CHAPTER 8
ACCESSING DATA
PROCESSING DATABASES

The programs we have written so far have demonstrated Visual Basic .NET's controls and programming language, as well as application design considerations and good programming techniques. But as business application programs they fall short because of their limited ability to access data. To get data into these programs we either included the data as part of the program itself, or we designed the program so that users could key in information. But the data that even a small business must process into reports, customer invoices, or employee paychecks are far too extensive to be coded into a program or to be rekeyed by a human every time the program is run.

The way to efficiently manage data is to store it in files on disk. The data files and program are separate. When the program is executed, the user provides the name of a file that contains data to be processed, and the program reads and uses these data. Since data files are not part of any program, they never appear in a Visual Basic .NET project window. A data file simply exists on disk, ready to be used by any program that can correctly interpret its contents.

This practice is superior to making data part of the program for two reasons. First, the program need not be modified when the data change—only the data file needs to be modified. Second, different programs can share the same data files; this eliminates the need to replicate the data, and hence reduces data redundancy. Data redundancy has several negative effects, including excessive storage, the need to update data in more than one place, and the possibility that not all duplicate data items will get updated, resulting in inconsistent values for the same data item.

Managing large amounts of related data that are processed by different programs can be difficult unless the data and relationships are organized carefully. Most businesses achieve this by placing their data in a database. **A *database* is an organized collection of data describing entities (things) and relationships that exist between entities.** (Relationships and entities will be defined and discussed in detail in this chapter.) You can think of a database as a large data file on disk that contains related data organized in a particular way. An example of a noncomputer "database" is the phone book, which stores names, addresses, and phone numbers in a structured manner for many individuals; if you imagine such a list on a computer disk, you will begin to understand the arrangement and order of data in a database.

A business typically uses a program called a *database management system (DBMS)* to create and maintain a database, which contains the company's data. After the database has been created, many different programs can access the data it contains. Commercially available DBMSs include Microsoft Access, Micrsoft SQL Server and Oracle Corporation's Oracle.

Because databases are so widely used, Visual Basic .NET provides tools for developers to build database-access capability into their programs. In this chapter we explore some of these tools. To give you the background needed to understand how programs could work with databases, we begin with a discussion of database structure and use. We then turn to a discussion of ADO.NET that provides the classes to support database access within the Microsoft .NET framework. Visual Basic.Net uses the various

classes defined in ADO.NET to access and modify databases. Following this, we discuss how to write programs in Visual Basic .NET that access databases using ADO.NET objects. Finally we look at several helpful tools available within Visual Basic .NET that help us work with various database components.

We address two related topics in appendixes: Appendix C talks about configuring and using Microsoft's MSDE Server and Appendix D provides a brief introduction to the SQL Select statement.

Objectives

After studying this chapter you should be able to

- Understand and use relational databases.
- Understand a subset of the ADO.NET object model including data providers, data adapter, connections, and data sets.
- Apply the ADO.NET object model within Visual Basic .NET to access databases.
- Use the properties and methods of the DataGrid control to display records from a DataSet.
- Write SQL Select queries to extract data from a database.
- Bind controls such as TextBoxes and Labels to a DataSet.
- Create Master/Detail DataSets.
- Use several tools within Visual Basic .NET to manage and use databases.

8.1 A RELATIONAL DATABASE PRIMER

The first step in writing programs that access data in a database is to understand how the database is arranged, and this section discusses these fundamentals. Since most databases in current use share a common structure, once you master the general principles you can quickly surmise what you need to know about any database in order to write programs that work with it.

Entities and Relationships

Every database contains two kinds of information:

1. ***Entities*, which are any of the things of interest to a business and about which the business collects data,** such as products, employees, suppliers, customers, purchases, and sales.
2. ***Relationships*, which express real-world associations between entities.** Examples of relationships among entities are products purchased by customers, and employees responsible for particular sales.

Because a database can include a large number of entities and relationships, database designers often use an ***entity-relationship diagram* (ERD) to document a database's structure.**

The simple ERD in Figure 8.1 shows three entities: Publisher, Title, and Author. The data stored about a Publisher might include a publisher identifier, name, and address. The data stored about a Title might include the International Standard Book Number (ISBN), title, and publisher. The data stored about an Author might include an author identifier, name, and year born.

<insert old figure 8.1>

Figure 8.1 Entity-relationship diagram

A relationship is represented in an ERD by a line joining two entities. The symbols at the ends of the line show an important fact about the relationship: "how many" of the entity at one end can be related to "how many" of the entity at the other end. The term ***cardinality*** is used to describe **the number of one entity that can be related to another entity**. Figure 8.2 shows the cardinality symbols and their meanings.

<insert old figure 8.2>

Figure 8.2 ERD symbols used to show cardinality

To interpret a relationship appearing in an ERD, you must read it in both directions: once from left to right, and once from right to left. When you read from left to right, ignore the symbol at the left end of the line, and when you read from right to left, ignore the symbol at the right end of the line.

Let's interpret the relationship between Publisher and Title in Figure 8.1. When read from left to right, it means "a publisher can be related to zero or more titles." When read from right to left, it means "a title can be related to one publisher." Combining the interpretations from reading in both directions, we say that this is a one-to-many relationship. In the abstract language of the ERD, it says "one publisher is related to many titles and a title is related to one publisher." Common sense tells us that it expresses the real-world fact that any given publisher may publish many different titles, but any given title is published by a single publisher.

How about the relationship between Title and Author in Figure 8.1? When read from left to right it means "a title can be related to one or more authors." When read from right to left it means "an author can be related to one or more titles." This is a many-to-many relationship. The diagram indicates that any given title has at least one author (maybe more) and that any given author has written at least one title (maybe more).

Relational Database Tables

A database stores both the data describing the entities and the relationships that exist between the entities. Several approaches are used to store this information but the most common is the relational database. A ***relational database*** **stores the data for each entity in a table with rows and columns**. We use the relational database approach in this chapter because of its popularity and because it is compatible with Visual Basic .NET.

Figure 8.3 shows an example table that stores data about the Author entity. Note the alternative terms for the parts of relational database tables. The columns in the table are called either fields or attributes. The rows are referred to as either rows or records.

<insert old figure 8.3>

Figure 8.3 Author table in a relational database

Each table in a relational database can have a ***key field*****, which consists of a specific field or combination of fields guaranteed to be unique from one row to another**. In the Author table in Figure 8.3, Au_ID is the key field. Thus, you would not see two rows in this table that have the same value of Au_ID. **A key field that is a combination of two or more fields is called a *compound key***.

From the ERD in Figure 8.1 we know that the Author entity and the Title entity are related to each other. Observe in Figure 8.3 that the Author table contains only data about authors; specifically, it does not contain any data that establish the relationship between

"authors" and their "titles." The way we store data about a relationship depends on the cardinality of the relationship. We will return to this issue shortly.

Figure 8.4 shows example data from the Publisher and Title tables.

<insert old figure 8.4>

Figure 8.4 Some records from the Publisher and Title tables

Notice that the field PubId exists in both of these tables. In the Publisher table this field uniquely identifies the records; that is, each publisher has a different PubId. Therefore, PubId is the Publisher table's key field. In contrast, in the Title table you can see more than one title with the same PubId. The PubId field is a nonkey field in the Title table.

One-to-Many Relationships. When **one table's key field appears in a second table and is not the key field in the second table, it is called a *foreign key* in the second table**. Foreign keys link, or associate, the rows in the two tables. Specifically, foreign keys implement one-to-many relationships. The relationship is established by placing the key field of the "one" entity's table into the "many" entity's table as a foreign key.

Study the placement of the key and nonkey fields in the Publisher and Title tables. The first row of the Title table shows a book titled Guide to ORACLE published in 1990 by PubId 13. Using this value for PubId, we search the Publisher table to find a match. We find the match at row 13 and discover that the publisher is McGraw-Hill. The common field (PubId) allows us to link the two rows of the two tables; we can then use the combined fields of both rows to produce reports or to answer user questions about books and publishers.

Since PubId is unique in the Publisher table, we would expect to find only one match when we search this table for PubId = 13. This agrees with the ERD, which showed that a title can be related to one publisher. But if we search the Title table for PubId = 13, we find many matches. This also agrees with the ERD, which showed that a publisher can be related to many titles.

Many-to-Many Relationships. We can also link tables to establish many-to-many relationships. If you scan the Author table (Figure 8.3), you'll see that it contains no information to associate it with the Title table. Similarly, the Title table (Figure 8.4b) contains no information to associate it with the Author table. Many-to-many relationships are not established by foreign keys. So how do we implement a many-to-many relationship? The answer is by constructing an entirely **new table, called a *correlation table*, or *intersection table*, which contains the key fields from both tables for the entities in the many-to-many relationship**.[1] Figure 8.5 shows such a table, named Title/Author.

<insert old figure 8.5>

Figure 8.5 Title/Author table used to support a many-to-many relationship

Verify the many-to-many relationship by looking down the ISBN column of the correlation table in Figure 8.5. Note that book 0-0702063-1-7 is associated with three authors (26, 65, and 104). Similarly, looking down the Au_ID column you can see that author 59 is associated with two titles (0-0704077-5-4 and 0-0704079-9-1).

[1] Correlation tables are described in Shlaer and Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data* (Yourdon Press, 1988).

Some correlation tables include additional data fields. These additional fields contain facts related to both entities. For example, in the Title/Author table we might want to store the royalties paid to each author for a specific book. In this way we could record different royalty amounts for each author/book combination. A **correlation table that includes additional fields beyond the two key fields is called an *associative object***.

Normalized Databases

This text will not teach you to design databases. You will, however, have to write code to access data from databases, so you will encounter foreign keys, correlation tables, and associative objects. You may be curious as to how database designers decide where to place information. **The process of deciding what data goes into each table is based on the desire to eliminate or reduce potential problems, and is called *normalization***.[2]

One problem that normalization solves is excess data redundancy, that is, storing the same information more than once in a database. For example, suppose you are storing employee addresses. Without proper planning, you might store this information once in a payroll table and again in a human resources table. Obviously this redundancy requires extra storage space, but more importantly, it can lead to data inconsistencies. If an employee moves and notifies the payroll office, payroll will correct the address. However, unless the payroll department notifies the human resources department, the old address may remain in the human resources table. Now there are two different addresses for the same person.

The precise process associated with normalization is beyond the scope of this text. However, a simple rule can be used to judge whether a table is in satisfactory form, known as ***third normal form*: a table is in third normal form if the nonkey fields depend on the key field, the whole key field, and nothing but the key field**.[3]

In a normalized database each nonkey field must be determined only by the key field. A table with Employee Number (key field), Employee Name, Department Code, and Department Name is not in third normal form because Department Name can be determined by the Department Code (which is not the key field). That is, if you know what the Department Code is, you can tell what department you are talking about, so you know the Department Name. We would normalize this table by splitting it into two tables: one table with Employee Number (key field), Employee Name, and Department Code, and a second table with Department Code (key field) and Department Name.

In addition, if you have a compound key (two or more fields), then each nonkey field should be determined by all the fields that make up the compound key, not just some of them. For example, a table with Student Number and Course Identifier as the compound key, and Student Name and Grade as nonkey fields, is not in third normal

[2] The inventor of relational databases, Edgar F. Codd, coined the term *normalization*: "We all have trouble organizing even our personal information. Businesses have those problems in spades. It seems to me essential that some discipline be introduced into database design. I called it normalization because then-President Nixon was talking a lot about normalizing relations with China. I figured that if he could normalize relations, so could I." [Matthew H. Rapaport, "A 'Fireside' Chat," DBMS 6, no. 13 (1993): 54–60.]

[3] Data are said to be unnormalized (bad), in first normal form (1NF—better), in second normal form (2NF—better yet), or third normal form (3NF—reasonably good). There are forms beyond 3NF but they generally solve relatively rare problems within the data.

form. The problem is that the Student Name can be determined by just part of the compound key, the Student Number. Again, we would normalize this table by splitting it into two tables. The first would include the original compound key Student Number and Course Identifier, plus the Grade (which needs both fields to determine its value). The second table would include the Student Number (key field) and Student Name.

Exercise 8.1. For each of the following tables, indicate if they are normalized (3rd normal form) or not normalized. If they are not in 3rd normal form, describe why this is true. The notation TableA = { field1 + field2 + field3 }means that the table (TableA) includes three fields (field1, field2, and field3) and the underline means that field1 is the key field. Clearly indicate any assumptions you are making.

a. Employee = { EmplNo + Name + Address}
b. Product = { ProdNo + Desc + Price + QuantityOnHand }
c. Book = (BookId + AuthorID + Title + AuthorName }
d. Transcript = { StudentNo + CourseId + Date + Grade }

Exercise 8.2. Referring to Exercise 8.1's instructions, determine if the following table definitions are either normalized (3rd Normal form) or not normalized.

a. Product = { ProdNo + Desc + CustNo + CustName + Price }
b. Customer = { CustNo + Name + Address + PhoneNo }
c. Transaction = { ProdNo + CustNo + Date + Quantity }
d. SalesPerson = { EmplNo + EmplName + DeptCode + DeptName }

Database Queries

A relational database is made up of tables, each focusing on a single entity. But what happens when we want to use the database to answer questions, called queries in database terminology? Often the data needed to answer the query comes from more than one table. Thus, with a relational database, one must be able to combine data from several tables in order to provide useful information to the user.

Most relational database systems use a query language called ***structured query language (SQL)*** **to specify how to combine data in related tables and how to select only the desired data**. Visual Basic .NET uses SQL, which we discuss in greater detail in Appendix D. Let's look at a simple query now, to complete our introduction to relational databases.

Suppose that you have a database with the three tables shown in Figure 8.6. Suppose, too, that the user of this database wants to query the database by asking for a list of all students who completed BA 420. The user wants the list to include the student number (StNo), student name (StName), course identifier (CourseId), course name (CourseName), and grade (Grade). The data needed to answer this query come from all three tables. The common fields in the three tables make it possible to determine which rows are appropriate to answer the query.

<insert old figure 8.6>

Figure 8.6 Sample database with three tables

The computer answers the query by searching the tables for the data it needs. In this case, it starts out in the Course table and finds a match for BA 420 in row 2. The computer now has access to the correct course name. Then the computer searches the Transcript table row by row looking at the CourseId field. Each time it finds a match with BA 420 (rows 2, 3, and 4), it uses the value of the student number to search the Student

table. For example, the first match, found in row 2 of the Transcript table, yields student number 123. This student number is found in row 1 of the Student table and the name Jim is extracted. The computer now has all the information it needs to produce one line of output (the first student it found who took BA 420). This process continues with the second match for BA 420 in the Transcript table (row 3), and then again for the next match (row 4).

The process of searching tables and matching common fields is the general solution employed by the computer to answer relational database queries. There are additional options and types of queries, but the basic idea is the same.

It should be pointed out that using a query language is different than the procedural programming we have seen so far. With query language code, you do not tell the computer how to reach a goal, but instead pass a request to the DBMS. The DBMS then determines how to accomplish the task. The developer's role with a query is to determine how to phrase the request so that the desired result is returned by the DBMS.

8.2 AN INTRODUCTION TO ADO.NET

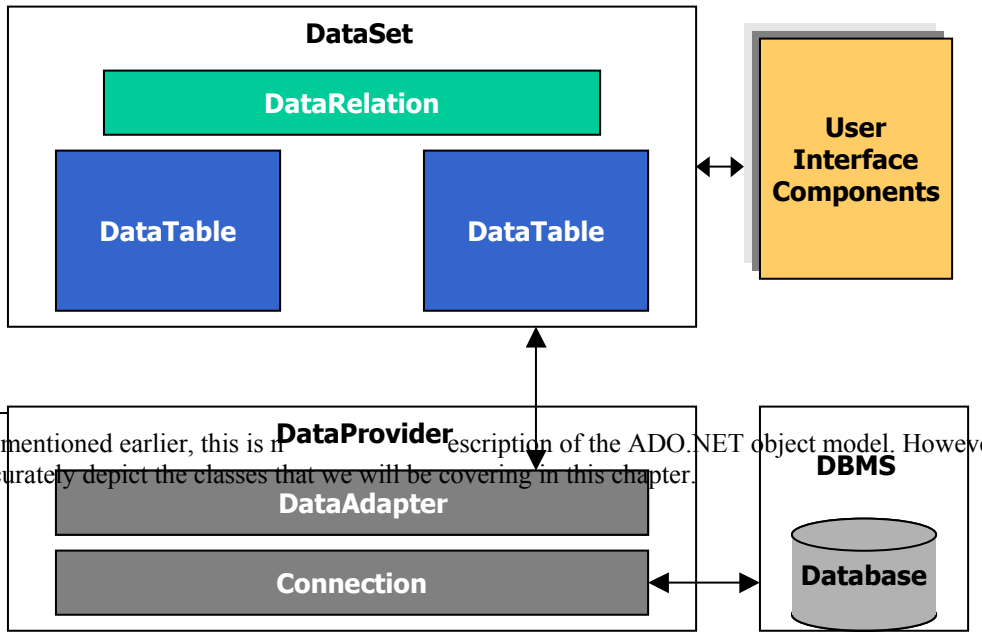
In order to manage data in a database, Visual Basic .NET uses the technology provided by ADO.NET. ***ADO.NET* (ActiveX Data Objects) is a comprehensive technology that includes an extensive set of classes to manage data**. It is very complex and entire books have been published dealing exclusively with the various aspects of the technology. Our intent in this chapter and in Chapter 9 is to provide a foundation for using the ADO.NET technology in the context of Visual Basic .NET but there will be many features that will not be covered due to extensive nature of the technology.

In this chapter we cover the foundations of ADO.NET and how this technology can be used in the context of relational databases. We also cover some of the wizards and tools available within Visual Basic .NET to work with relational databases. In Chapter 9, we cover how ADO.NET can be used to manage data in a format known as XML (e**X**tensible **M**arkup **L**anguage).

An Overview of ADO.NET

A simplified model[4] of the classes we will be using from ADO.NET is shown in Figure 8.7.

Figure 8.7 Classes to be used from ADO.NET



[4] As we mentioned earlier, this is not a complete description of the ADO.NET object model. However, it does accurately depict the classes that we will be covering in this chapter.

Let us walk through Figure 8.7 to see how everything works within the ADO.NET model. We start with the actual database. This could be a Microsoft SQL Server database, a Microsoft Access database, an Oracle database, or practically any other type of database. A Visual Basic .NET application does not actually manage the database – it calls on the appropriate DBMS by passing it commands typically using SQL (Structured Query Language). This point is important to understand because it means that our Visual Basic .NET application is relatively independent from any specific DBMS. If one decides to change from one DBMS to another, it requires changing very few statements in the Visual Basic .NET application.

As you can see in the Figure 8.7, ADO.NET uses a ***DataProvider*** **object to interact directly with database and its DBMS**. The DataProvider class has two classes to help in the interaction. The first is the Connection class. A ***Connection*** **object makes the physical connection to the database**. To do this, it needs to know information such as where the database is located and the appropriate technology that is needed to make this physical connection. The second class is the ***DataAdaptor*** class. This class's **primary responsibility is to act as an intermediary between the database and a DataSet** (discussed next). It is also responsible for managing how any changes in the DataSet should be reconciled against the actual database.[5]

In addition to the DataProvider, ADO.NET uses the DataSet class to hold the actual data. The easiest way to think about a ***DataSet*** **is as a collection of one or more tables**. In order for these tables to be related, the DataSet also has a ***DataRelation*** class. **Objects from this class include information on how two tables are related including the fields common in the two tables that are used to relate one to the other**.

User interface components such as TextBoxes, ComboBoxes, and DataGrids can work directly with a DataSet object for displaying and modifying data that is managed by the DataSet.

In summary, assume that you want to show a list of state abbreviations in a ComboBox and that these abbreviations are stored in a database. Starting with the ComboBox, it would need to be associated with a DataSet that contained a table with 50 rows, one for each state abbreviation. In order to get this data, the DataSet would need to be associated with a DataAdapter that was in turn associated with a Connection. Finally this Connection would talk directly with the DBMS where the abbreviations were originally stored.

In the sections that follow we will use a "wizard" to help us create our DataAdapter and Connection objects. We will also use Visual Basic's "Generate Dataset" tool to help build the DataSet component(s). We will begin with DataSets that have a single table and later move to more complex DataSets that have multiple tables and a DataRelation to associate the tables.

8.3 USING ADO.NET WITH VISUAL BASIC .NET

Now that we have seen the components needed to support relational database access using ADO.NET, we turn our attention to how we use this technology within Visual Basic .NET. We will look at four different applications and will be using the "pubs"

[5] There is also a Command class that is involved in this process. However, we will not be working directly with any objects from the Command class.

database that comes with Visual Basic .NET. We will also be using the Microsoft Data Engine (MSDE) that also comes with Visual Basic .NET. MSDE is a desktop version of Microsoft's SQL Server. To see information on using MSDE and to install MSDE (if not already installed) see Appendix C.

Data Access Using The DataGrid Control

The ***DataGrid*** **control provides a method of displaying records from a database in a grid-like manner**. By setting several properties of this control, it is very easy to associate it to the records in a DataSet. However, before discussing the DataGrid, we need to build a DataAdapter and DataSet. We assume that a new Windows Application project has been created. When we are done, we will have a DataGrid on our form that will display records from the publishers table in the "pubs" database.

Building the DataAdapter. Visual Basic .NET provides a wizard to help create a DataAdapter. We will take advantage of this wizard to simplify the process and minimize the amount of code we have to write.

If you look at the Data tab on the Toolbar, you will see a control labeled OleDbDataAdapter. Figure 8.8 shows this tool. Since we are using the MSDE server, we could also choose the SqlDataAdapter, which is optimized for working with SQL Server 7.0 or later. We will use the OleDbDataAdapter because it is more generic, providing ADO.NET access to any OLE DB-compatible data source

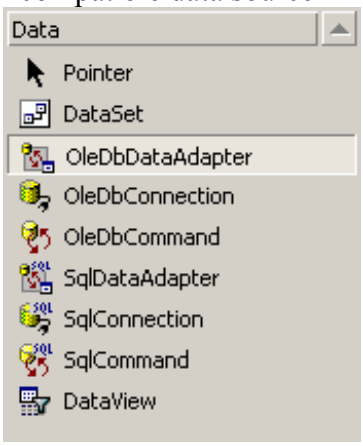


Figure 8.8 The OleDbDataAdapter Tool

If you place an OleDbDataAdapter on the form, the Data Adapter Configuration Wizard starts automatically. After displaying an Introduction page, the next page is used to establish a connection to the actual database. Figure 8.9 shows this page.

