read())
    Console.WriteLine("{0}", reader.GetInt32(0));
  reader.Close();

  OleDbCommand cmd1 = con.CreateCommand();
  cmd1.CommandText = "SELECT OrderNumber FROM Orders "
                   + "WHERE OrderDate BETWEEN ? AND ?";
  OleDbParameter p1 = new OleDbParameter();
  OleDbParameter p2 = new OleDbParameter();
  cmd1.Parameters.Add(p1);
  cmd1.Parameters.Add(p2);                         // Note 6
  p1.Value = new DateTime(1999,4,1);               // Note 7
  p2.Value = new DateTime(1999,4,30);
  reader = cmd1.ExecuteReader();
  Console.WriteLine
    ("    Using a stored procedure to find April orders");
```

```
    while(reader.Read())
      Console.WriteLine("{0}", reader.GetString(0));
    reader.Close();

    OleDbCommand cmd2 = con.CreateCommand();
    cmd2.CommandText = "SELECT CustomerName FROM Customer "
                     + "WHERE BalanceDue > ?";
                                                    // Note 8
    OleDbParameter p3 = new OleDbParameter();
    cmd2.Parameters.Add(p3);
    p3.Value = new Decimal(0.0);                    // Note 9
    reader = cmd2.ExecuteReader();
    Console.WriteLine("   Using a stored procedure to find "
              + "customers with non-zero balance");
    while(reader.Read())
      Console.WriteLine("{0}", reader.GetString(0));
    reader.Close();

    con.Close();
  }
}
```

```
Made the connection to the Sales database
   Using a stored procedure to find quantity of radios
32
   Using a stored procedure to find quantity of computers
9
   Using a stored procedure to find April orders
1
4
   Using a stored procedure to find customers with non-zero
   balance
Fred Flynn
Darnell Davis
```

Note 1: `OleDbParameter param = new OleDbParameter();`

We create an `OleDbParameter` object for each parameter in the query.

Note 2: `cmd.Parameters.Add(param);`

The `Parameters` property keeps an `OleDbParametersCollection` of the parameters. We add the object we just created.

Note 3: `param.Value = "radio";`

We set the value of the parameter to `"radio"` and execute the query.

Note 4: `Console.WriteLine("{0}", reader.GetInt32(0));`

We use the `GetInt32` method to find an integer value.

Note 5: `param.Value = "computer";`

Now we get the benefit of having already processed the query. We do not need to process it again. We just set the parameter to "`computer`" and execute the stored procedure.

Note 6: `cmd1.Parameters.Add(p2);`

When the stored procedure has two parameters, we create a second `OleDbParameter` and add it to the `Parameters` collection.

Note 7: `p1.Value = new DateTime(1999,4,1);`

We pass a `DateTime` instance for a date parameter.

Note 8:
```
cmd2.CommandText = "SELECT CustomerName FROM "
      "Customer WHERE BalanceDue > ?";
```

We use a relational operator in the condition.

Note 9: `p3.Value = new Decimal(0.0);`

We pass `Decimal` to represent a currency value.

Transaction Processing

Often when using a database we need to execute several statements to perform the desired transaction. For example, if a customer places a new order we will update the `Orders` table with another order, the `OrderItem` table with the items ordered, and the `Customer` table with a new `BalanceDue`. We would be unhappy if an error occurred after some but not all of these changes were made. C# allows us to manage transactions so we commit the changes to the database only when they complete without error.

The default is to commit the change as soon as we execute the update. The statement

```
oleDbTransaction trans=con.BeginTransaction();
```

changes from the default behavior to require that we explicitly commit changes using

```
trans.commit();
```

If we have already executed some updates and decide we do not want to commit them, we can roll back to the point at which we executed the last commit, undoing these changes using

```
trans.Rollback();
```

For example, if we have begun a transaction, after executing the queries