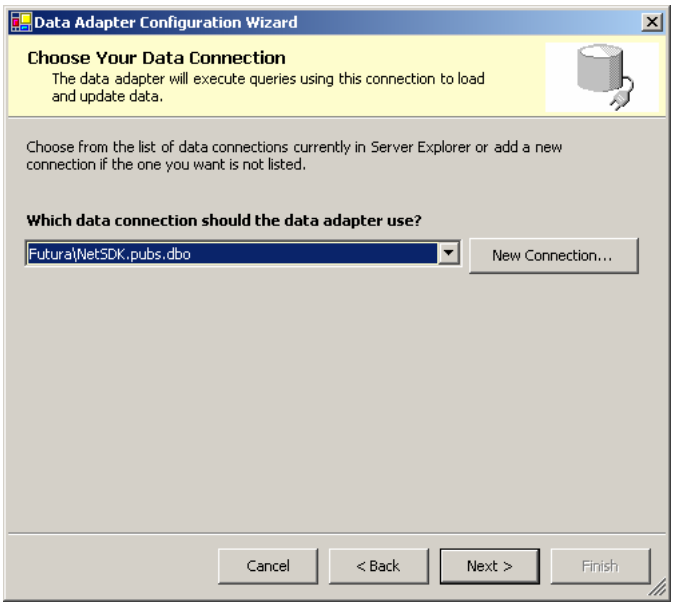


Figure 8.9 The Data Connection page of the Data Adapter Wizard

As you see in Figure 8.9, a connection already exists on the computer that the example is running on. If a connection has not been created, you need to click on the New Connection… button. Doing so brings up a dialog box like the one shown in Figure 8.10.

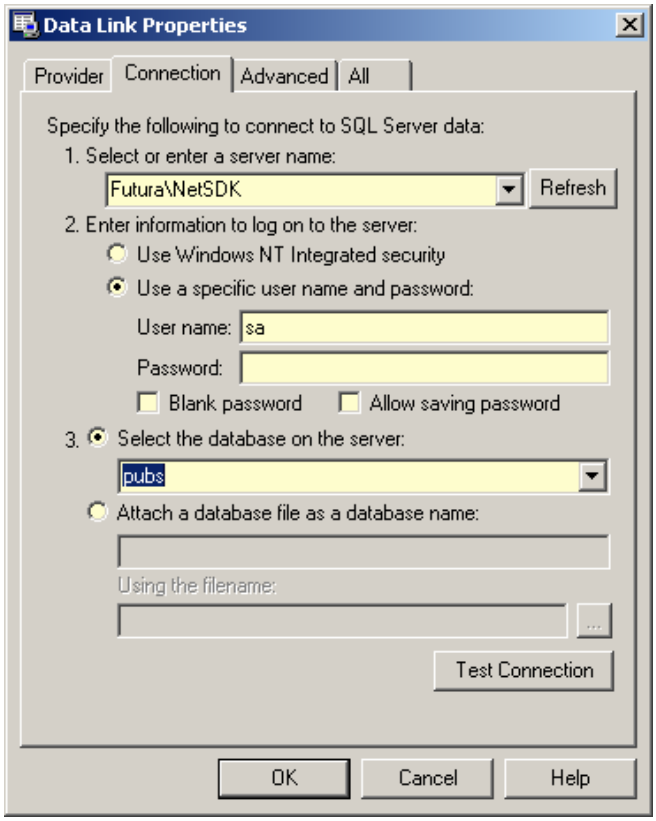


Figure 8.10 Setting up a connection to a database

As shown in Figure 8.10, you need to complete the information in three areas. The first is the name of the database server. In this example, the server is named PICO\NetSDK. The name of the server on the computer can be found by placing the cursor over the icon of the server manager on the Start bar. Figure 8.11 shows how the name of the server is displayed when the cursor is placed over the server manager icon.

Figure 8.11 Determining the name of the server

The second thing you must specify is information needed to log on to the server. You should select "Use Windows NT Integrated security" unless instructed to do otherwise by you instructor.

Finally you need to specify the database. The drop-down list shows all the databases installed on the server. You should see the database named "pubs" and you should select this and then click on OK. Note that if no databases are shown or if you see an error message, this means that something is wrong in either step 1 or 2.

After completing the Data Connection page and clicking on Next, you will see the Query Type page. As shown in Figure 8.12, you should select the "Use SQL statements" option.

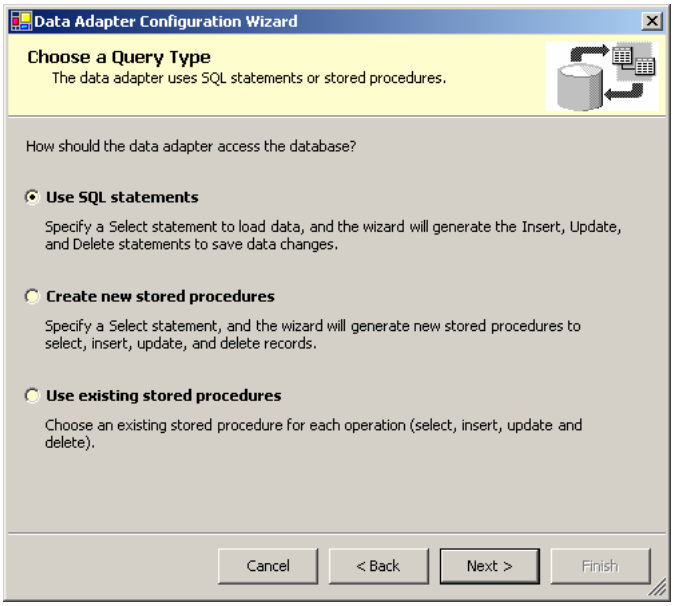


Figure 8.12 Specifying the Query Type

The next page, shown in Figure 8.13, is used to specify the SQL query. ***SQL* (Structured Query Language) is the industry standard way of specifying the data that you want to extract from the database**. Appendix D provides details on the syntax of the SQL select statement that is used to perform an SQL query. However, this wizard provides a Query Builder tool that automates the creation of an SQL Select statement.

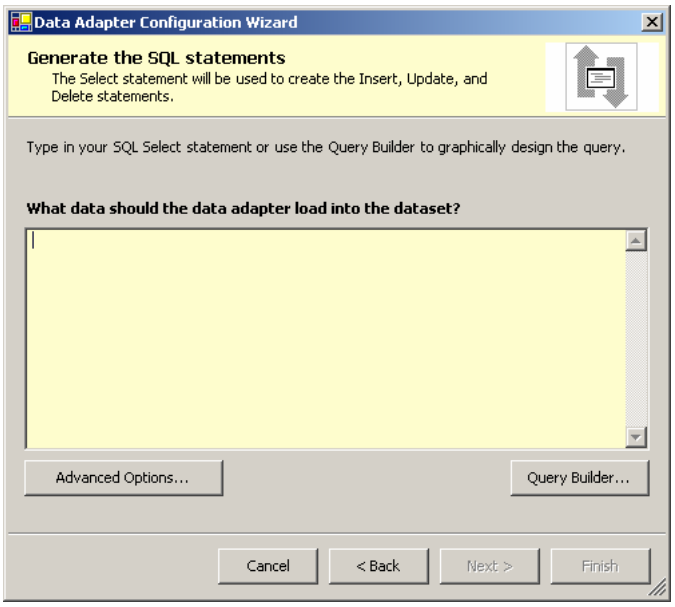


Figure 8.13 Specifying the SQL Query

If you click on the Query Builder… button, you will see the dialog box shown in Figure 8.14. In this initial dialog, you must specify the table(s) that contains the data you want to query. For this example, we want to query the "publishers" table so we select it and then click on Add and then click on Close. Later we will create a more complex query that involves data from several tables. After closing the "Add Tables" window, you need to select the fields to be retrieved by clicking on the appropriate check boxes. For this case we want all the fields so we click on the "* (All Columns)" check box (see Figure 8.15).

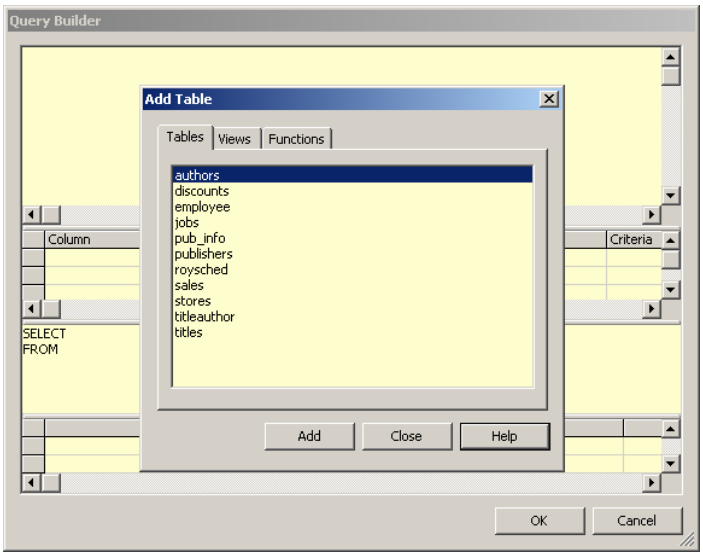


Figure 8.14 Selecting the "publishers" table to be used in a query

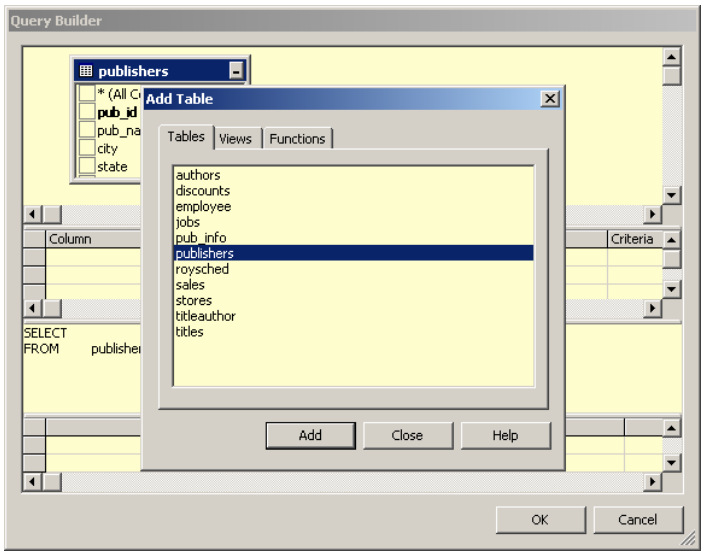


Figure 8.15 Selecting "All Columns" from the publishers table

After clicking on OK, you will see the SQL Select statement that was created by the wizard as shown in Figure 8.16.

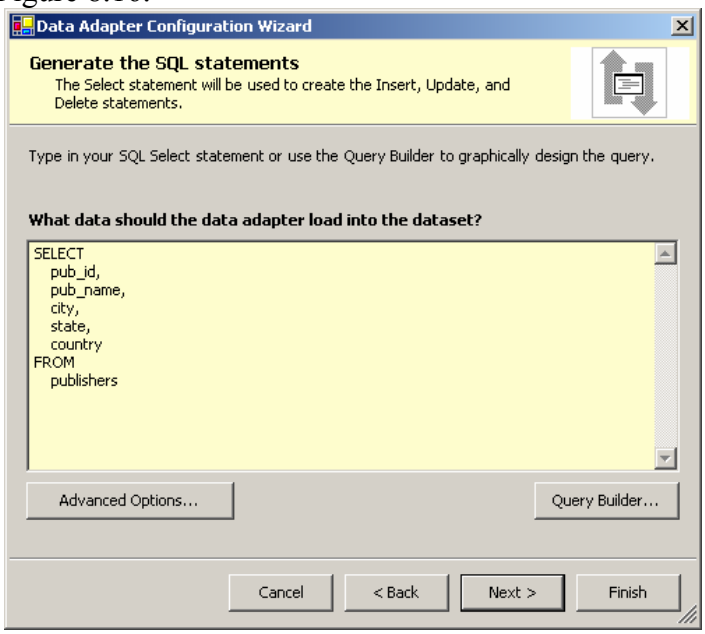


Figure 8.16 The SQL Select statement created by the wizard

Click on Next and you will see a summary of the actions performed by the wizard. As seen in Figure 8.17, the wizard generates Select, Insert, Update, and Delete statements. Thus, you have the ability to not only see the selected records, you also have the ability to insert new records and update or delete an existing record.

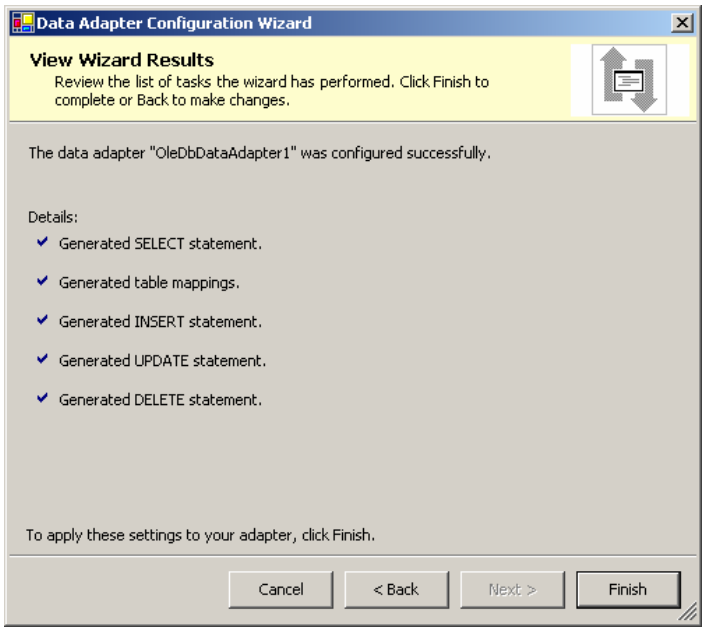


Figure 8.17 A summary of the actions performed by the wizard.

Clicking on Finish completes the process. As you see in Figure 8.18, an OleDbDataAdapter and OleDbConnection have been added to the project and are shown in the tray below the Form Designer.

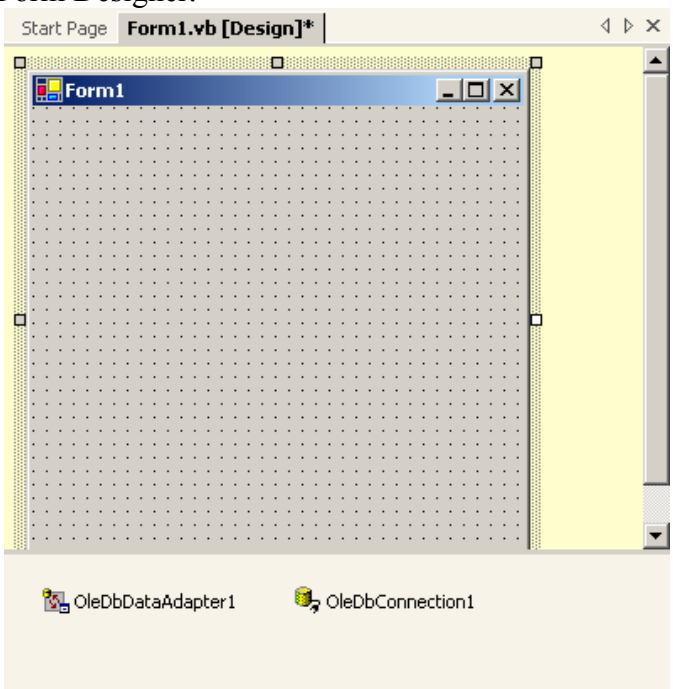


Figure 8.18 The OleDbDataAdapter1 and OleDbConnection1 added to the project

Building the DataSet. Now that we have created a DataAdapter, we need to create a DataSet. Be sure that the Form Designer is selected and then select Generate DataSet… from the Data menu. A dialog box like the one shown in Figure 8.19 will be displayed. As seen in this figure, a new dataset is being created with the name "dsPublishers". This DataSet is associated with the publishers table and SQL Select statement as defined in

OleDbDataAdapter1 that we created earlier. The CheckBox labeled "Add this dataset to the designer" should also be checked.

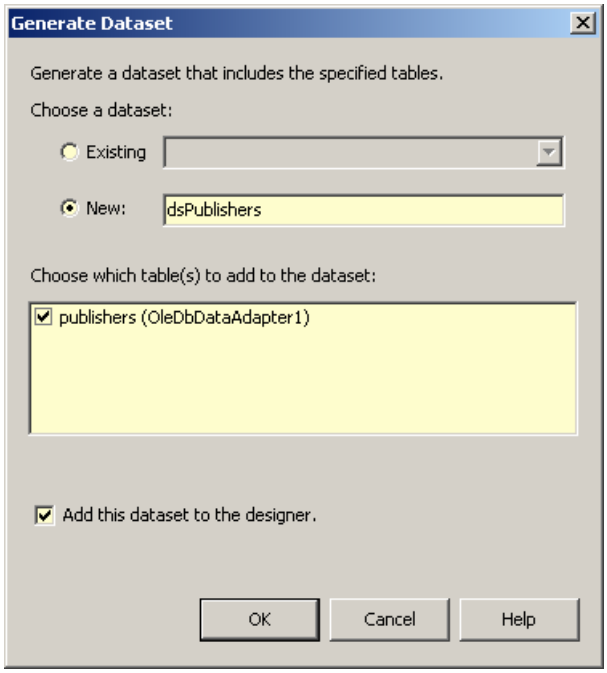


Figure 8.19 Creating a Dataset

After clicking on the "New" radio button, adding the dataset name "dsPublishers" and ensuring that the "Add this dataset…" is checked, click on OK. You should now see the DataSet added to the designer (Figure 8.20). Note that the name in the designer is DsPublishers1, not dsPublishers as shown in Figure 8.19. This is because you are creating a class named dsPublishers in the Create Dataset window (Figure 8.19). In Figure 8.20 you are seeing an object named DsPublishers1 that was instantiated from the class dsPublishers and placed in the design tray by Visual Basic .NET.

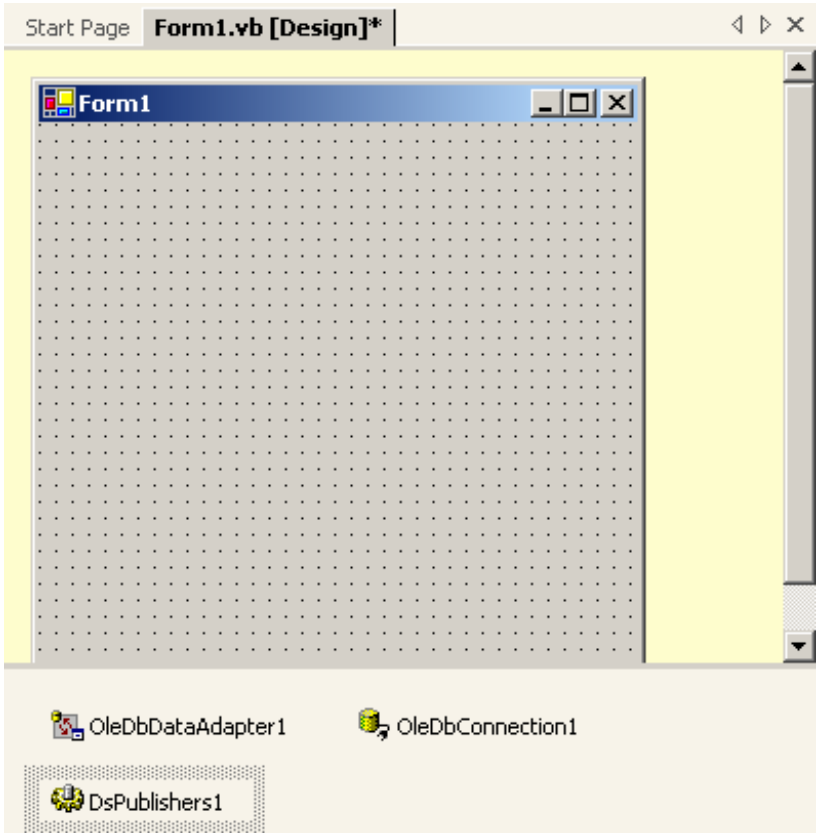


Figure 8.20 The DsPublishers1 DataSet added to the Form Designer

Adding the DataGrid. We are now ready to add a DataGrid to the form and set its properties to connect it to the DataAdapter and DataSet. The DataGrid is found on the Windows Forms tab in the toolbox as shown in Figure 8.21.

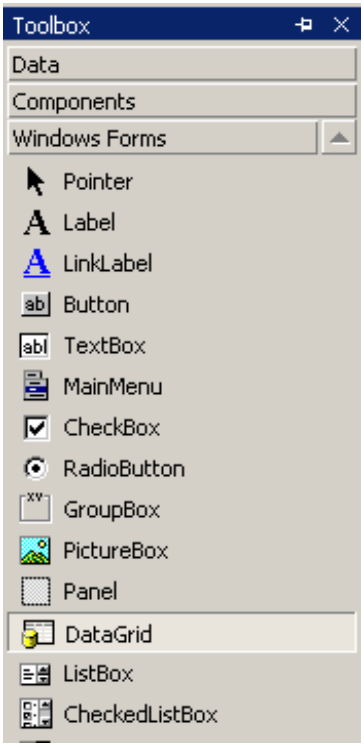


Figure 8.21 The DataGrid in the Toolbox

Place a DataGrid on the form. After doing so, set the DataSource and DataMemeber properties as shown in Figure 8.22. These two properties are the only properties you need to set to establish the relationship between the DataGrid and the actual DataSet.

<combine into one figure>

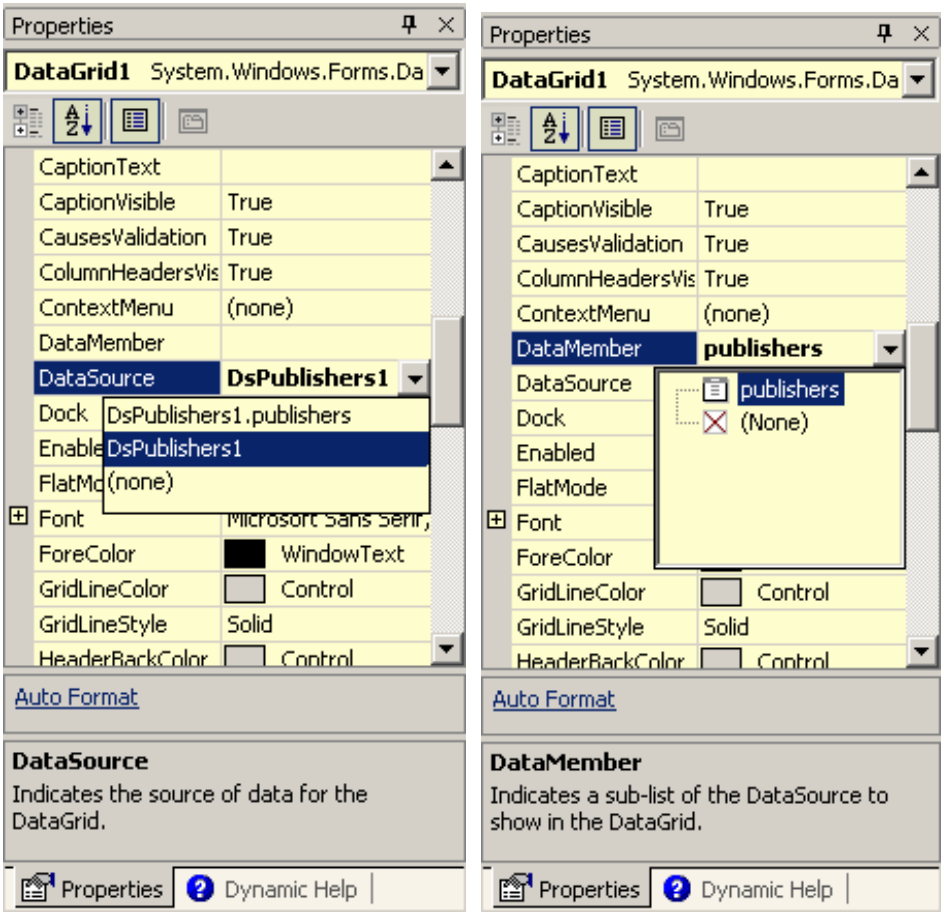


Figure 8.22 Selecting the DataSource and DataMember properties for the DataGrid.