```
INSERT INTO Item VALUES (555555,'CD player',10)
```

✓Test Your Understanding

Try It Yourself ➤

19. Run Example 15.7 to execute the query that returns the names of customers who placed orders on March 22, 1999, but this time add the condition that the `OrderDate` is March 22, 1999, before the join condition that `Customer.CustomerID = Orders.CustomerID`. This shows we can enter conditions in any order.

✓

SUMMARY

- C# allows us to create database tables; insert, update, and delete data; and query a database from a C# program. Relational databases store data in tables, and each table has a key that uniquely identifies each row. As our example, we use the `Sales` database with five tables. The `Customer` table has `CustomerID` as its key. The `Orders` table has `OrderNumber` as its key, but also includes the foreign keys `CustomerID` and `SalespersonID`, which refer to entries in the `Customer` and `Salesperson` tables to eliminate duplicating the information in the `Orders` table. The `OrderItem` table has a compound key `(OrderNumber, ItemNumber)`; we need both values to identify an order item.
- Structured Query Language (SQL) provides an interface to database systems from different vendors. Users can write statements that each database will translate to process the desired request. In this text we use the `CREATE`, `INSERT`, `UPDATE`, `DELETE`, and `SELECT` statements. The `CREATE` statement defines data in a table. This statement may use data types that are valid in a particular database system. In this text we use `VARCHAR(N)`, a variable size character string of maximum size `N`, `INTEGER`, `DATE`, and `DECIMAL`.
- Once connected to the database, we use the `CreateCommand` method to create a command with an `ExecuteNonQuery` method we can use to execute SQL statements to create a new table or to insert values into a table. We could also create and populate tables using the database system, outside of C#.
- To retrieve information from the database we use the `ExecuteReader` method, which returns a `DataReader` to execute SQL `SELECT` statements. To get the fields in a row we use the `getXXX` method, where `XXX` is the type of the data. We use `GetInt32` for an `INTEGER` field and `GetString` for a `VARCHAR` field.
- The `SELECT` statement has various options, including a `WHERE` clause to add conditions, `SELECT DISTINCT` to remove duplicates, and `ORDER BY` to sort the result. A `SELECT` statement may refer to one table or may join information from several tables.
- The connection class provides a method that gives information about the database. We can find the names of its tables and the names and types of the columns of each table. We can create a `DataSet` and use these techniques to find information about it.
- Aggregate functions compute values using all the rows of the table. We use `SUM`, `MAX`, `MIN`, `AVG`, and `COUNT` in our examples. Stored procedures allow us to pass arguments to a statement to reuse it without having to repeat its translation to an efficient implementation in the database system. Transactions permit us to rollback SQL commands in the event the whole sequence did not complete successfully.

- Our case study builds a graphical user interface for the `Sales` database, allowing users to specify various parts of a `SELECT` statement and execute it.

PROGRAM MODIFICATION EXERCISES

1. Modify Example 15.2 to read the data from a file to insert into the tables.
2. Modify Example 15.7 to use the most appropriate `getXXX` method rather than the `GetString` method referred to in Note 15.
3. Modify Example 15.7 to allow `>=`, `<=`, `>`, and `<` operators in addition to `=`.
4. Modify Example 15.7 to check that exactly two columns, from different tables, have been selected when the user presses the `Join` button.
5. Modify Example 15.7 to add a `Checkbox` to require that the query removes duplicates from the result.
6. Modify Example 15.7 to check that exactly one column has been selected when the user presses the `Enter Value` button.
7. Modify Example 15.7 to add column headings in the output.
8. Modify Example 15.7 to allow the user to keep executing queries.
9. Modify Example 15.2 to create a `Sales1` database that is like `Sales` except that it has `LastName` and `FirstName` fields, instead of `CustomerName`, in the `Customer` table.

PROGRAM DESIGN EXERCISES

10. Write a graphical user interface for the `Sales` database that lists all customer names in one `ComboBox` and all products in another. When the user selects a customer name and a product and presses the `Submit` button, display a list with the customer name, product, quantity, and date of orders by customers with that name for that product. Use stored procedures wherever possible.
11. Write a graphical user interface for a salesperson using the `Sales` database. The salesperson should be able to enter new orders. Rollback the order if, after part of an order has been entered, a part of the order cannot be filled because of insufficient quantity of a product.
12. Develop an `Account` database to use with an electronic banking system. Provide a user interface for a client to transfer funds from one account to another. The user should be able to select the source and target accounts and enter an amount to transfer.
13. Design and populate a database for a car rental system. Allow the client to check availability of a category of car and to make reservations.
14. Design and populate a database for a record collection. Provide a screen for the collection's owner to add and remove items, to change entries, and to search.
15. Design and populate a database for sports records. Use an almanac or search the Web for sample data. Provide a screen for the user to add and remove items, to change entries, and to search.

OBJE(