The DataGrid on the form will show the fields you specified in the SQL Select statement as column headings (see Figure 8.23).

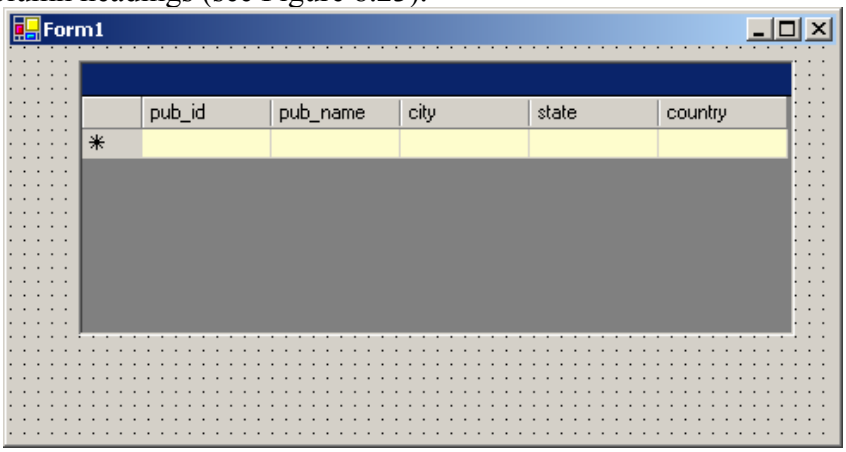


Figure 8.23 The DataGrid after setting the DataSource and DataMember properties

If you run the application at this time, you will see the DataGrid but no data will be displayed (the values "(null)" will be displayed). The reason for this is because the DataSet has not been "filled" with the data from the DataAdapter. Add a Button to the form and then create a Click event like the one shown in Figure 8.24.

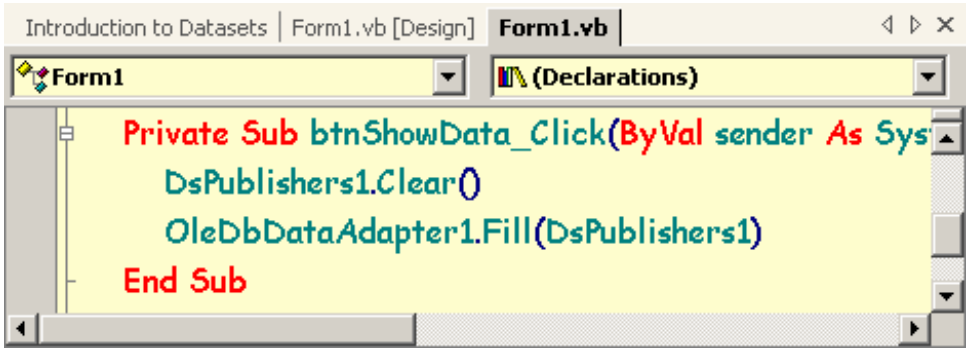


Figure 8.24 Code to fill the DataGrid with data

The statement DsPublishers1.Clear() clears the DataSet (DsPublishers1) of any data by removing all rows in all tables. Then the statement OleDbDataAdapter1.Fill(DsPublishers1) adds or refreshes rows in the DataSet to match those in the data source (database) the DataSet is associated with. When you run the application and click on the "Show Data" Button, you should see the records as shown in Figure 8.25.

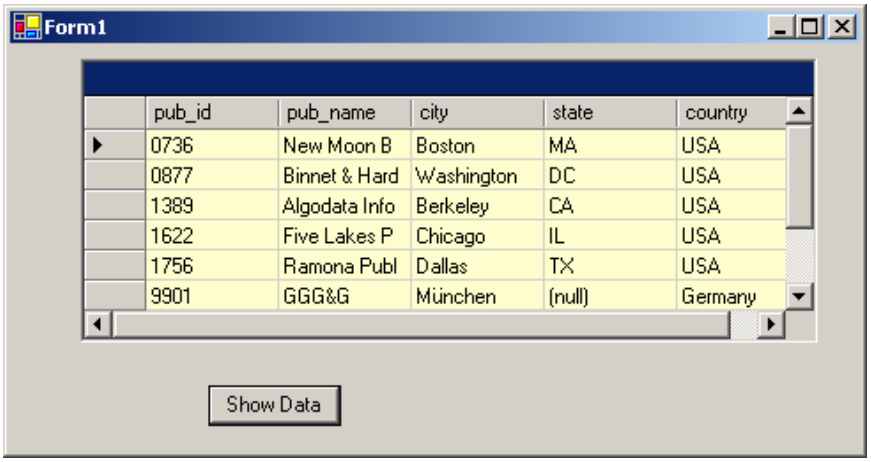


Figure 8.25 The running application with data displayed in the DataGrid

The DataGrid Control

As we just saw, the DataGrid control is designed to display the data stored in a DataSet (as well as other data sources not covered here). It is an extremely flexible control and somewhat complex with many useful properties and methods. In addition to displaying the rows from a table, if the DataGrid is bound to data with multiple related tables, the grid can display expanders in each row. An ***expander*** **allows navigation from a parent table to a child table**. Clicking on a node displays the child table, and clicking the Back button displays the original parent table. In this fashion, the grid displays the hierarchical relationships between tables. We will see an example of this later.

Properties. Some of the most common properties of the DataGrid control are shown in Table 8.x. We will illustrate many of these properties in examples.

Table 8.x Properties of the DataGrid control

Property	**Value**
Name	(grd) Gets or sets the name of the control.
AllowSorting	*Gets or sets a Boolean value indicating whether the grid can*

	be resorted by clicking on a column header.
AlternatingBackColor	*Gets or sets the background color of alternating rows for a ledger appearance.* By default, all rows have the same color (the BackColor property of the control). When you set the AlternatingBackColor to a new color, every other row is set to the new color. To reset the alternating background color to its default value, set the property to Color.Empty.
BackColor	*Gets or sets the background color of the grid*. The BackColor property determines the color of rows in the grid
BackgroundColor	*Gets or sets the color of the non-row area of the grid.* The BackgroundColor determines the color of the nonrow area of the grid, which is only visible when no table is displayed or if the grid is scrolled to the bottom, or if only a few rows are contained in the grid.
BorderStyle	*Gets or sets the grid's border style*. Available styles include BorderStyle.None, BorderStyle.FixedSingle, and BorderStyle.Fixed3D
CaptionBackColor	*Gets or sets the background color of the caption area*.
CaptionFont	*Gets or sets the font of the grid's caption*.
CaptionForeColor	*Gets or sets the foreground color of the caption area*.
CaptionText	*Gets or sets the text of the grid's window caption*.
CaptionVisible	*Gets or sets a Boolean value that indicates whether the grid's caption is visible*.
ColumnHeadersVisible	*Gets or sets a Boolean value indicating whether the column headers of a table are visible*.
CurrentCell	*Gets or sets which cell has the focus. Not available at design time.* Uses a DataGridCell class to set the current cell. For example, to set the current cell of the DataGrid grdDemo to row 1 and columns 1, you would say grdDemo.CurrentCell = New DataGridCell(1,1).
CurrentRowIndex	*Gets or sets index of the selected row*. The DataGrid uses zero-based indexing with the first row and first column having the index value 0.
DataMember	*Gets or sets the specific list in a DataSource for which the DataGrid control displays a grid*. If a DataSource contains multiple sources of data, you should set the DataMember to one of the sources. For example, if the DataSource is a DataSet that contains three tables named Customers, Orders, and OrderDetails, you must specify one of the tables to bind to.
DataSource	*Gets or sets the data source that the grid is displaying data for*.
Enabled	*Gets or sets a Boolean value indicating whether the control can respond to user interaction.*

FirstVisibleColumn	*Gets the index of the first visible column in a grid.*
ForeColor	*Gets or sets the foreground color (typically the color of the text) property of the DataGrid control.*
GridLineColor	*Gets or sets the color of the grid lines.*
GridLineStyle	*Gets or sets the line style of the grid.* Options include DataGridLineStyle.None and DataGridLineStyle.Solid.
HeaderBackColor	*Gets or sets the background color of all row and column headers.*
HeaderFont	*Gets or sets the font used for column headers.*
HeaderForeColor	*Gets or sets the foreground color of headers.*
ParentRowsBackColor	*Gets or sets the background color of parent rows*.
ParentRowsForeColor	*Gets or sets the foreground color of parent rows*.
ParentRowsVisible	*Gets or sets a Boolean value indicating whether the parent rows of a table are visible.*
PreferredColumnWidth	*Gets or sets the default width of the grid columns in pixels*. Set this property before resetting the DataSource and DataMember properties or the property will have no effect.
ReadOnly	*Gets or sets a Boolean value indicating whether the grid is in read-only mode.*
RowHeadersVisible	*Gets or sets a Boolean value that specifies whether row headers are visible.*
RowHeaderWidth	*Gets or sets the width of row headers*. The default is 50.
SelectionBackColor	*Gets or sets the background color of selected rows.*
SelectionForeColor	*Gets or set the foreground color of selected rows.*

Methods. There are a number of methods that can be used to change the appearance of the DataGrid as well as associate it to a DataSet at runtime. Table 8.x shows the common method of the DataGrid control and explains what they do.

Table 8.x Methods of the DataGrid control

Method	**Behavior**
Collapse(*row*)	*Collapses child relations, if any exist for all rows, or for a specified row.* The argument *row* indicates the number of the row to collapse. If set to -1, all rows are collapsed.
Expand(*row*)	*Displays child relations, if any exist, for all rows or a specific row.* The argument *row* indicates the number of the row to expand. If set to -1, all rows are expanded.
IsExpanded(*row*)	*Gets a Boolean value that indicates whether a specified row's node is expanded or collapsed.*

IsSelected(*row*)	*Gets a value Boolean indicating whether a specified row is selected.*
Select(*row*)	*Selects a specified row.*
SetDataBinding(*dataSourceObject, dataMemberString*)	Sets the DataSource and DataMember properties at runtime.
UnSelect(*row*)	*Unselects a specified row.*

Exercise 8.3. Create a Windows Application like the one that was just described except display all the fields in the "titles" table from the "pubs" database. Use a DataGrid to display the records from the table.

Example 8.1 Using a Complex Query

In this example we use a DataGrid to display the records from several tables. The "pubs" database includes three tables (among others) named authors, titleauthor, and titles. These tables are related to each other as shown in the ERD in Figure 8.26.

Figure 8.26 Relationships between the Autors, TitleAuthor, and Title tables

We will create a query that displays the author's last name, and phone number; the book's title, price and year-to-date sales; and the author's royalty percentage for a specific book. We begin by creating a new Windows Application project. As described previously, we need to first build an OleDbDataAdpator that is connected to the "pubs" database. When we get to "Generate the SQL statement" dialog, click on the "Query Builder" button, add the three tables, and select the six fields as shown in figure 8.27.

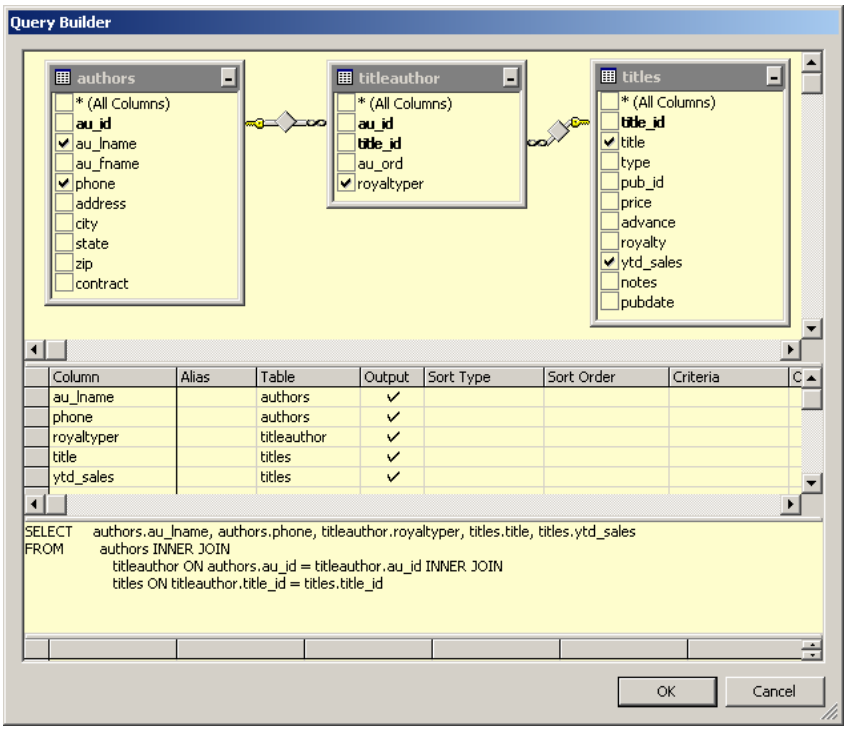


Figure 8.27 The Query Builder dialog showing the tables and fields for our query

After completing the Data Adaptor wizard, view the code generated by Visual Basic .NET for OleDbAdapter1 by right clicking on this component and then selecting View Code. Open the region titled "Windows Form Designer generated code" by clicking on the "+" sign to the left of the region box and scroll down until you see the SQL statement auto-generated by Visual Basic .NET. Figure 8.28 shows this code.

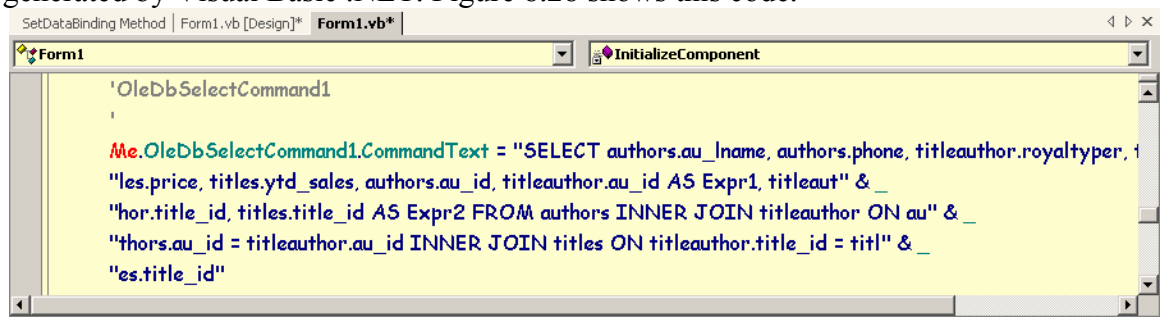


Figure 8.28 The SQL Select statement generated by Visual Basic .NET

Unfortunately this SQL Select statement is not what we want. Visual Basic .NET added four additional fields that we did not ask for.[6] We can easily correct the situation by simply removing the extra code. The code we need to delete is highlighted in Figure 8.29.

[6] Microsoft may change this either in a future Service Release or the next major release. Thus, you may not have to make the edit to the SQL described above.

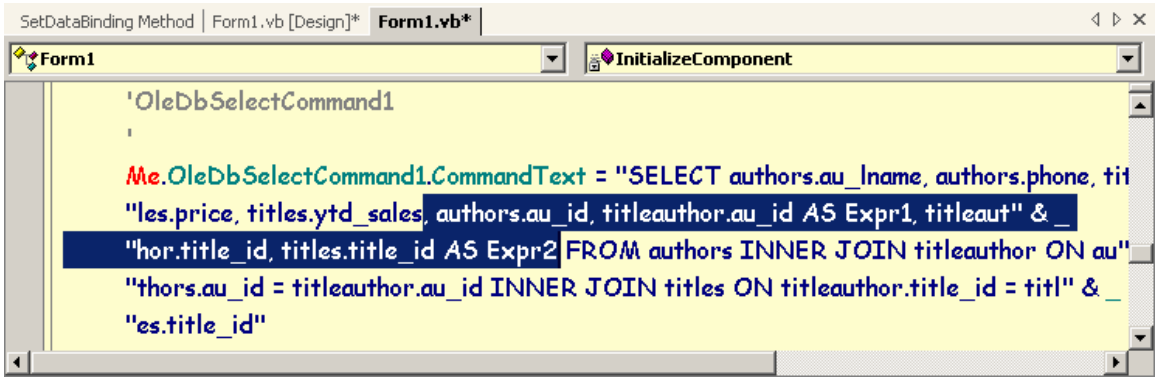


Figure 8.29 Code that we want to delete from the auto-generated SQL statement

Highlight this code and delete it. Now we need to build the DataSet. Select Generate Dataset… from the Data menu and create a New DataSet named dsAdvancedQuery (remember that you must have the form designer selected). The "authors" table should already be selected.

Add a DataGrid to the form and set its DataSource property to DsAdvancedQuery1 and its DataMember property to authors. Finally create a Form_load event and add the code shown in Figure 8.30.

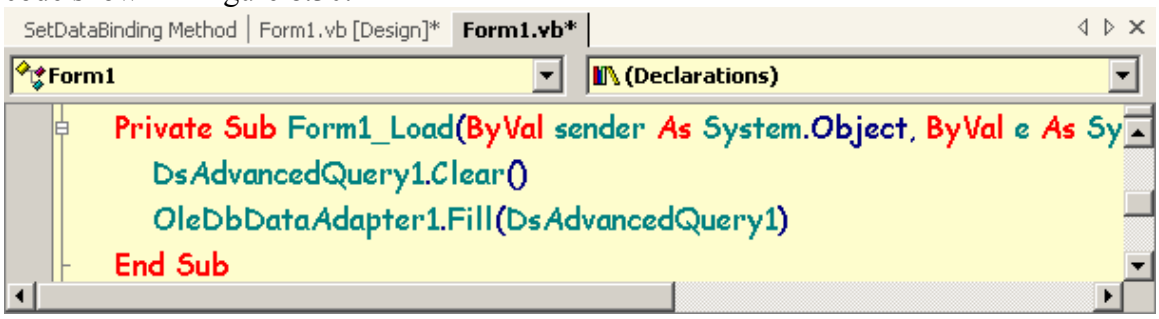
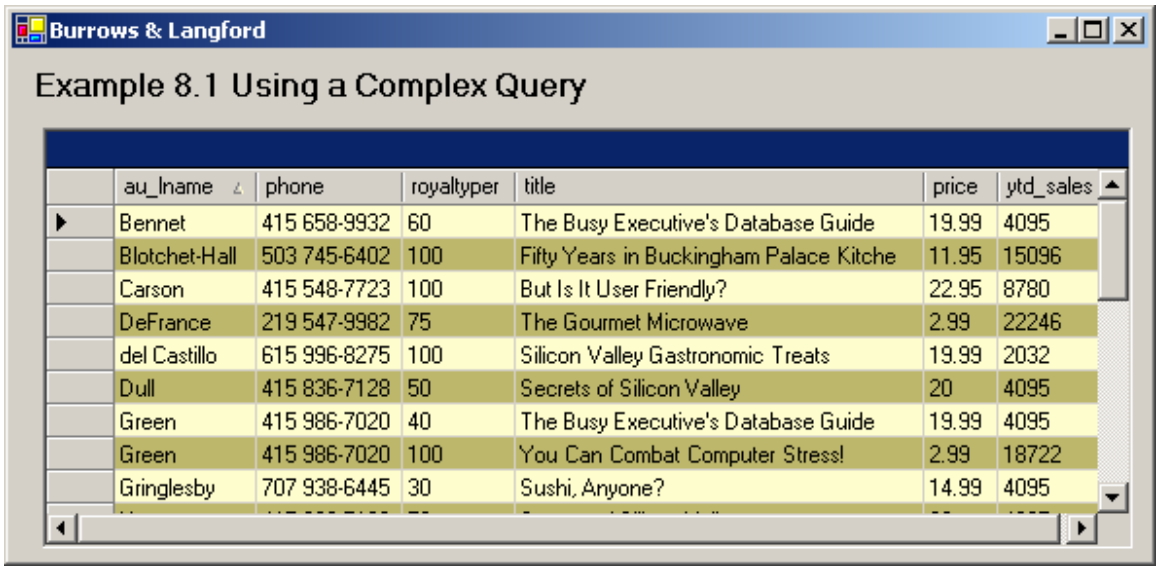


Figure 8.30 The Form_load event that populates the DataGrid

When you run the application, you should see the form shown in Figure 8.31. The DataGrid on this form has its AlternatingBackColor property set to DarkKhaki and its ReadOnly property set to True. The user has also clicked on the au_lname heading to sort the rows by this field.



au_lname	phone	royaltyper	title	price	ytd_sales
Bennet	415 658-9932	60	The Busy Executive's Database Guide	19.99	4095
Blotchet-Hall	503 745-6402	100	Fifty Years in Buckingham Palace Kitche	11.95	15096
Carson	415 548-7723	100	But Is It User Friendly?	22.95	8780
DeFrance	219 547-9982	75	The Gourmet Microwave	2.99	22246
del Castillo	615 996-8275	100	Silicon Valley Gastronomic Treats	19.99	2032
Dull	415 836-7128	50	Secrets of Silicon Valley	20	4095
Green	415 986-7020	40	The Busy Executive's Database Guide	19.99	4095
Green	415 986-7020	100	You Can Combat Computer Stress!	2.99	18722
Gringlesby	707 938-6445	30	Sushi, Anyone?	14.99	4095

Figure 8.31 Example 8.1 at runtime.

Exercise 8.4. Modify Example 8.1 so that it displays the author's state and the title's price in addition to the fields already displayed.

Exercise 8.5. Using the sales, titles, and stores tables in the pub database, create a Windows Application the uses a DataGrid to display the quantity sold at each store for each title. Include the qty, title, stor_name, and state (from the stores table) as fields in the DataGrid.

Example 8.1a Using a Complex Query Without Using the Data Adaptor Creation Wizard

In the first two applications (the simple and complex queries), we used the Data Adapter Creation Wizard to configure an OleDbDataAdapter. Some people prefer to not use Wizards because of the argument that the underlying details of the process are hidden. By hiding these details, it then becomes more difficult to set up configurations that are not supported by the Wizard.

In this variation of Example 8.1, we manually configure the data adapter using the Properties window. Once the data adapter is configured, the remaining steps of Example 8.1 continue unchanged. We will assume that a new connection must be created as well as a new data adapter.

We begin by first creating a new project and then dragging an OleDbDataAdapter to the new form. When the Data Adapter Creation Wizard appears, we click on Cancel. The form designer with the new data adapter in the designer in the tray should appear as seen in Figure 8.32.

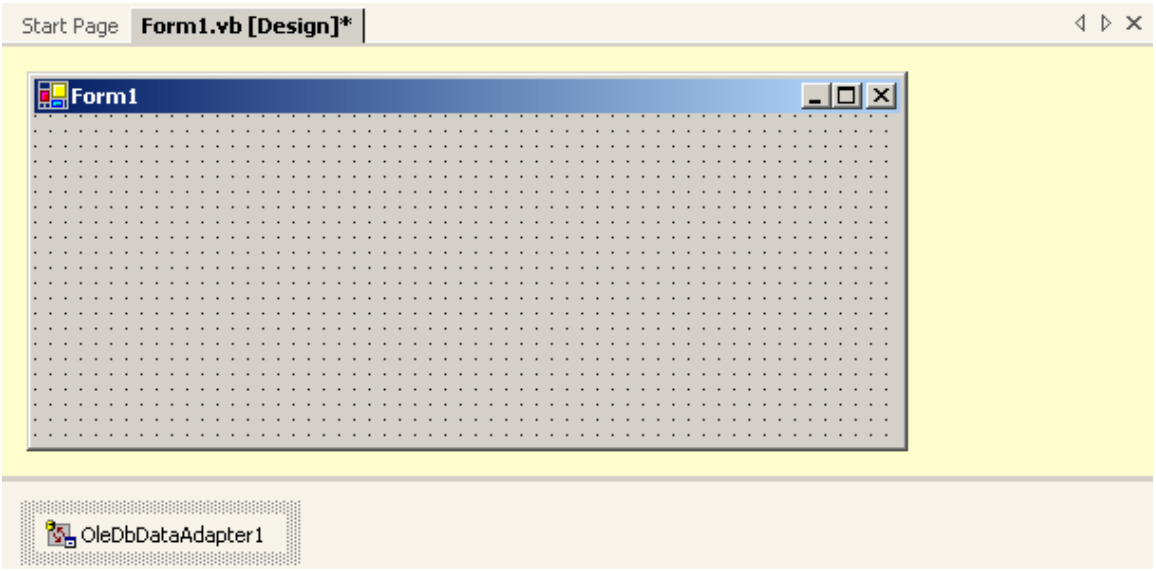


Figure 8.32 Example 8.1a with the OleDbDataAdapter added to the design tray

With the data adapter selected, we need to modify some properties in the Properties Window. The Properties Window for the OleDbDataAdapter1 is shown in figure 8.33.

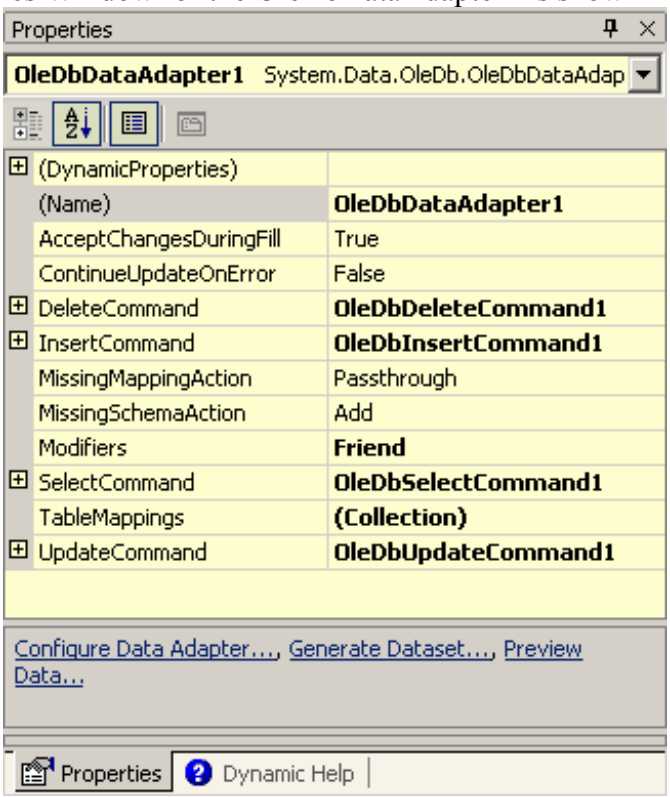


Figure 8.33 Properties Window for the OleDbDataAdapter1 component

We first change the name to odaAuthorTitles. We then click on the SelectCommand plus sign (expansion symbol) to view the SelectCommand properties. Figure 8.34 shows the expanded SelectCommand.

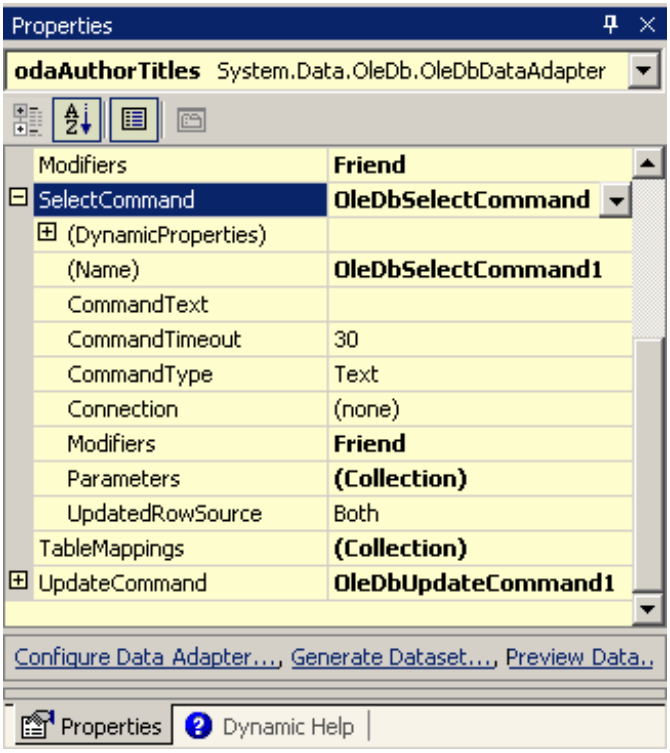


Figure 8.34 Expanded SelectCommand property

We first need to define a Connection. Click on the Connection property's value box, click on the drop-down arrow, and then select New from the list of choices. Figure 8.35 shows this drop-down list.

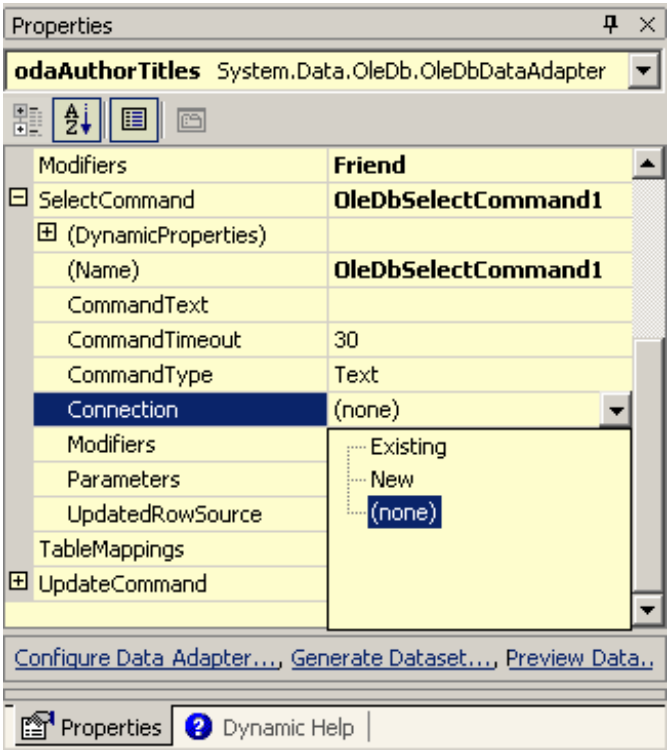


Figure 8.35 Choices available for the Connection property of the SelectCommand

After selecting New, a new Data Link Properties dialog box just like shown in Figure 8.10 will be displayed. Complete this dialog box as shown in Figure 8.36.

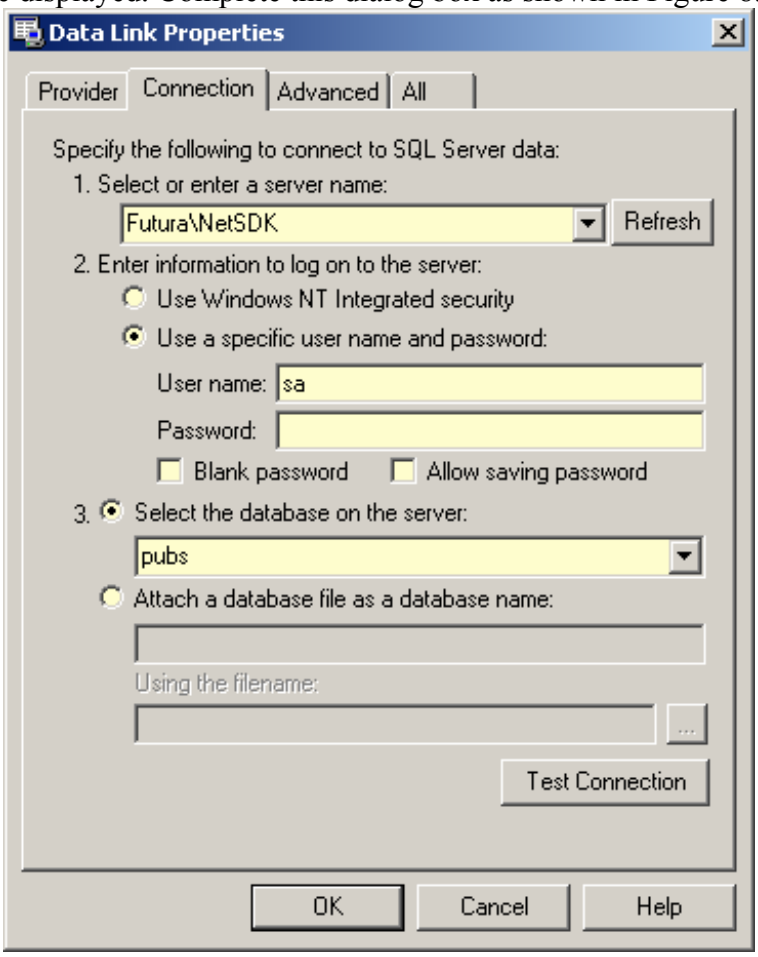


Figure 8.36 The Data Link Properties dialog box used to define the Connection

Next we need to define the CommandText property. Click on this property and click on the ellipses and a Query Builder Window will be displayed. Fill this out like that shown in Figure 8.27. When you are done it should look like that shown in Figure 8.37.

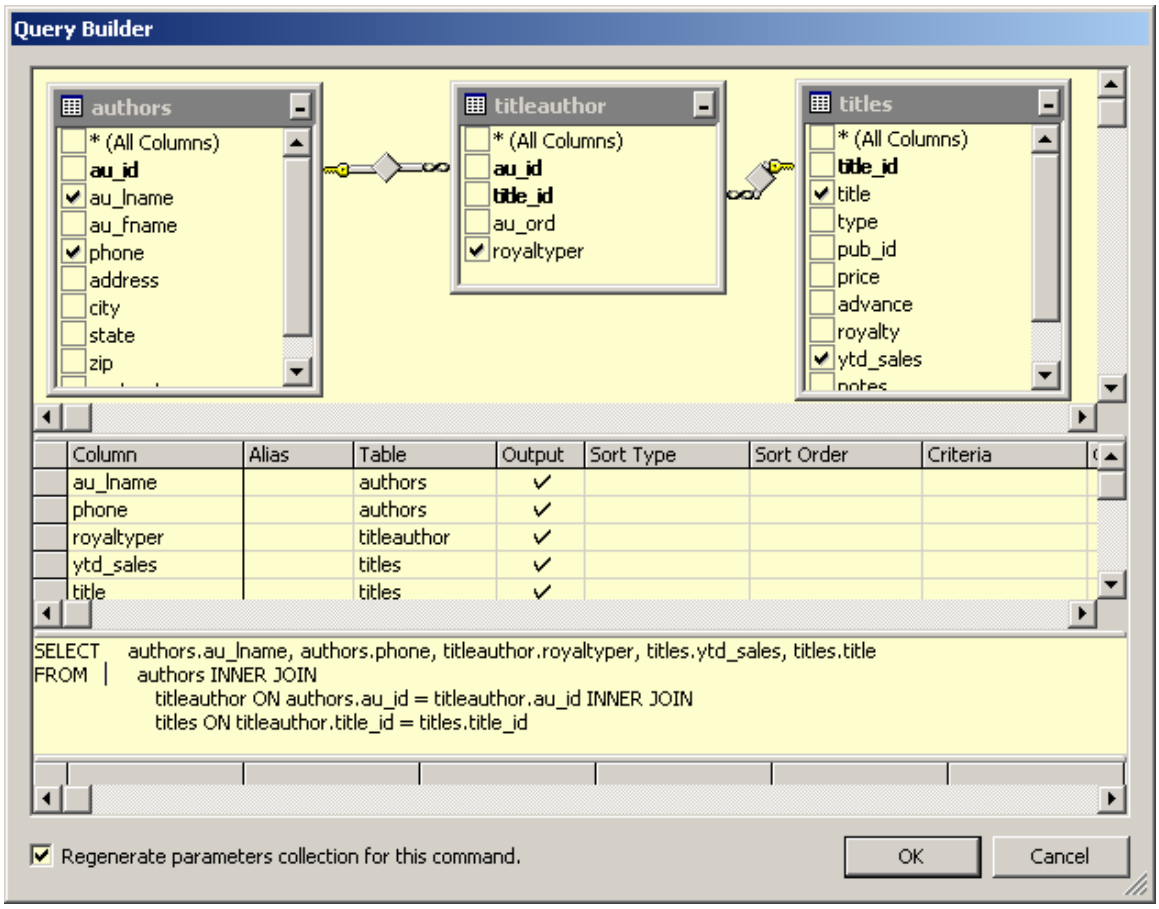


Figure 8.37 Building the SQL using the Query Builder tool

From this point on, the process is the same as discussed on Example 8.1. You need to create a DataSet and add a DataGrid to the form and set its DataSource property equal to DsAuthorTitle1._Table. You then add a Form_Load event with code like that shown in Figure 8.30. The running application is shown in Figure 8.38.

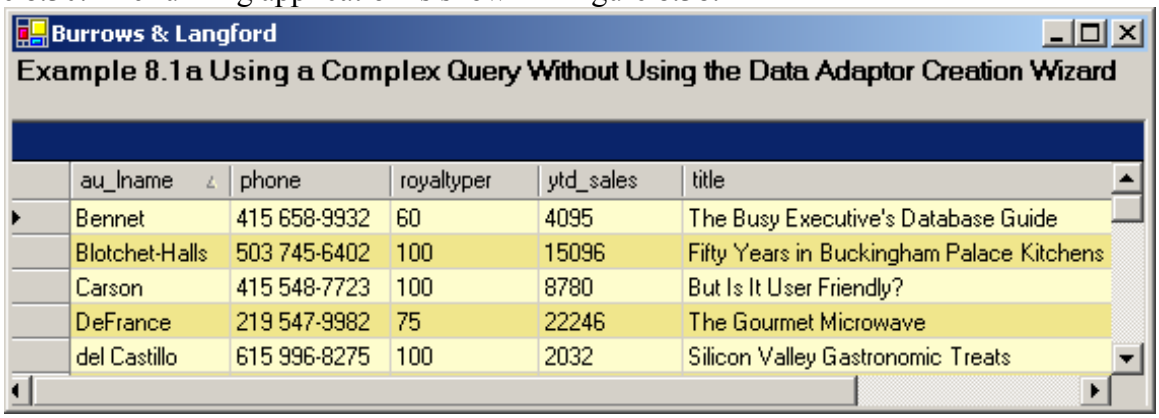


Figure 8.38 Example 8.1a at run time

Note that using this approach generated SQL that did not need to be modified as was the case with the SQL created within the Data Adapter Creation Wizard. In addition, only an SQL Select statement was defined. If you want to define an SQL Insert, Delete,

or Update command, you would need to modify the CommandText properties for these three properties of the data adapter.

In the examples that follow we will use the Data Adapter Creation Wizard but feel free to experiment with setting the values manually like was done in Example 8.1a.

Updating a table

ADO.NET supports the updating the original data source if you want to include that capability in your application. As the user changes data in a DataGrid for example, the in-memory copy of the data stored in the DataSet object is also changed. The data adapter can then cause the original data source to be changed with using its Update method. This method examines every record in the specified data table in the DataSet and, if a record has changed, sends the appropriate Update, Insert, or Delete command to the database.

Example 8.2 Updating a Database

This example allows the user to make changes to records in the authors table in the "pubs" database. The data are automatically loaded into a DataGrid when the form is loaded. When the user wants to save changes, she clicks on the "Save Changes" button. Figure 8.39 show Example 8.2 at runtime. In this example, a new record has been added (au_id = 111-11-1111) and the phone number for Heather McBadden has been changed.

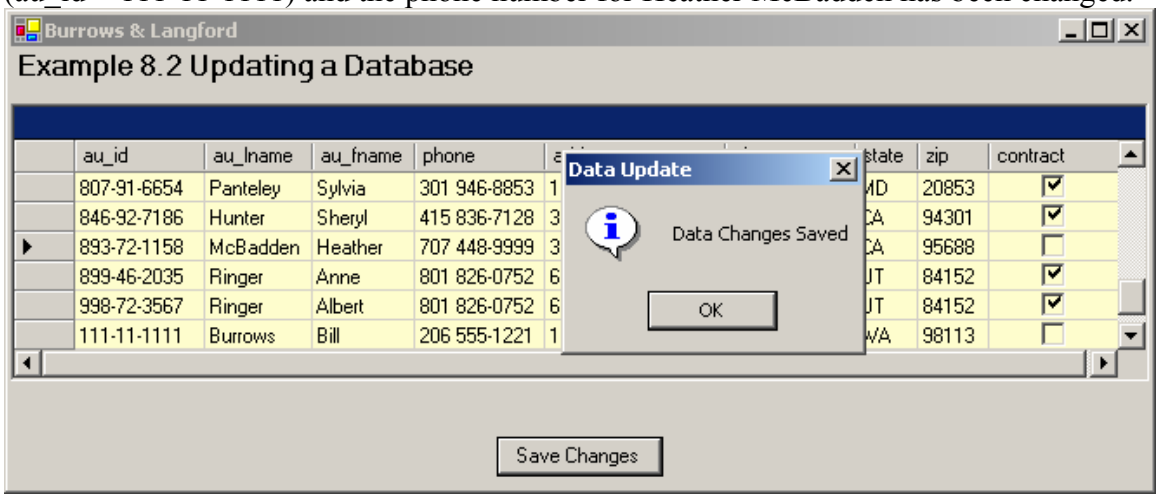


Figure 8.39 Example 8.2 at runtime

For this example we have created an OleDbDataAdapter with an SQL statement that includes all the fields from the authors table. A DataSet was then created using the DataAdapter and the authors table. Finally, a DataGrid was added with its DataSource property set to DsAuthors1 and the DataMember property set to authors.

The code for the form's load event and the click event for the "Save Changes" button is shown in Figure 8.40. In this code you can see the use of the Update method that causes contents of the DataSet DsAuthors1 being used to update the actual database data associated with the data adapter (OleDbDataAdapter1).

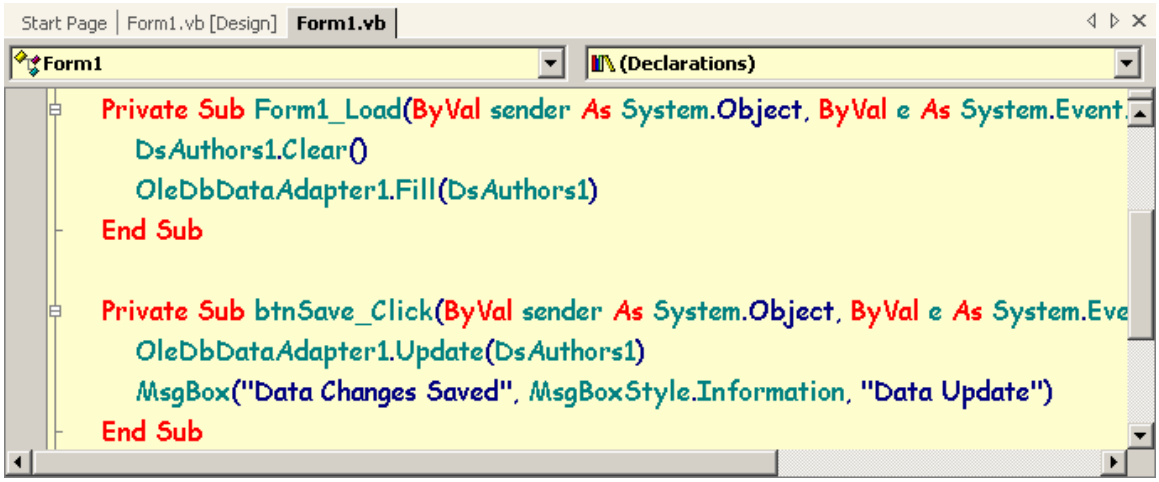


Figure 8.40 Code for Example 8.2

Exercise 8.6. Modify Exercise 8.3 so that it has the ability to save changes.

Exercise 8.7. The DataSet component has a property named HasChanges that is a Boolean property set to true if the DataSet includes values that have been changed and differ from the data in the underlying database. Modify Example 8.2 by removing the "Save Changes" button and replacing it with an "Exit" button. The click event for the "Exit" button should use the HasChanges property to determine if any changes have been made in the DataSet. If they have, the message "Changes have been made. Do you want to save them?" should be displayed in a message box. If the user clicks on Yes, then save the changes and then exit the application. If the user clicks on No, exit the application without saving the changes.

Parameterized Queries

Often users want to specify information that will be used as the basis for a query. For example, the user might want the data displayed in the DataGrid in Example 8.2 to be limited to authors in a particular state. The user would specify the appropriate state code and this would then be used to select the matching records from the database. In this case, the **state code the user enters is referred to as a *parameter* of the query**.

We will look at two examples that use parameters in their queries. These examples differ in the way they display the results to the user. In our first example (Example 8.3) we present the results of the query in TextBox controls one record at a time. In the second example, the results of the query are presented in a DataGrid control.

Example 8.3 Parameterized Query Using TextBox Controls

In this example we "bind" TextBox controls to a DataSet. ***Data binding* refers to the process of associating one or more properties of a Visual Basic .NET control to a specific data source** such as a DataSet. For example, it is very common to bind the Text property of a TextBox control to a specific column (field) in a DataSet. As a result, the value of the field for the currently active row in the DataSet will be displayed in the Text property of a bound TextBox.

There are two types of data binding available – simple data binding and complex data binding. ***Simple data binding* allows the property of a control to be bound to a single data element such as a single field within a DataSet.** ***Complex data binding***

allows the property of a control to be bound to more than one data element in a DataSet. For example, we have already seen how the DataGrid control can display many columns and many rows from a DataSet. Other controls, including ComboBox and ListBox controls, also support complex data binding.

In this example, we ask the user to enter a state abbreviation into a TextBox and then retrieve records from the authors table that match the state abbreviation. The application at execution time is shown in Figure 8.41.

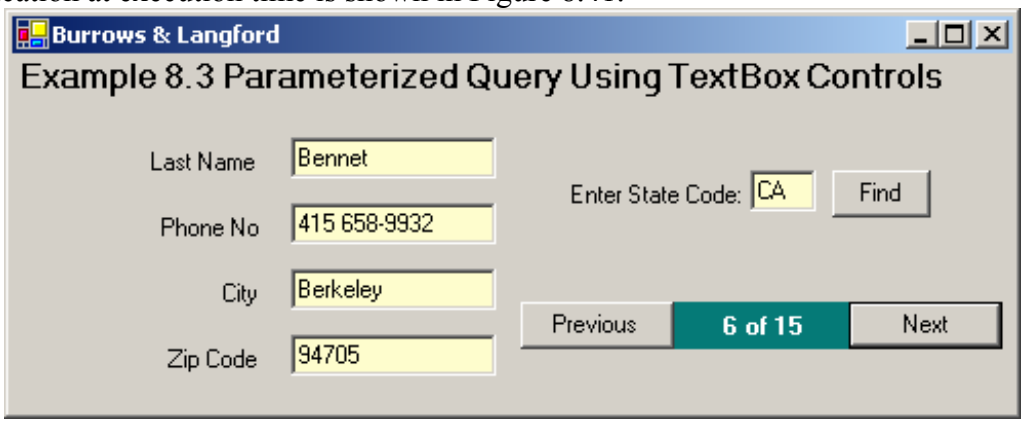


Figure 8.41 Example 8.3 at runtime

In creating this example, we need to add a OleDbDataAdapter and DataSet as in the previous examples. However, there is a difference in the SQL generated within the OleDbDataAdapter wizard from what we have already seen. The SQL select statement that we need would be similar to the following.

SELECT au_lname, phone, city, zip FROM authors WHERE (state = 'CA')

However, there is a problem with the select statement above; we do not know at design time that the user will want to see authors in California ('CA'). SQL has a solution for our problem called a parameterized query. The SQL statement we want to use is:

SELECT au_lname, phone, city, zip FROM authors WHERE (state = ?)

At runtime, the "?" is replaced with a value provided by the user by calling the appropriate methods of the data adaptor.

To create the SQL statement using the OleDbDataAdapter wizard, we bring up the Query Builder, select the au_name, phone, city, and zip code fields and enter "= ?" in the Criteria field for the state field. Figure 8.42 shows the Query Builder panel including this entry. If you look at the select statement in the lower part of the panel, you will see how the "WHERE" clause has been created with the criterion (state = ?). Also note two other things. First, observe that the state field does not have a checkmark next to it in the Output column. We need this field for selecting records from the DataSet but we do not need it for display purposes. Second, notice the symbol to the right of the field "state" in the authors table at the top of the panel. This icon indicates that records are "filtered" based on this field. **The term "*filter*" is used to describe the process of selecting a subset of records from the original set**.

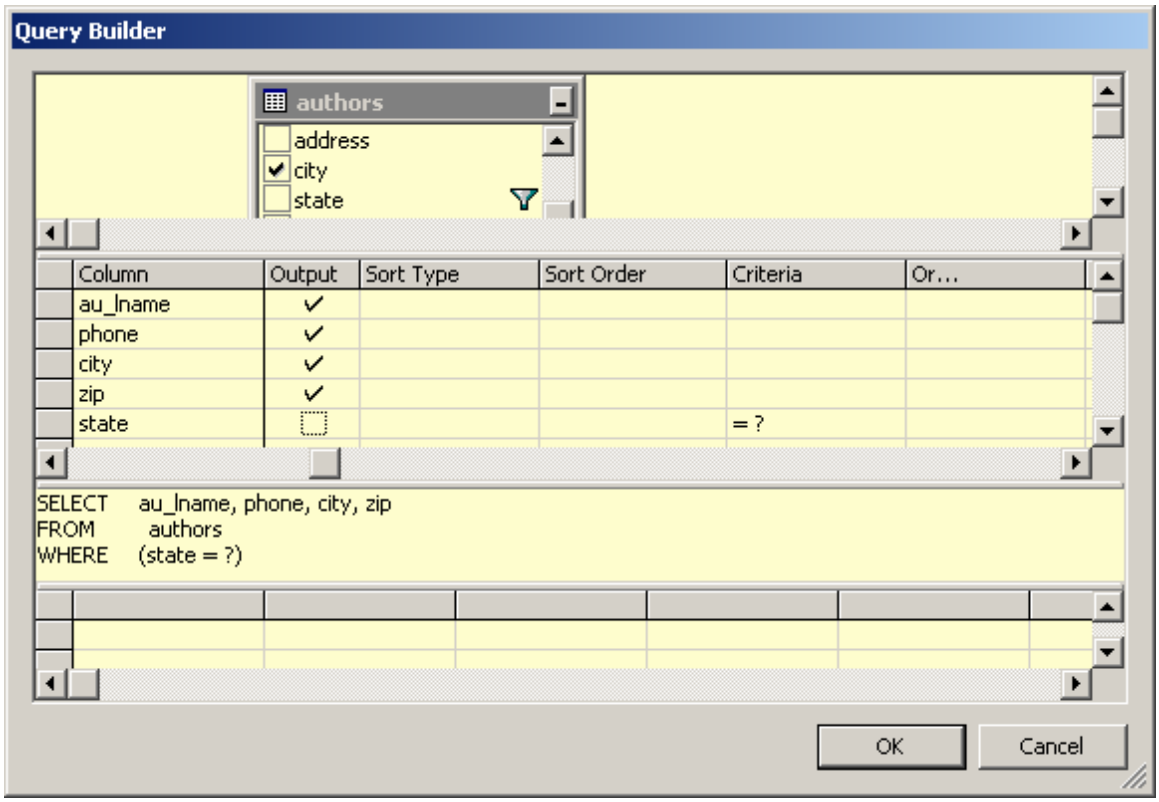


Figure 8.42 Using the Query Builder to build a parameterized query

After creating the data adapter, a DataSet and other components need to be added to the project. Note that there is a label between the Previous and Next buttons. This label has a blue-green background with white fore color making the font white.

To bind the four TextBox components (Last Name through Zip Code), you need to go to the Properties window and open the (DataBindings) property by clicking on the "+" sign and then binding the Text property to the appropriate field in the DataSet. Figure 8.43 shows the process of binding the "au-lname" field to the TextBox named txtName.

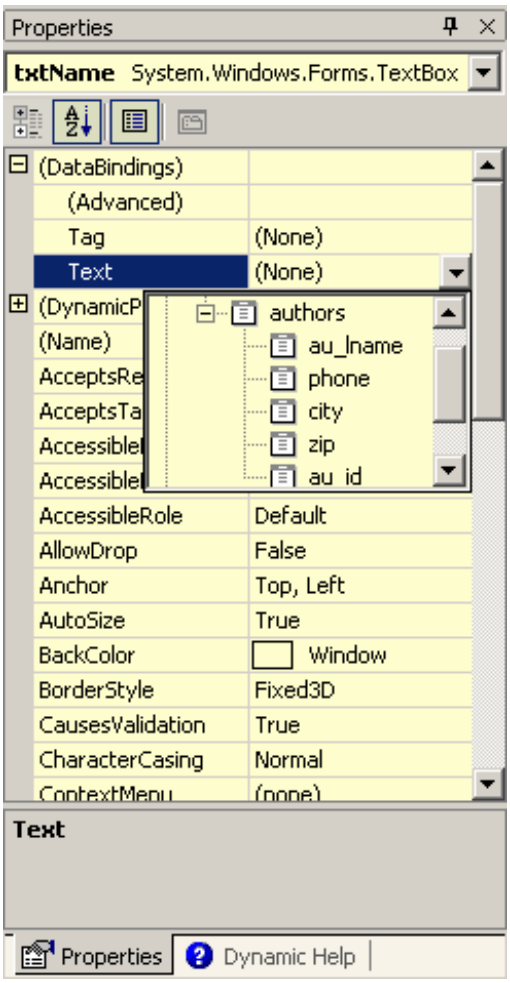


Figure 8.43 Binding a TextBox to a field in the DataSet

After binding the four TextBox controls to their associated fields, we need to write the code to show the selected records as well as navigate from record to record. To begin, we write the code for the "Find" button. Figure 8.44 shows the click event for this button.

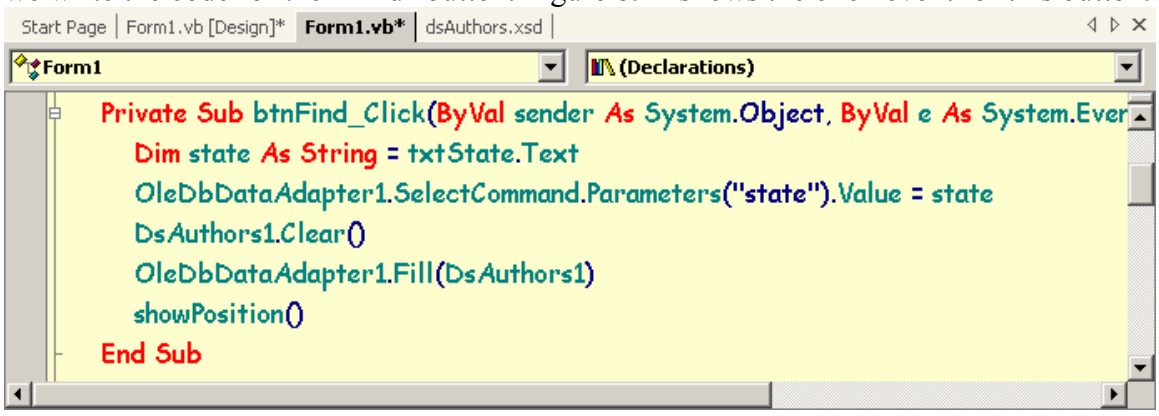


Figure 8.44 Code to find records for a specific state

There are two statements that need explaining. The first is

```
OleDbDataAdapter1.SelectCommand.Parameters("state").Value = state
```

This statement searches the parameter collection for the Select command associated with the data adapter to see if the parameter, "state" in this case, exists. If it does, a value (on the right of the equal sign) is assigned to the parameter. That is, it replaces the question mark in the original Select statement,

SELECT au_lname, phone, city, zip FROM authors WHERE state = ?

with the value of the expression on the right-hand side of the assignment statement. There may be more than one parameter in the Select statement and if there were, you would need to assign a value to each parameter.

The second statement that needs explaining is the procedure showPosition(). This procedure, whose code is shown in Figure 8.45, updates the record status in the label between the Previous and Next buttons.

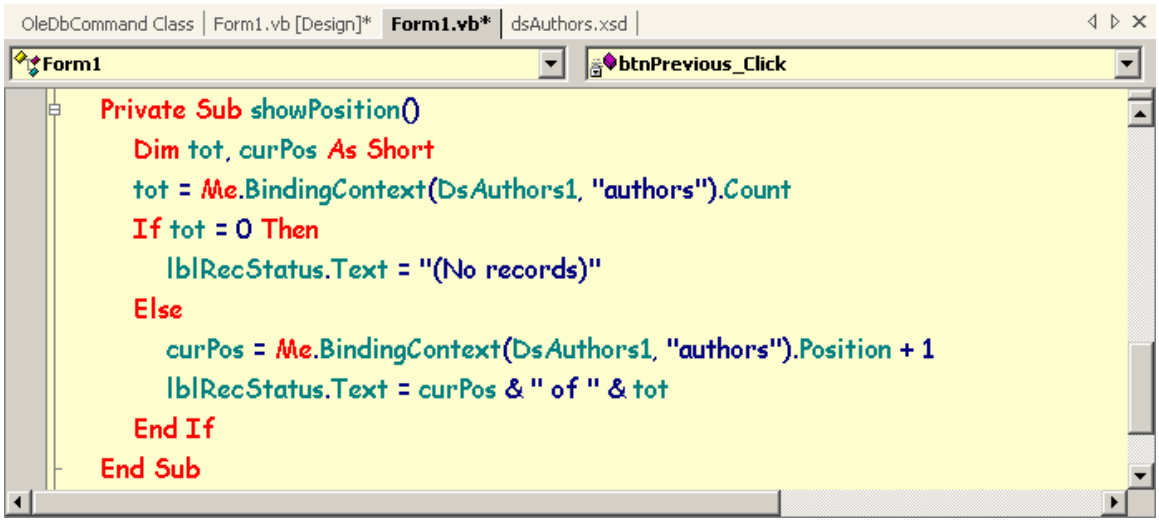


Figure 8.45 Code for the showPosition() procedure

This procedure first determines the total number of records in the DataSet. If the total is zero, it places the text "(No records)" into the label. Otherwise it determines what the current record index is and creates the text similar to "3 of 15" where the "3" is the current position and the 15 is the total.

The two references

```
Me.BindingContext(DsAuthors1, "authors").Count
Me.BindingContext(DsAuthors1, "authors").Position
```

need explanation. The reference to "Me" is a reference to the form object where the code exists. Each form has a ***BindingContext*** object associated with it. This object **keeps track of all the data sources associated with the form**. The parameters (DsAuthors1, "authors") are necessary because it is possible for a form to have more than one data source associated with it. The parameters are used to find the correct data source.

One final explanation relating to the Position property is necessary. The records in the DataSet are numbered beginning with zero. So if there are 5 records (the Count property would be equal to 5), the records are indexed with the numbers 0, 1, 2, 3, and 4. If the current record index were 2 for example, it would be the 3rd record. If we didn't add 1 to the Position property when we displayed the status information, we would end up saying "2 of 5" when we are referring to the 3rd record.

Finally we need to write the code for the Previous and Next buttons. This code is shown in Figure 8.46. In this code we are again referencing the Position property of the data source like we did in the showPosition() procedure. Here we either increment or decrement the value of this property moving forward or backward one record. The system also controls the value of the Position property to be sure it stays legal (0 ≤ Position < Count). That is, if Position equals 0 (the first record) and the user clicks the Previous button, the value of Position will not go negative.

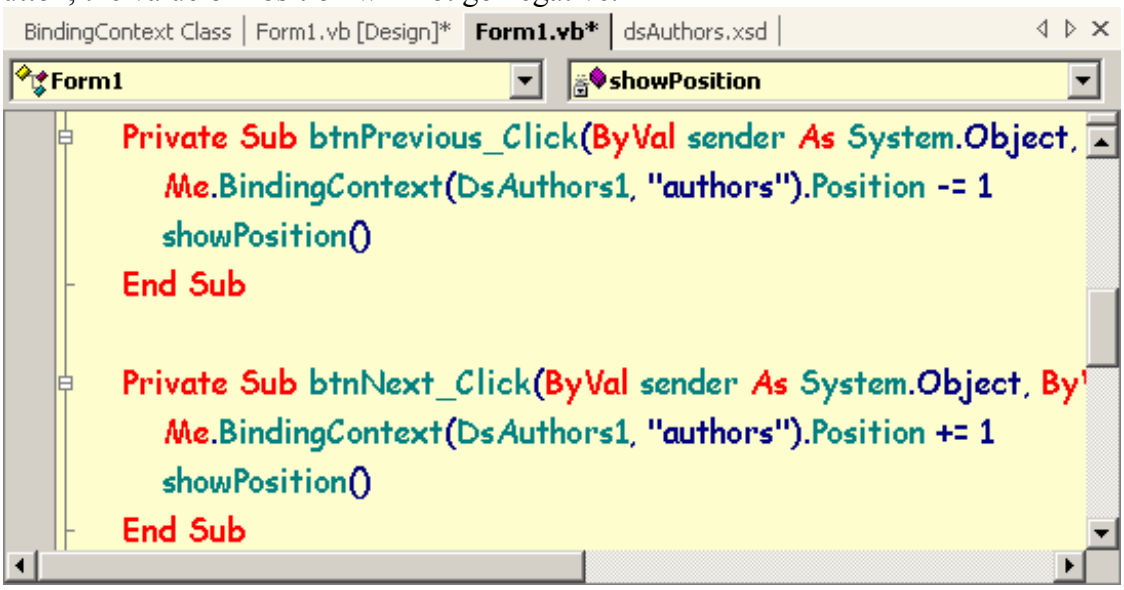


Figure 8.46 Code for the Previous and Next buttons

Exercise 8.8. Modify Example 8.3 so that it finds records that match a specific zip code entered by the user instead of a state code.

Exercise 8.9. Modify Exercise 8.5 so that it displays records based on a state code entered by the user at runtime.

Our second example dealing with parameterized queries is very similar to Example 8.3. It differs in that it provides the user with a ComboBox component filled with all the state codes found in the database. This makes the application user-friendlier in that the user simply selects the state he is interested in and does not have to remember what the valid codes are.

Example 8.4 Parameterized Query with ComboBox and DataGrid Controls

Figure 8.47 shows this example at runtime. The ComboBox on the left is populated with state codes from the database. When the user selects a code, the DataGrid on the right is filled with records from the database that match the user-selected state code.

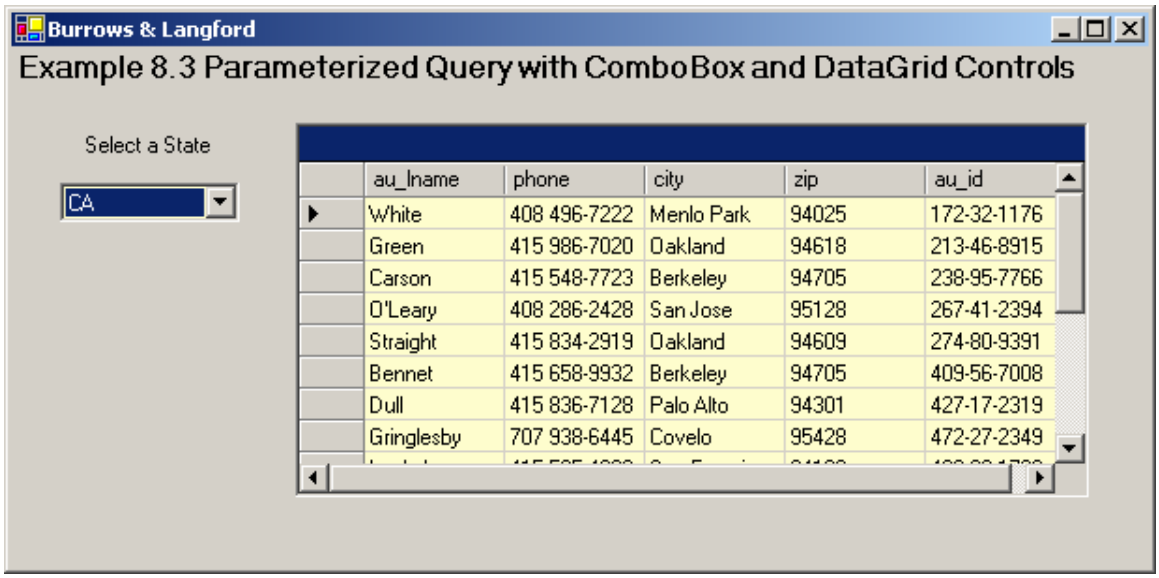


Figure 8.47 Example 8.4 at runtime

We must create two DataSets using two different data adapters. We begin by creating an OleDbDataAdapter just like the one we created for Example 8.3. This data adapter will be used to retrieve the records from the database to fill the DataGrid. The SQL for this provides for a parameterized query with the expectation that the user will supply the code used to select the set of state-specific records. Figure 8.48 shows the Query Builder window that is used to create the SQL.

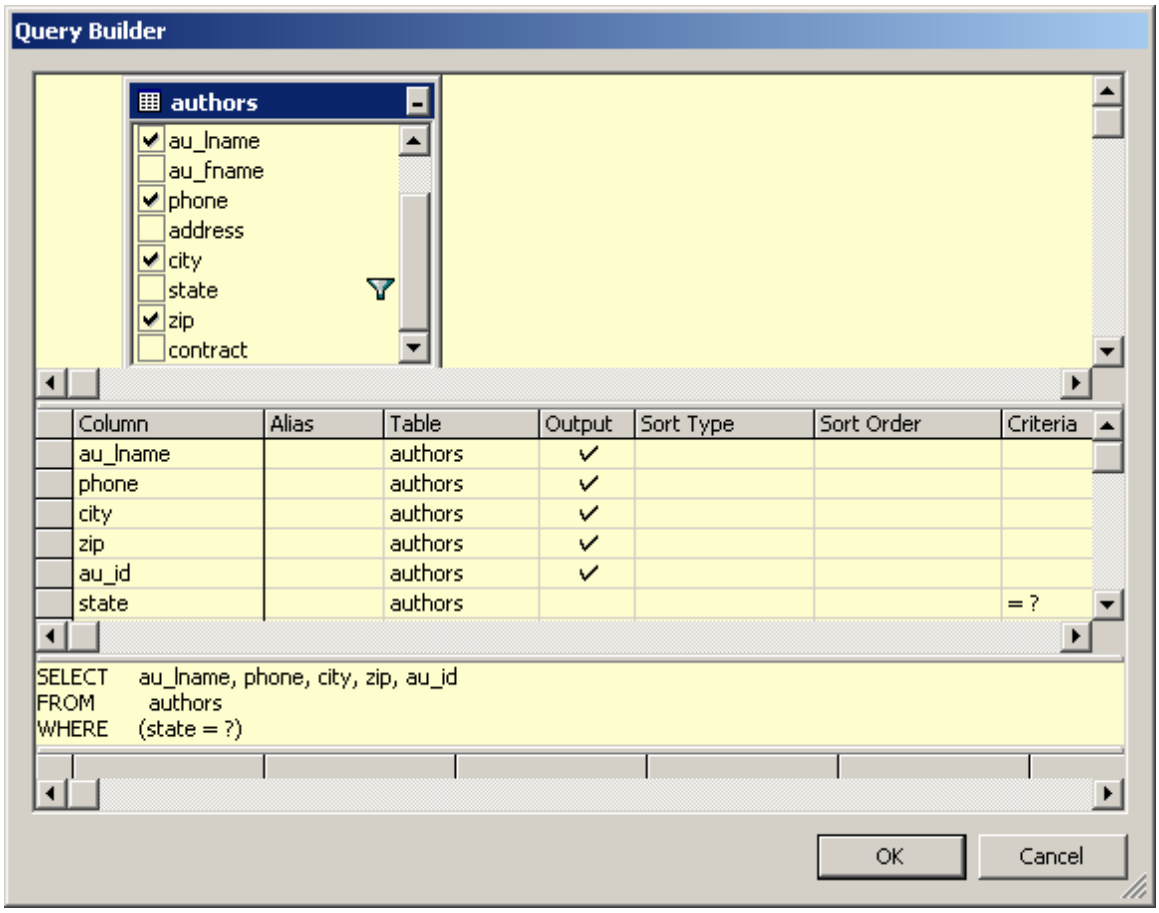


Figure 8.48 Query Builder window used to create the SQL statement in Example 8.4

After creating the OleDbDataAdapter (named OleDbDataAdapter1), a new DataSet object is created (named DsAuthors1).

The second OleDbDataAdapter is used to extract a unique list of state codes from the authors table to fill the combo box. However, there is a little problem that needs to be overcome. If we just select the state codes from the authors table, we end up with potential duplicates. For example, there are a total of 15 authors from California who all have "CA" as their state code. SQL has a way to solve this problem by adding the keyword **"*Distinct*" to the query. This keyword results in a set of records that are unique**, that is, if there are any duplicate rows, then they are reduced to a single row. Figure 8.49 shows the Data Adaptor Wizard for our second OleDbDataAdapter object. The Query Builder does not have a way to designate a Distinct so we need to enter the SQL statement directly as you can see in the figure.

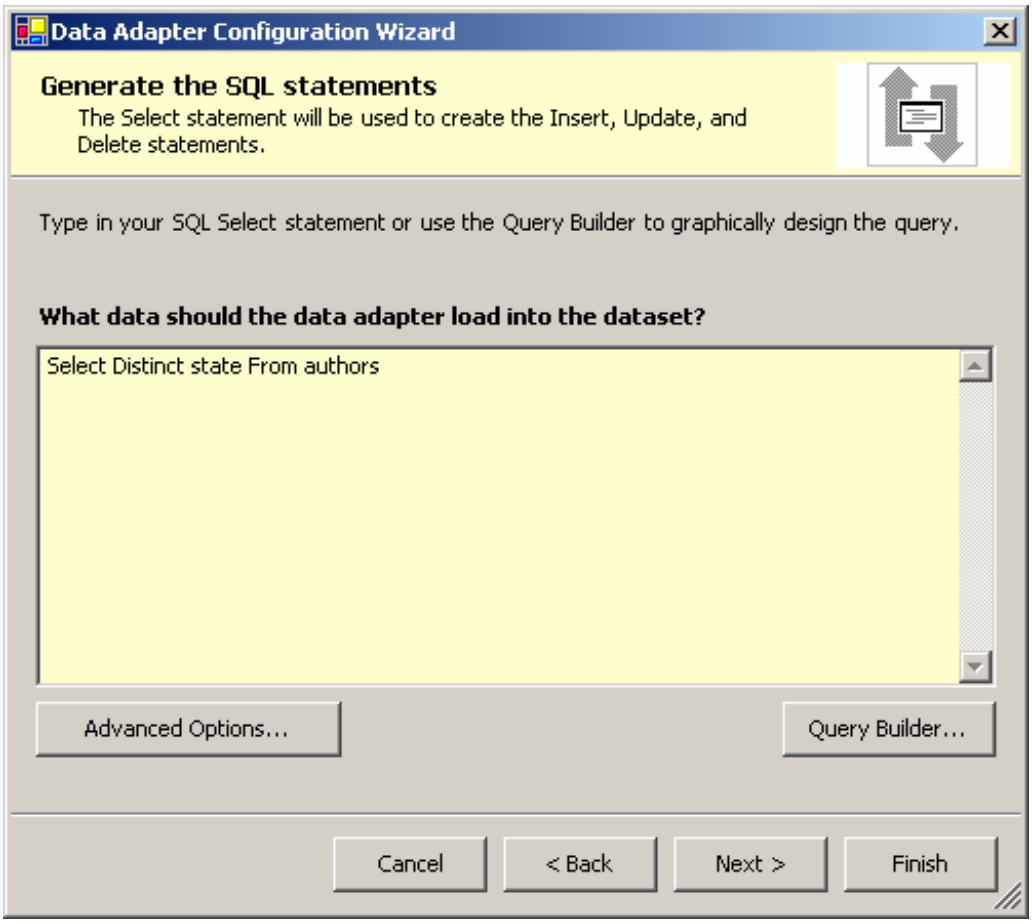


Figure 8.49 Specifying a Distinct Select query

After creating the second OleDbDataAdapter, named OleDbDataAdapter2, we create a new DataSet named DsStates1. You need to be careful to associate this new Dataset with the second OleDbDataAdapter. Figure 8.50 shows this action.

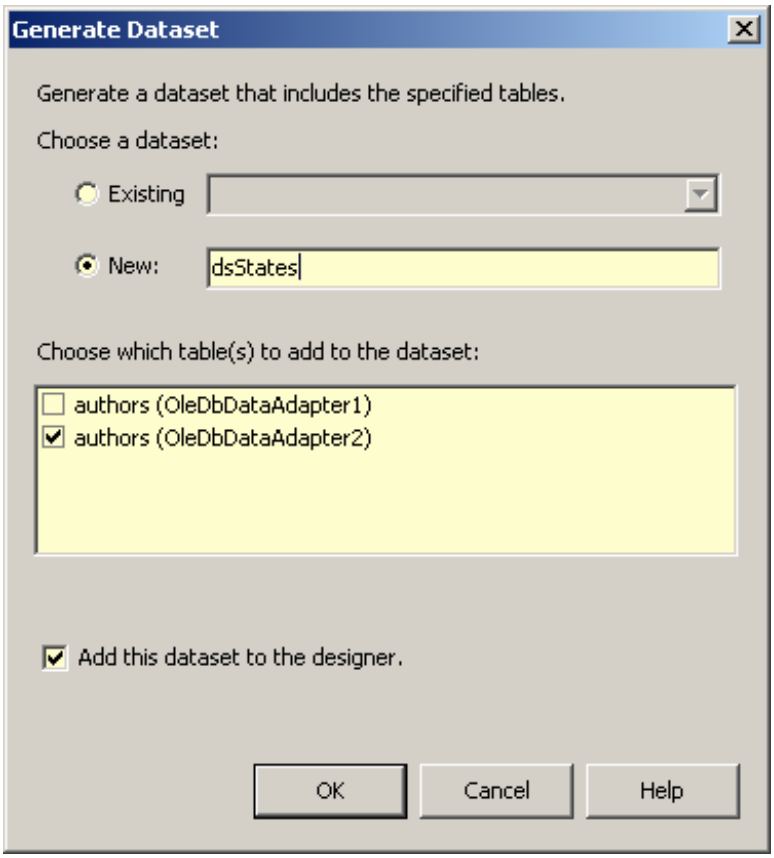


Figure 8.50 Creating the second DataSet associated with a distinct list of state codes

After adding the ComboBox and DataGrid controls to the form, we need to bind them to the appropriate data sources. For the ComboBox, we need to set its DataSource property equal to the data set DsStates1 and its DisplayMembers to authors.state. We also set the ComboBox's DropDownStyle property to DropDownList. Figure 8.51 shows these settings.

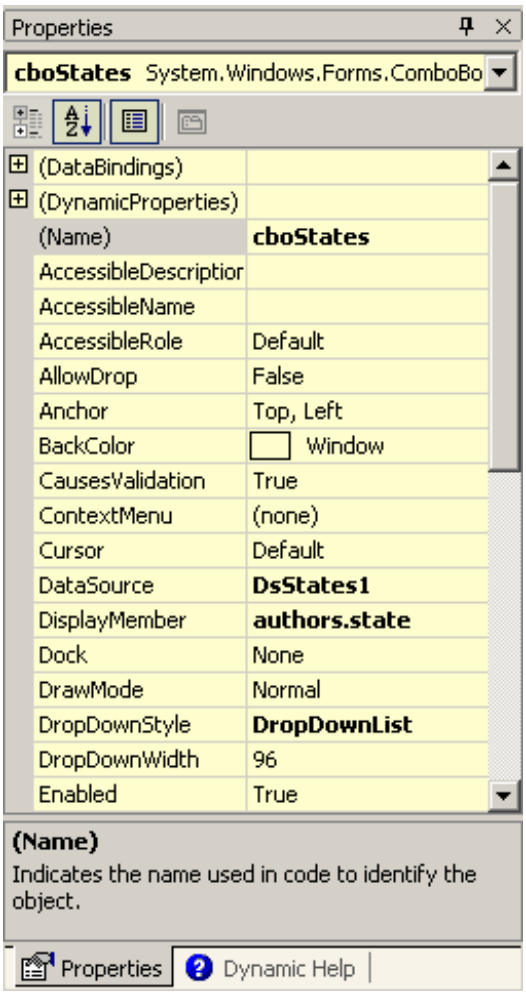


Figure 8.51 Setting property values for the DataSource, DisplayMember, and DropDownStyle properties

For the DataGrid control, we set its DataSource property to DsAuthors1.authors. Figure 8.52 shows the choices Visual Basic .NET provides for this control's DataSource property. You may find this list a bit confusing and are wondering how to decide which of the choices is appropriate, so let's go over each option. The choice DsStates1.authors refers to a DataSet that contains just a list of state codes extracted from the authors table. We know this because the DsStates1 data set is associated with the OleDbDataAdapter2 data adapter (that includes the SQL Select Distinct query for state codes). We know that this is not correct for the DataGrid.

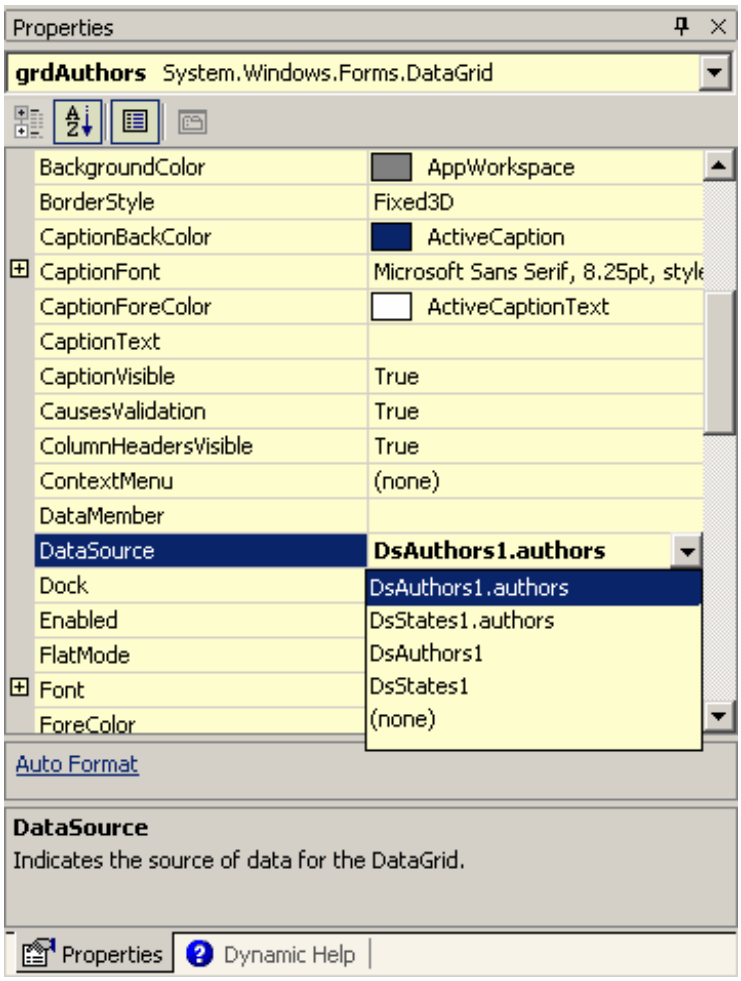


Figure 8.52 Choices of values for the DataGrid's DataSource property

The choice DsStates1 is probably not tempting (it only relates to state codes, not the fields shown in the DataGrid.) This leaves the two choices DsAuthors1.authors and DsAuthors1 (without a table qualification). A data set can include more than one table. However, our DataGrid can only display information from one table. If we chose to associate the DataSource property to DsAuthors1 (without a table qualification), we would not be providing enough information because the DataGrid wants to know which table in the DataSet it should be bound to. This is true even in our case where our data set only contains one table. So the correct choice for our purpose is DsAuthors1.authors.

After binding the two controls to the correct DataSets, we need to write the code to fill the ComboBox and DataGrid. The ComboBox is populated when the form first loads so we place the code in the form's load event. Figure 8.53 shows this code.

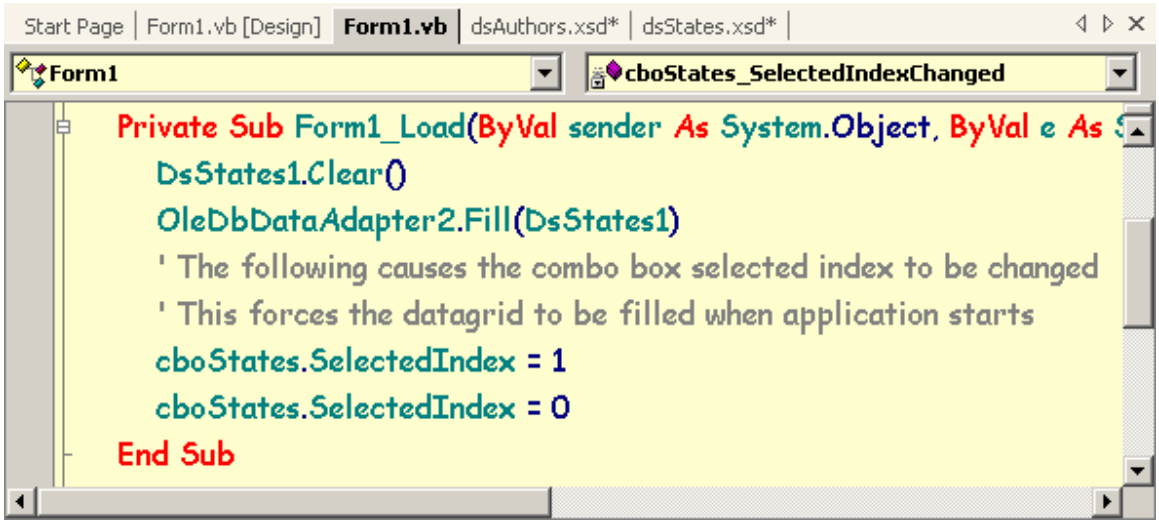


Figure 8.53 Code to populate the ComboBox with state abbreviations

This code first clears the DataSet object and it then fills it from data as defined in the SQL statement in the data adapter. The last two statements are included to guarantee that the DataGrid is populated when the application first starts. As you will see next, we use the ComboBox's SelectedIndexChanged event to populate the DataGrid. By changing the selected index in the Form1_Load event, the forces the ComboBox's SelectedIndexChanged event to be called.

The code for the ComboBox's SelectedIndexChanged event is shown in Figure 8.54. The code for the selected state is taken from the ComboBox and then used to set the parameter value in the parameterized query associated with OleDbDataAdapter1. Finally, the DataSet is cleared and then filled from the SQL statement associated with OleDbDataAdapter1.

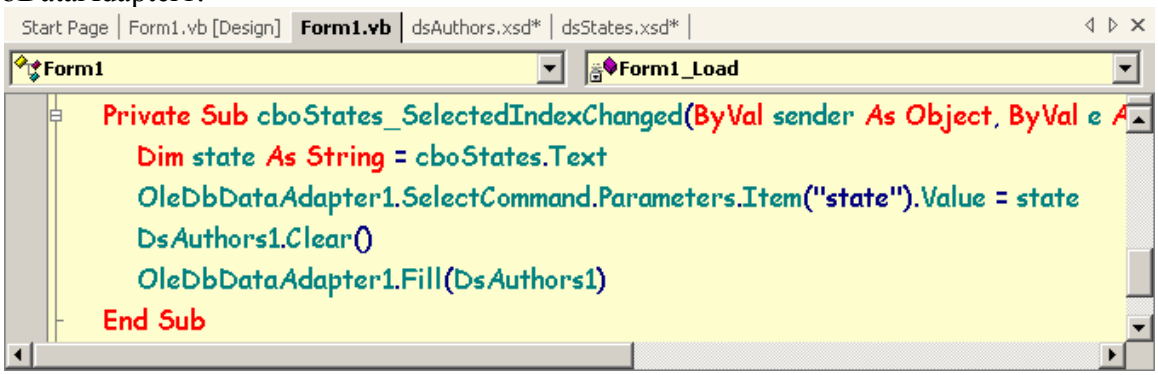


Figure 8.54 Code to populate the DataGrid when the user selects a new state

---

Exercise 8.10. Modify Example 8.4 so that the ComboBox displays (distinct) the zip codes instead of state codes. The user then selects records based on zip code values.

Exercise 8.11. Modify Exercise 8.5 so that it displays records based on a state code chosen by the user from a ComboBox at runtime. Be sure that the state codes are unique within the ComboBox.

Master/Detail record display

In all of our previous examples we have worked with a DataSet that had a single table. Example 8.4 had two DataSets but each had only a single table associated with it. Example 8.2 used a complex SQL query that involved three tables, but the result of the SQL query was a single table. Here we look at using a DataSet that includes two tables that are related with a common field.

When you have more than one table in a DataSet, you have what is called a ***Master/Detail DataSet***. This **defines a relationship where each master record (from one of the tables) is related to zero or more records from the second table**. For example, assume that you have one table called Department. Each record in this table has information on a specific department in an organization. You also have a table called Employee that lists all the employees in the organization. Each record in the Employee table includes a field named DeptCode that indicates which department the employee is assigned to. Here the master record would be a record from the Department table and it would be associated with zero or more detail records from the Employee table. As an alternative to the terms master and detail, we sometimes use the terms parent and child. Using this terminology, the departments are the parent records and the employees are the child records.

Example 8.5 Creating a Master/Detail Record Display

In this example, we look at the publishers in the SQL "pubs" database and their employees. The master (parent) records will be the publishers and the detail (child) records will be a list of employees for each publisher.

Figure 8.55 shows the application at runtime. You can see the "+" expander symbol in the left-most column in the table next to each master record. Clicking on the "+" drops down a link to the master's detail records. In Figure 8.55, publisher number 0877 has revealed a link to its employees.

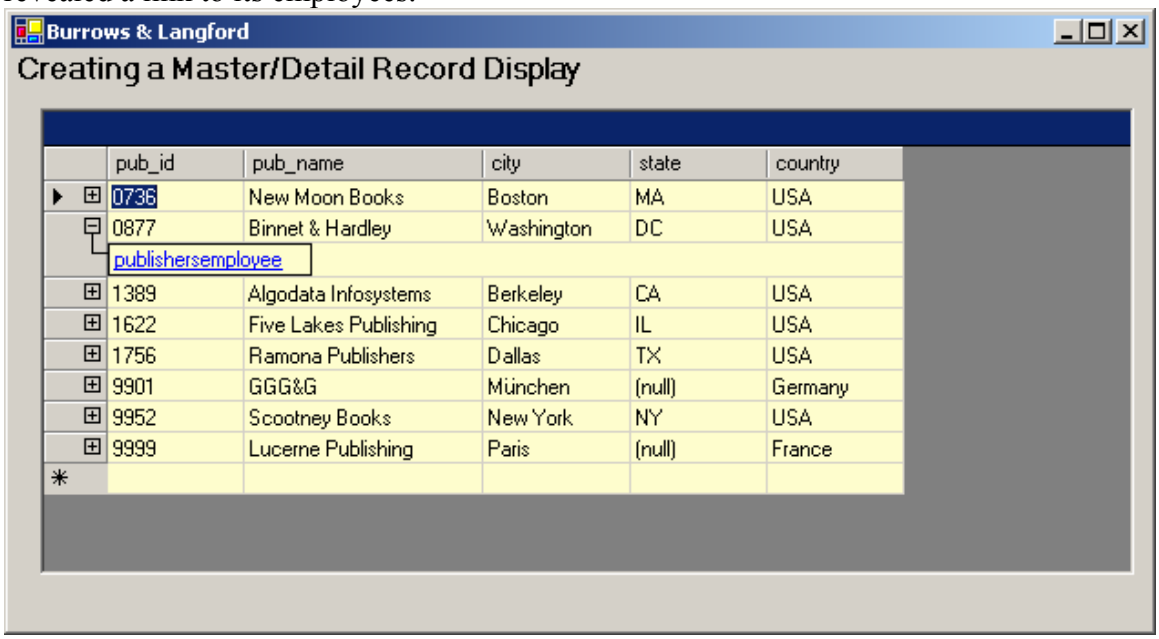


Figure 8.55 Example 8.5 at runtime with one master record's detail record link exposed

If you click on the detail link, you will see the screen shown in Figure 8.56. In this figure you see the 10 employees of publisher 0877. You can also see the publisher details

on the top of the DataGrid control. Note that there are two icons on the upper-right corner of the DataGrid. The left arrow (←) causes the details to be collapsed and the master (publisher) records will be displayed. The other icon, which looks like a rectangle with up and down arrows under it, is used to show/hide the master record detail at the top of the grid.

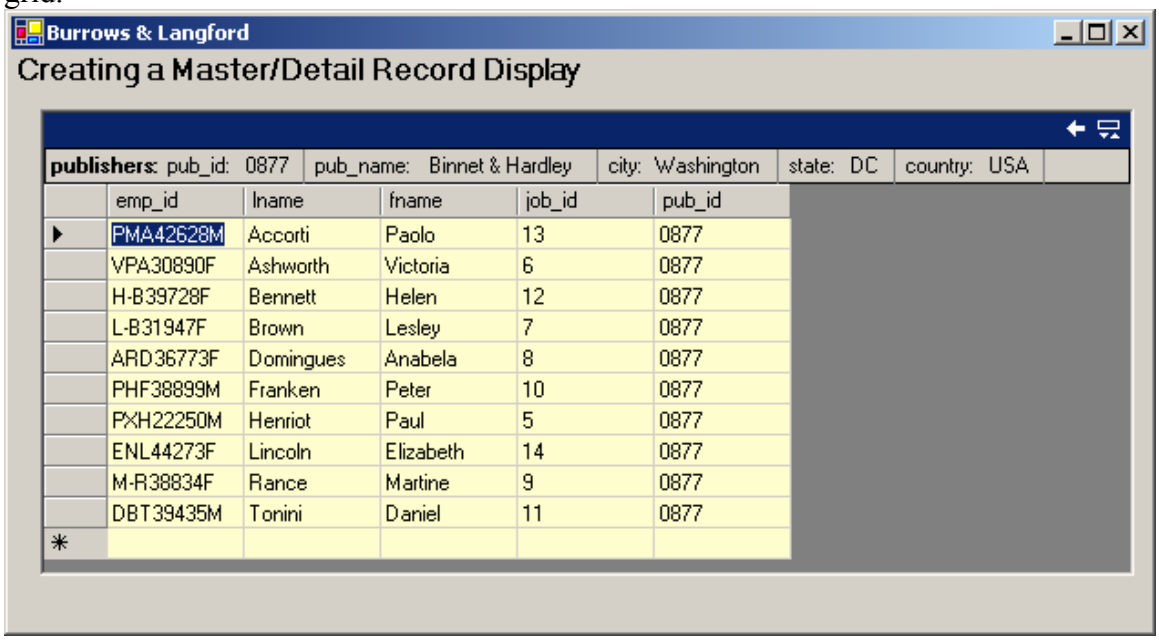
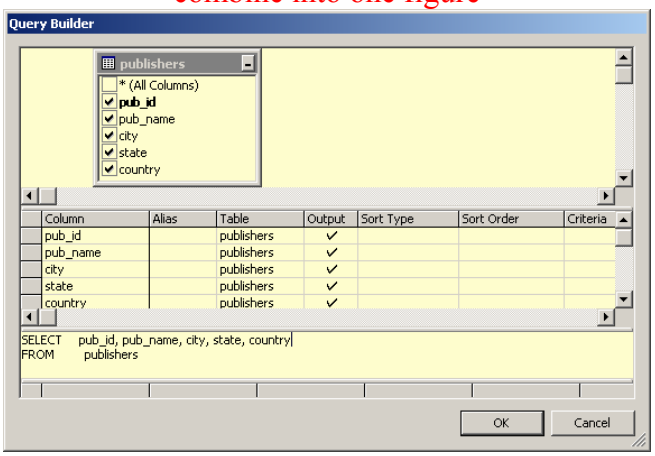


Figure 8.56 Example 8.5 with a publisher's details displayed

As explained earlier, to create a master/detail record display like this, one must create a DataSet that includes two related tables. We need to create two data adapters, one for the publishers table and a second for the employee table. We then need to create a DataSet that includes both of these tables and also has a relationship to link them together.

We start by creating two OleDbDataAdapters. Figure 8.57 shows the Query Builder for constructing the SQL for the publishers table and the employee table. There is nothing new here; just two separate data adapters (OleDbDataAdapter1 and OleDbDataAdapter2) each using a different table from the database.

<combine into one figure>



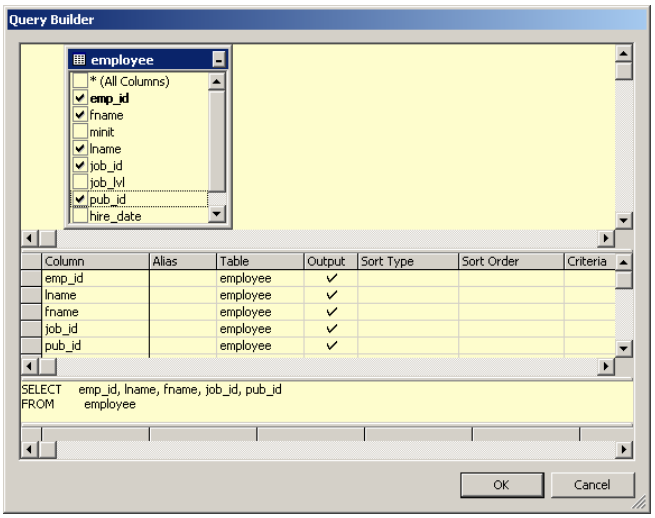


Figure 8.57 Query Builder screens for the publisher and employee tables

We next need to create a DataSet that includes the two tables. We start out as before by selecting Generate Dataset… from the Data menu. Figure 8.58 shows the window used to supply appropriate information to generate the DataSet. Note that we include both the employee and publishers tables in this DataSet.

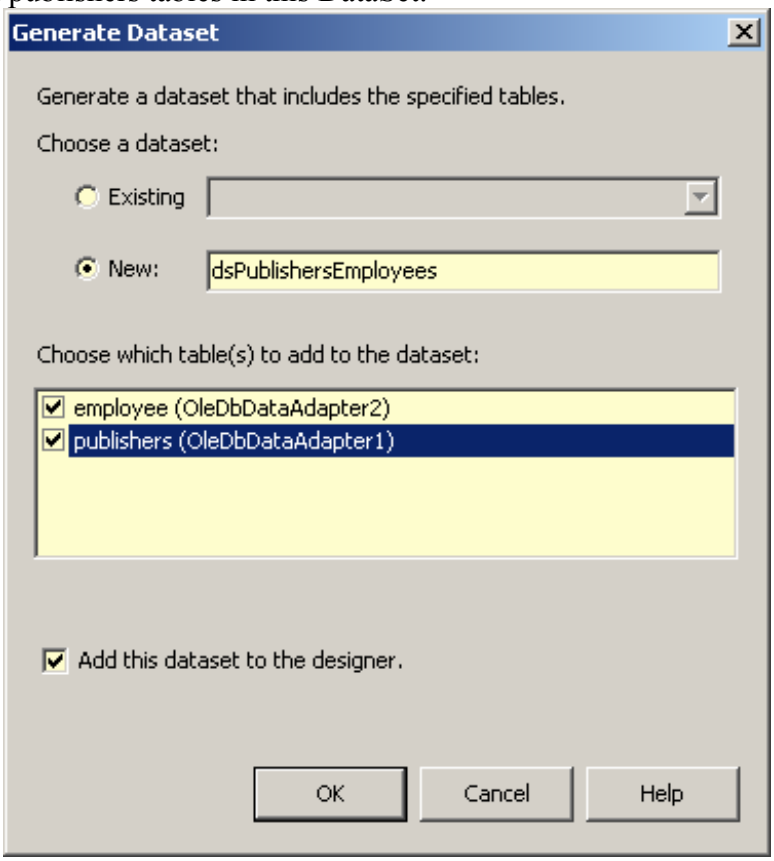


Figure 8.58 Creating a DataSet that includes two tables

After clicking on OK, you should see a new entry in the Solution Explorer. As shown in Figure 8.59, there is now an entry titled dsPublishersEmployees.xsd. This file

contains an **X**ML **S**chema **D**efinition (hence the extension xsd). We talk more about XML and its various supporting technologies in the next chapter.

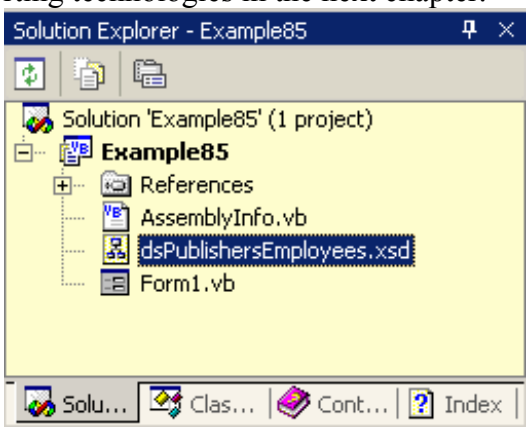


Figure 8.59 The XML Schema Definition for our DataSet

If you double-click on the dsPublishersEmployees.xsd entry in the Solution Explorer you will see something like that shown in Figure 8.60 in the designer window. You can see the two tables that are part of the previously defined DataSet.

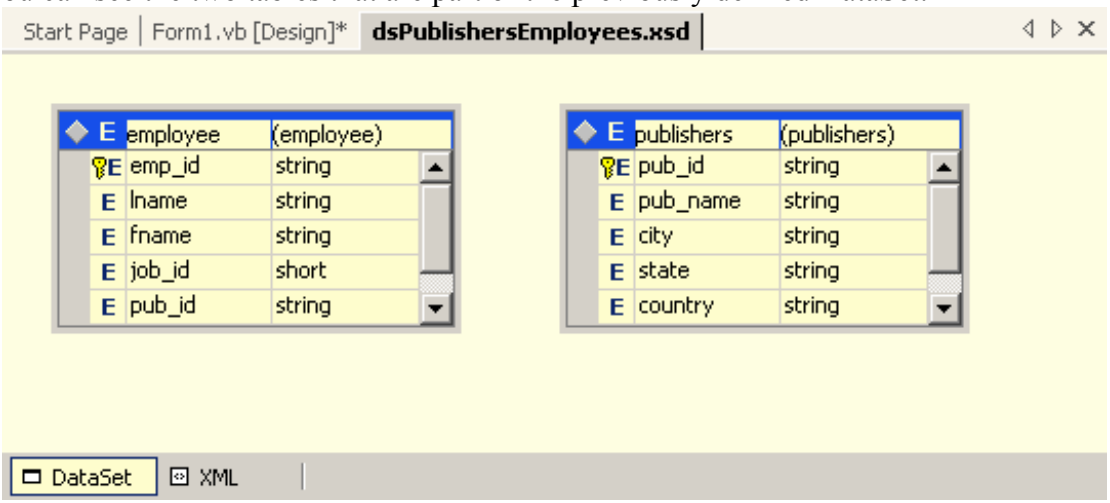


Figure 8.60 A graphical rendering of the dsPublishersEmployees DataSet in the designer window

What we need to do next is establish a relationship between the two tables. Dragging a Relation component from the XML Schema Toolbar to the designer window does this. Figure 8.61 shows the Relation component in the Toolbar.

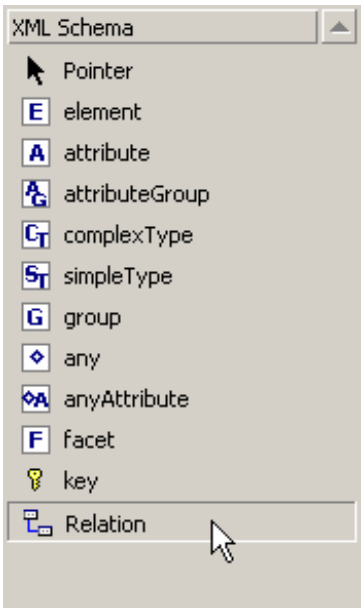


Figure 8.61 The Relation component in the XML Schema Toolbar

We drag the Relation component from the Toolbar to the table that will be the detail (child) table in the relationship. In our case this is the employee table. So we drag the Relation component to the employee table on the left.

After doing this, an Edit Relation dialog box like the one in Figure 8.62 will be displayed. Here we specify the Parent (master) and Child (detail) elements. We also define the field in each table that will be used to link the two tables together. In this case, this is the pub_id field common to each table.

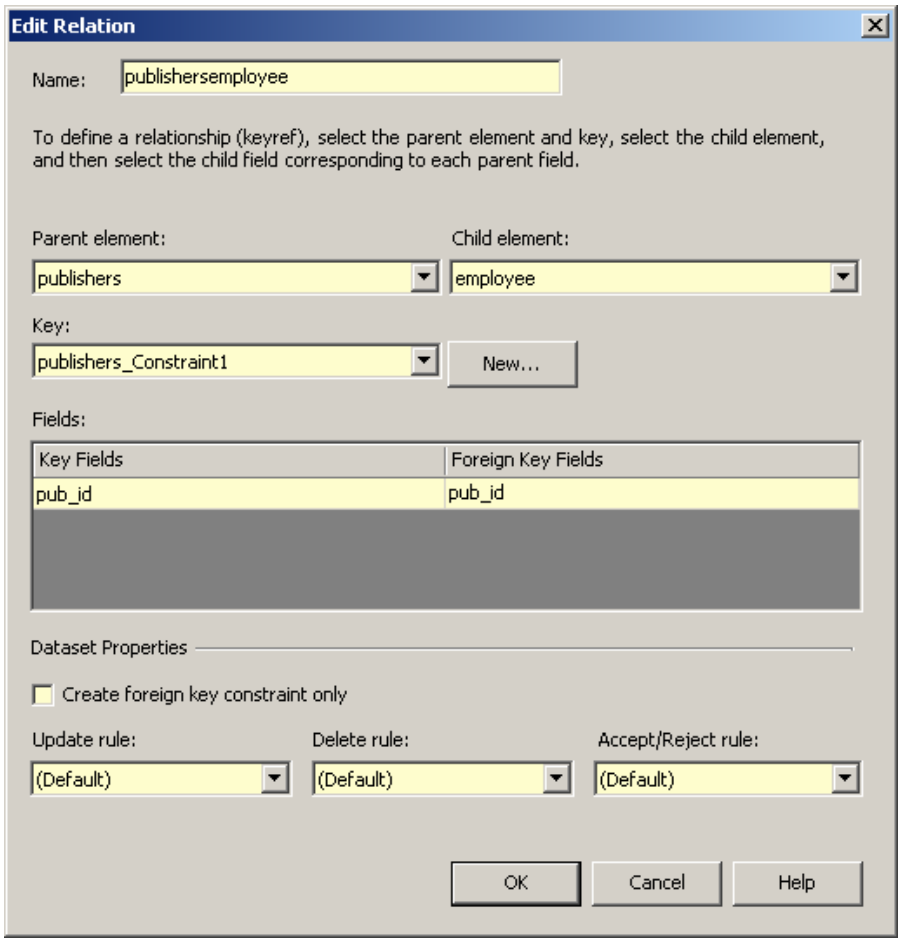


Figure 8.62 Defining the relationship between the publishers and employee tables

After clicking on OK, you should see an updated schema diagram showing the relationship between the two tables. This is shown in Figure 8.63.

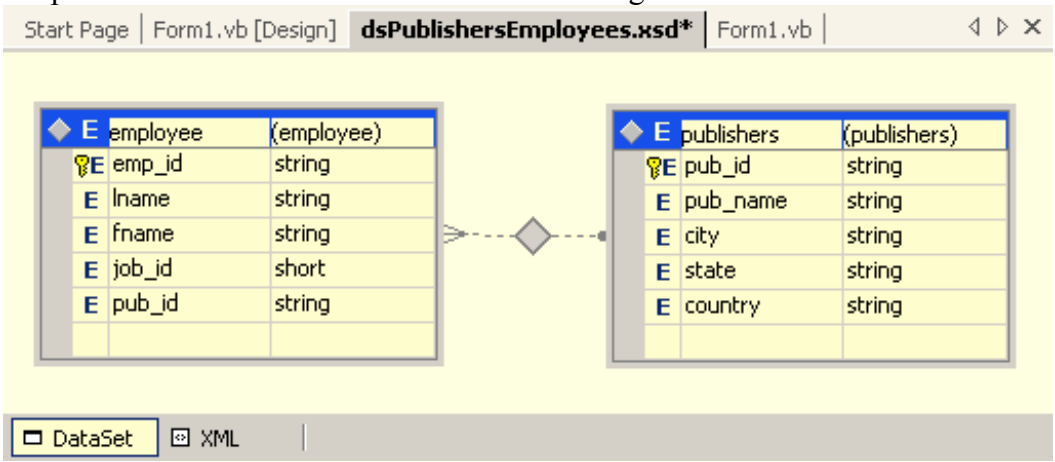


Figure 8.63 The Schema diagram including the new relationship

The next task that we need to perform is to place a DataGrid on the form and associate it to the appropriate data source. As seen in Figure 8.64, we do this by setting the DataSource property to equal DsPublishersEmployees1.publishers. We set the

property to this value so that the publishers records are shown as the master records. If we had chosen instead to set the DataSource property equal to DsPublishersEmployees1.employee, then we would initially see employee records when the application starts, but since these have no detail records associated with them, we could never see the publisher information. If we chose the third option DsPublishersEmployees1, the initial DataGrid would show no records, but instead a "+" expander allowing the user to expand either the publishers table or the employees table. While this is not the behavior we want for this application, it could be useful for other applications.

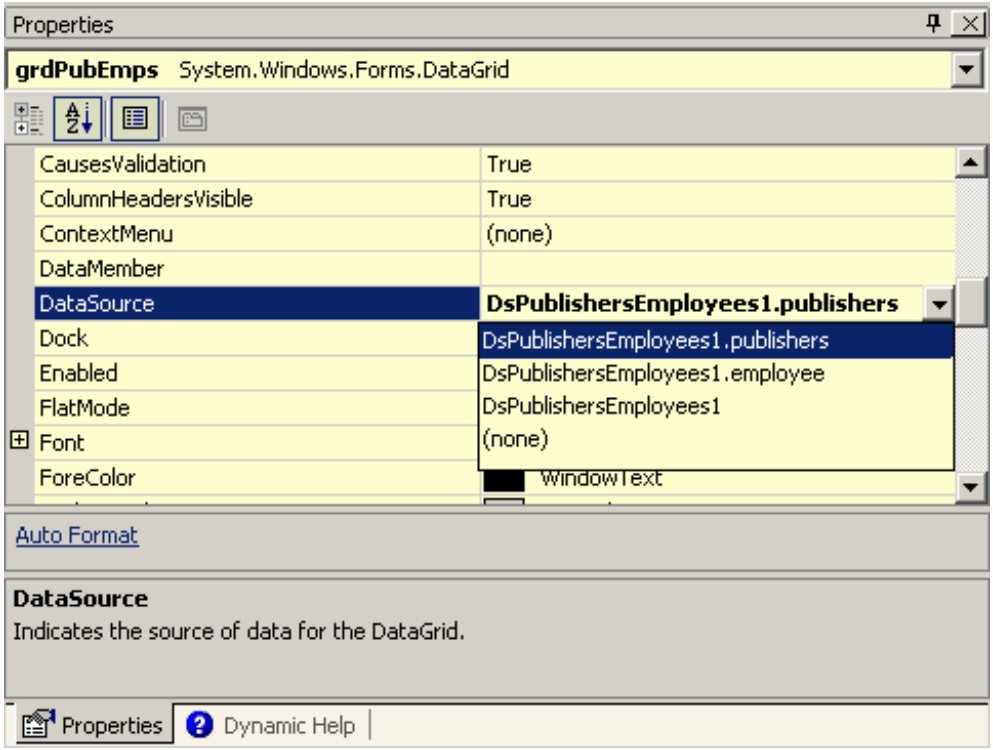


Figure 8.64 Setting the DataSource property value for the DataGrid

The final task is to add the code necessary the population the DataSets. We do this in the form's load event. Since the DataSet is associated with tables defined in two data adapters, we need to fill it from two sources. As seen in Figure 8.65, we do this with Fill() methods for each data adapter.

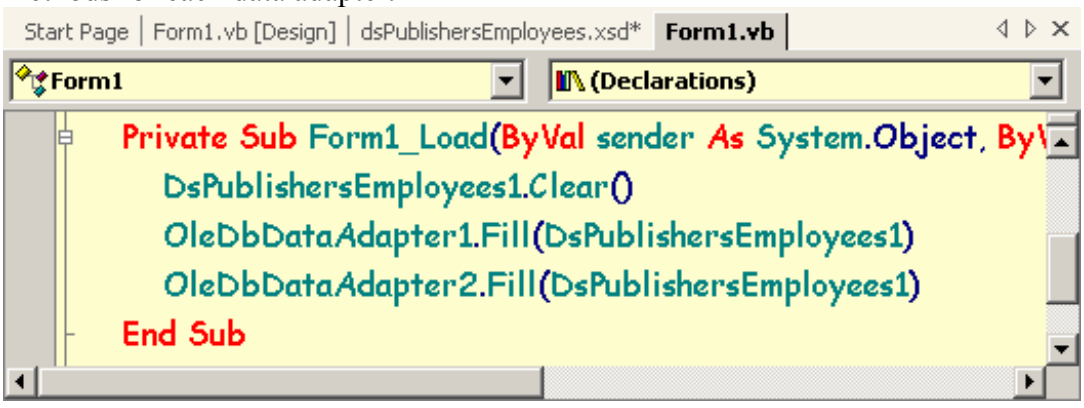


Figure 8.65 Code for Example 8.5

Exercise 8.12. Create a Windows Application similar to Example 8.5 that uses the "pubs" database and shows a DataGrid with records in the publishers table as the master records (you choose the fields to display) and records from the titles table as the detail records. For the detail records, display the title, price, advance, royalty, and pubdate fields.

Exercise 8.13. Create a Windows Application similar to Example 8.5 that uses the "pubs"database and shows a DataGrid with records in the stores table as the master records (you choose the fields to display) and records from the sales table as the detail records. For the detail records, display the ord_num, ord_date, and qty fields.

8.4 ADDITIONAL VISUAL BASIC .NET DATABASE TOOLS

In this section we look at using the Server Explorer (available under the View menu if it is not already visible) to work with databases. The Server Explorer (see Figure 8.66) provides easy access to both servers as well as Connections. In Figure 8.66, we see several Data Connections including the connection to the "pubs" MSDE Server database we having been working with in the previous section. We also see one server (the MSDE server that comes with Visual Basic .NET) named "PICO" (the author's computer).

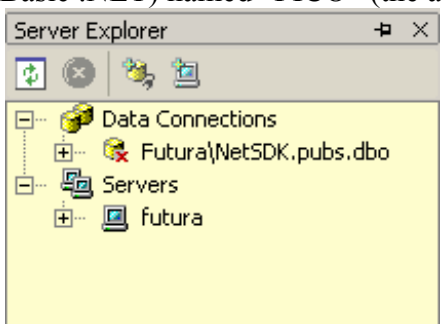


Figure 8.66 The Server Explorer

In this section, we show how to add a data connection to the server and then use this connection to create a data adapter and data set. In this case we add a data connection for a Microsoft Access 2000 database instead of an SQL Server database. We use an Exercise (Exercise 8.6) to demonstrate working with the Server Explorer.

Example 8.6 Using the Server Explorer to Process an Access 2000 Database

This example displays records from a Microsoft Access 2000 database named Listings.mdb, a database that stores real estate listings. Figure 8.67 shows this application at runtime.

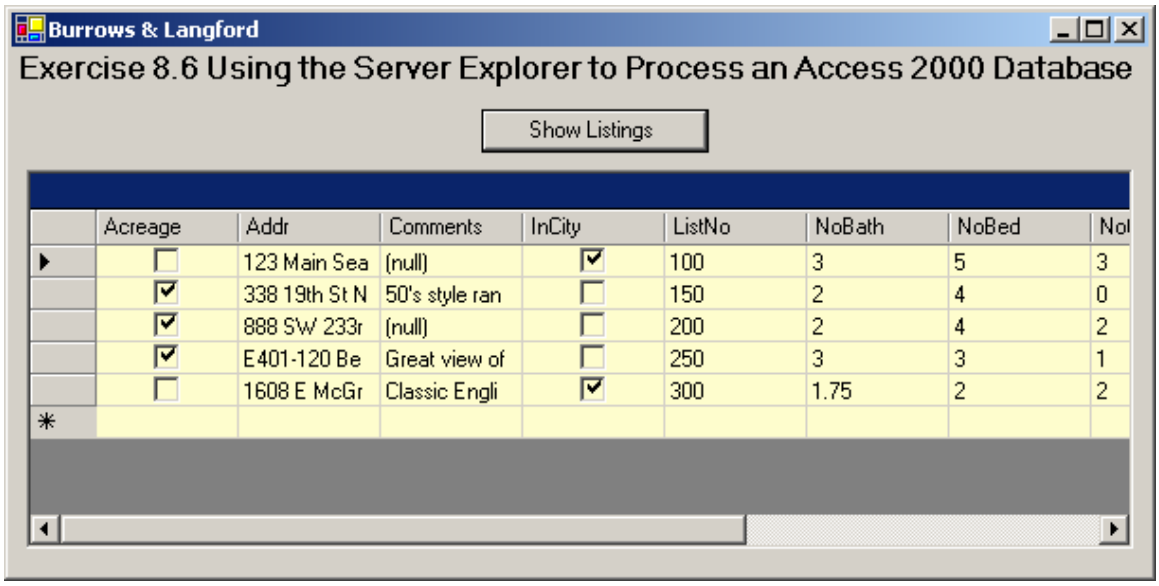


Figure 8.67 Example 8.6 at runtime

We begin by opening a new Windows Application project. Next we add a new data connection using the Server Explorer. Make sure the Server Explorer is visible by selecting Server Explorer from the View menu. Right-click on Data Connections and select Add Connection… from the popup menu as shown in Figure 8.68

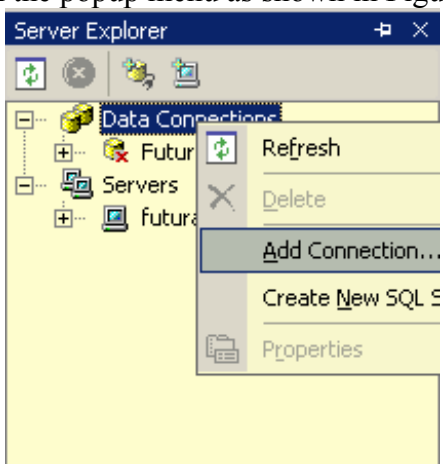


Figure 8.68 Popup menu generated by right clicking on Data Connections

You will next see the Data Link Properties dialog box as shown in Figure 8.69. Click on the Provider tab and select the "Microsoft Jet 4.0 OLE DB Provider". The "Jet" database engine is the technology used by the MS Access DBMS to manage its databases. If this provider is not displayed, then this means you will not be able to use MS Access databases on the specific computer.

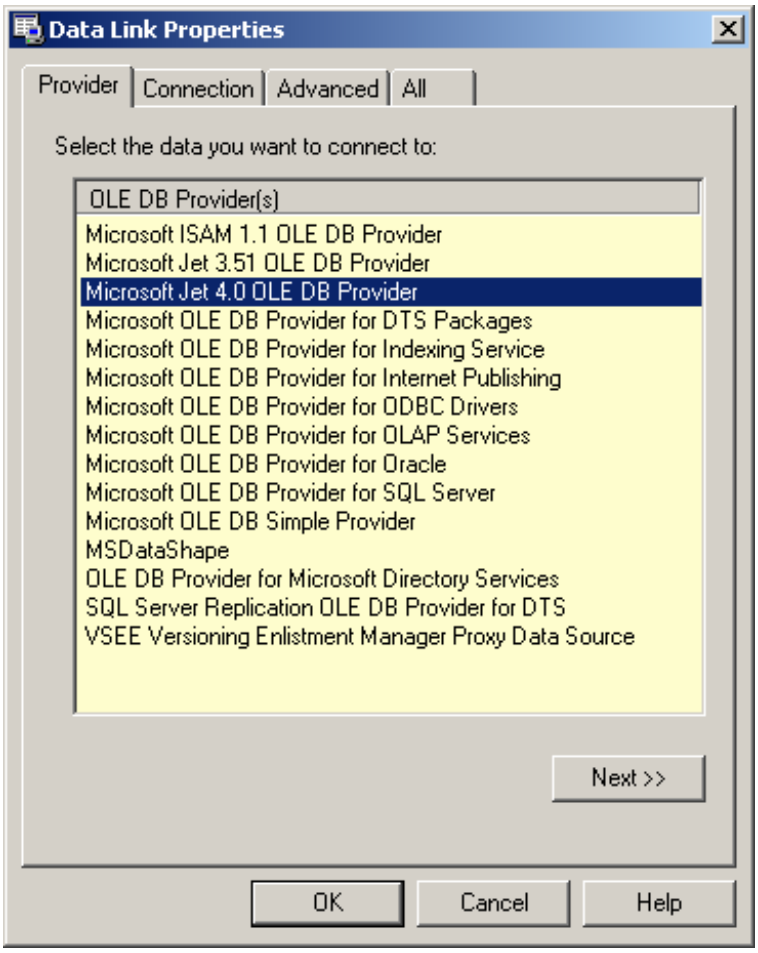


Figure 8.69 Selecting the Access Data Provider

After specifying the data provider, you next click on the Connection tab to specify the actual database information. As seen in Figure 8.70, the database (including its path) must be provided in box #1. Click on the ellipses (…) button to browse to find the database. If the database needs a user name and password, use box #2 to specify this information. When you are done, click on the Test Connection button to confirm that the connection parameters are correct. Click on OK when you are done.

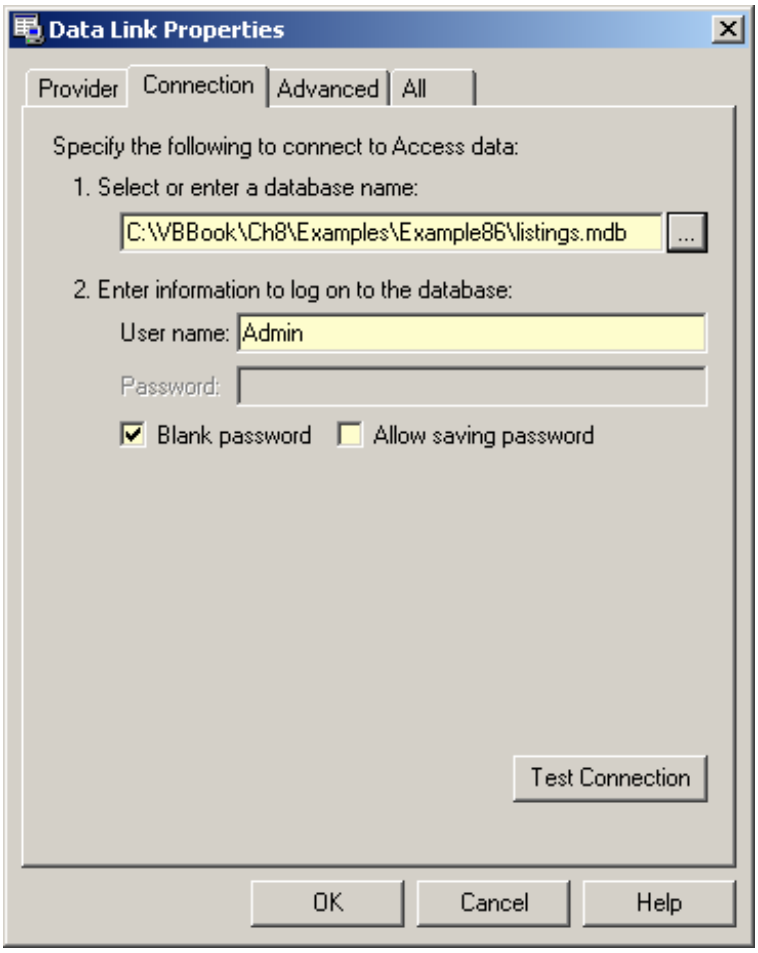


Figure 8.70 Specifying the actual database for the connection

When you are done, you should see that the connection has been added in the Server Explorer. Figure 8.71 shows what you should see.

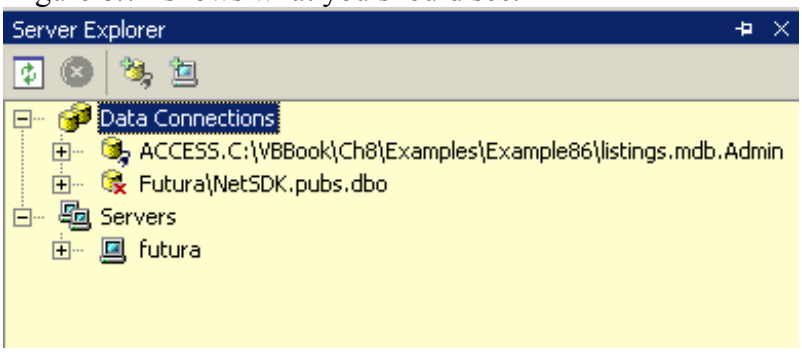


Figure 8.71 The new connection as seen in the Server Explorer window

You can use the Server Explorer to see how your database is designed. If you expand the data connection by clicking on the "+" symbol, you see more and more detail. Figure 8.72 shows the detail down to the field name level of the table named Listings.

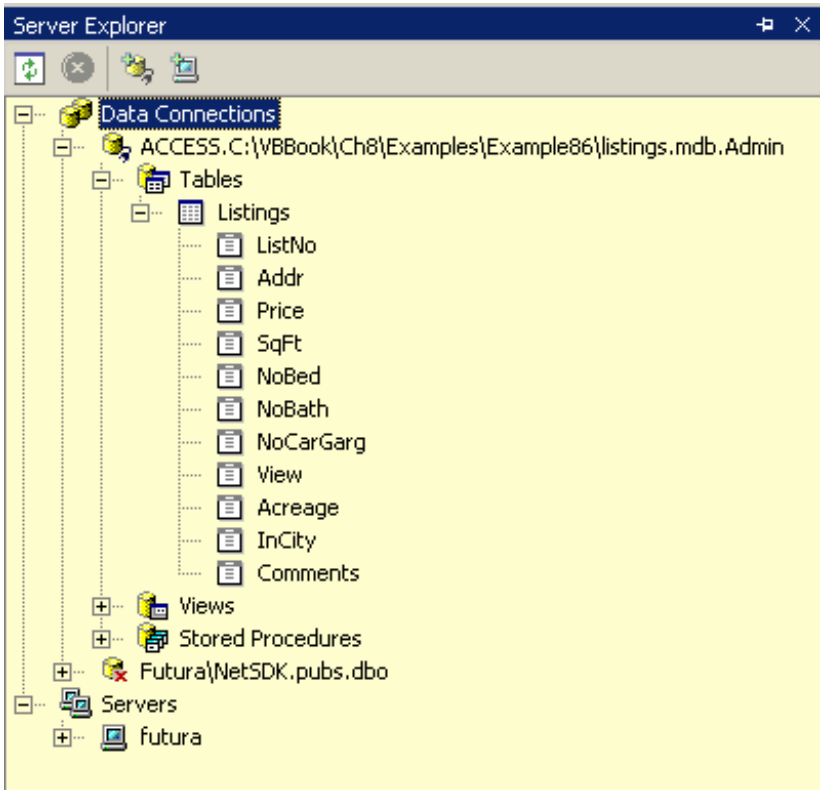


Figure 8.72 Viewing details by expanding the data connection

After you create the data connection, drag and drop it on your form in the designer window. This adds the connection to your form. You should see the data connection in the tray below the form in the designer window as shown in Figure 8.73.

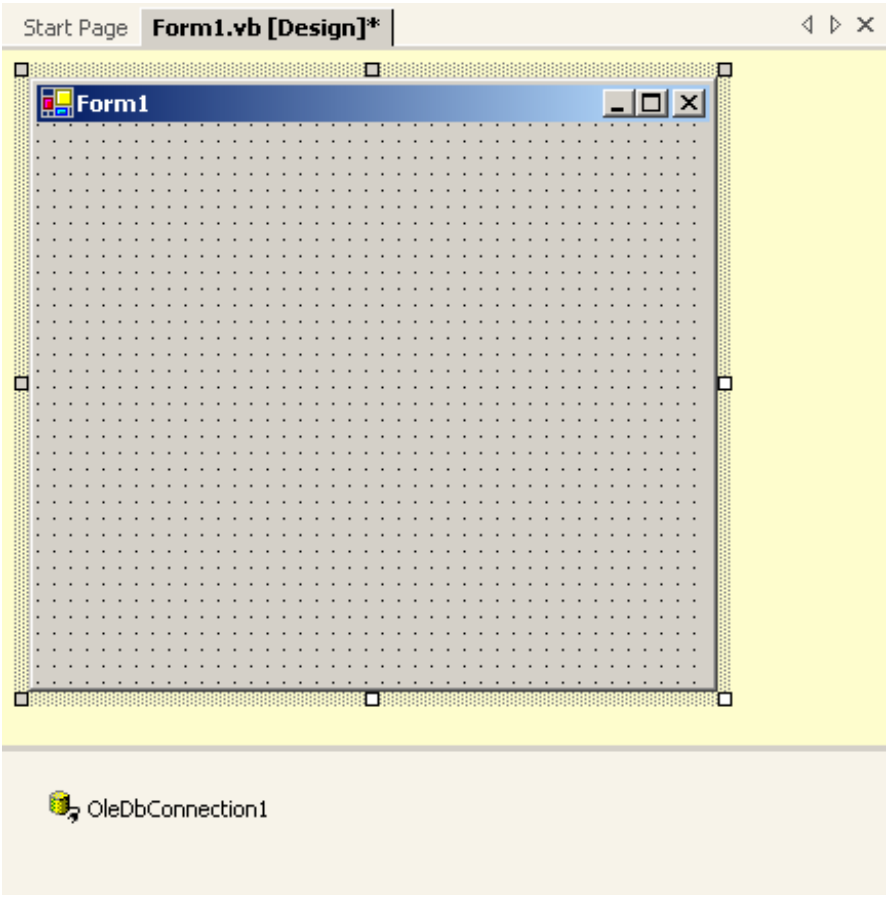


Figure 8.73 The data connection in the Designer Tray

Once we have a connection established, it's time to create a data adapter and data set just as we have done previously. We start by creating an OleDbDataAdapter using the Data Adapter Configuration Wizard. When we get to the Data Connection page, we choose the data connection just created as seen in Figure 8.74.

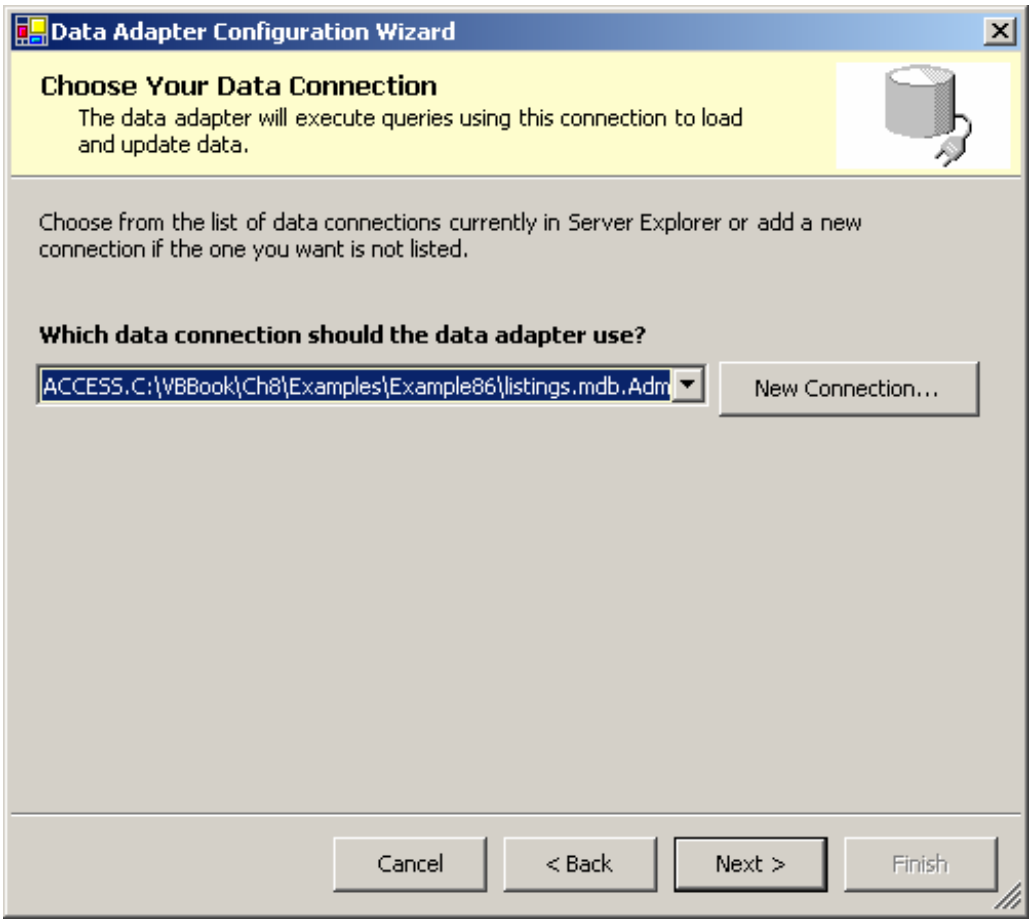


Figure 8.74 Selecting the data connection

We again use SQL to select the records to be used in the data set. Using the Query Builder, we select the check box to select all columns. You can see this as well as the SQL Select statement generated from the Query Builder in Figure 8.75.

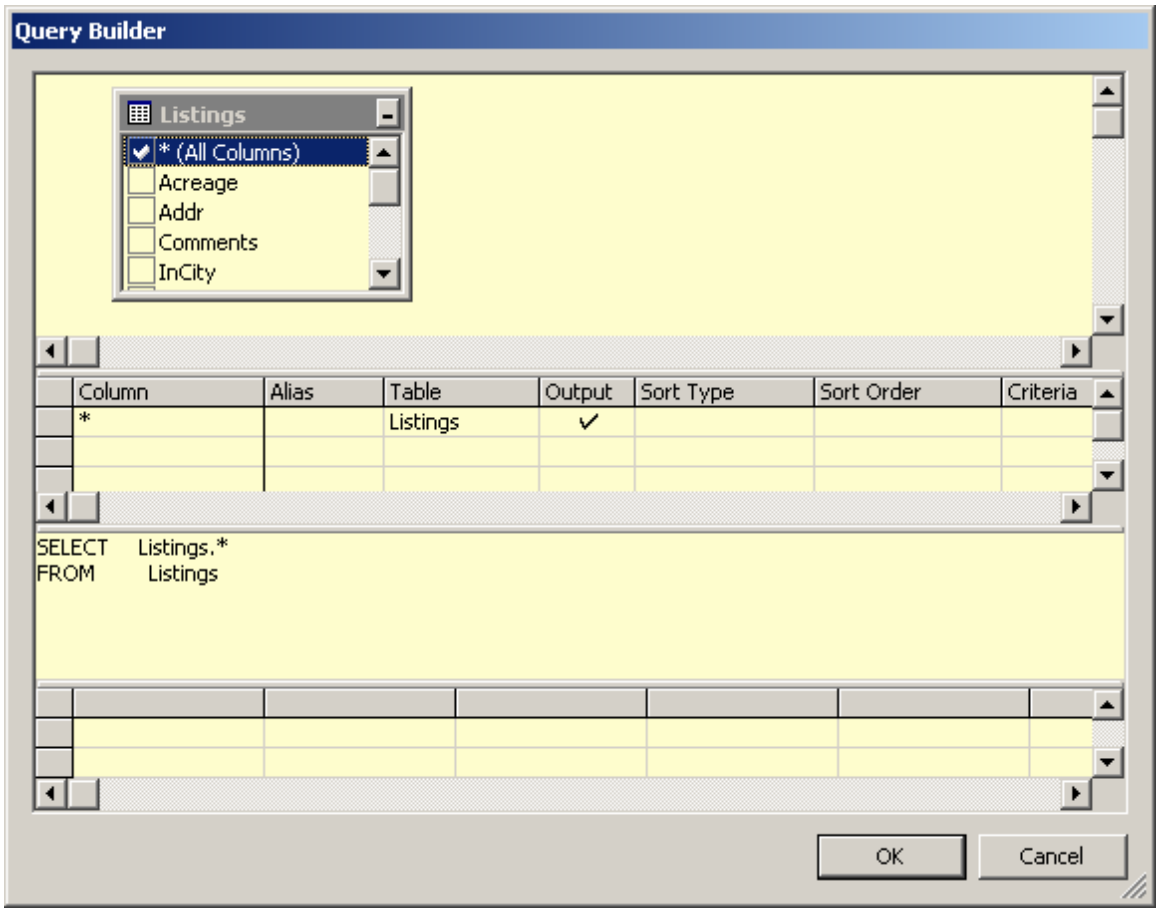


Figure 8.75 Selecting all the columns from the Listings table

After the OleDbDataAdapter is created, we need to create a new data set named dsListings by selecting Generate Dataset… from the Data window.

After creating the data set, we need to add a DataGrid and Button component to the form. The DataGrid's DataSource property value should be set equal to DsListings1.Listings.

Finally we need to write the code for the "Show Listings" button. Figure 8.76 shows the now familiar two lines that first clear the DataSet object and then fill it using the OleDbDataAdapter.

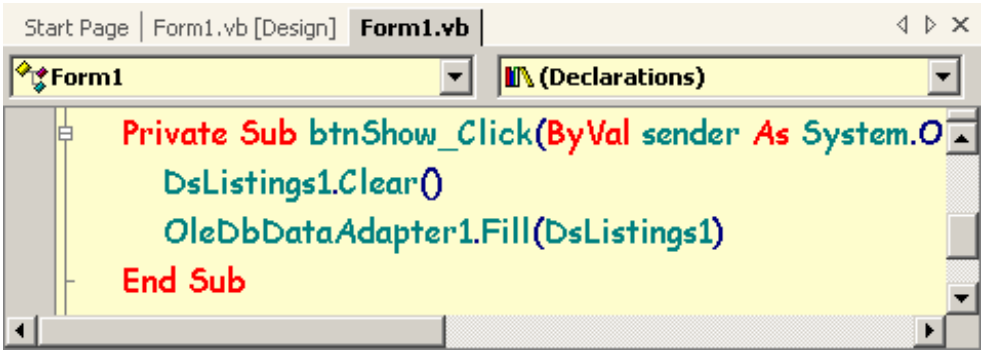


Figure 8.76 Code used to fill the DataSet object

There is another very useful feature called the Data Adapter Preview tool that you might find useful. This tool allows you to display the data associated with any data set defined for your project. Figure 8.77 shows two links at the bottom of the Properties window named “Generate Dataset” and “Preview DataSet”.

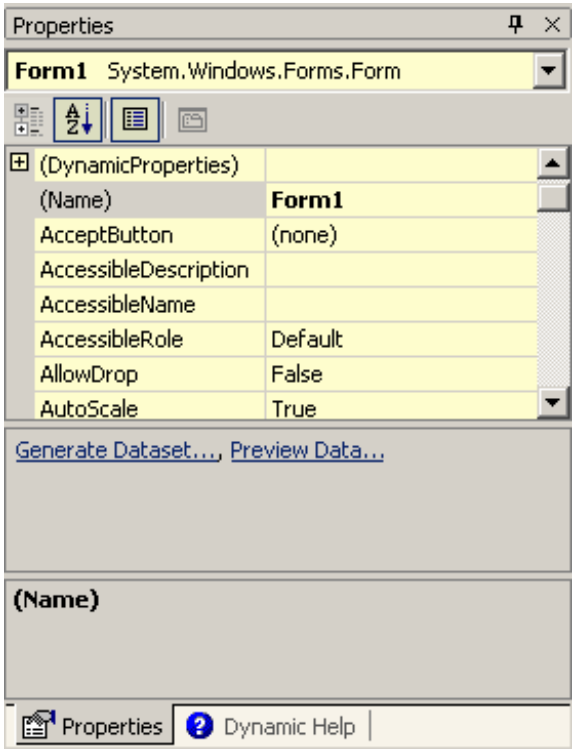


Figure 8.77 The Generate Dataset and Preview Dataset links at the bottom of the Properties window

Clicking on the Preview Dataset link brings up the Data Adapter Preview window as shown in Figure 8.78. You can use this to choose a data adapter (if your project has more than one) and a target dataset and table.

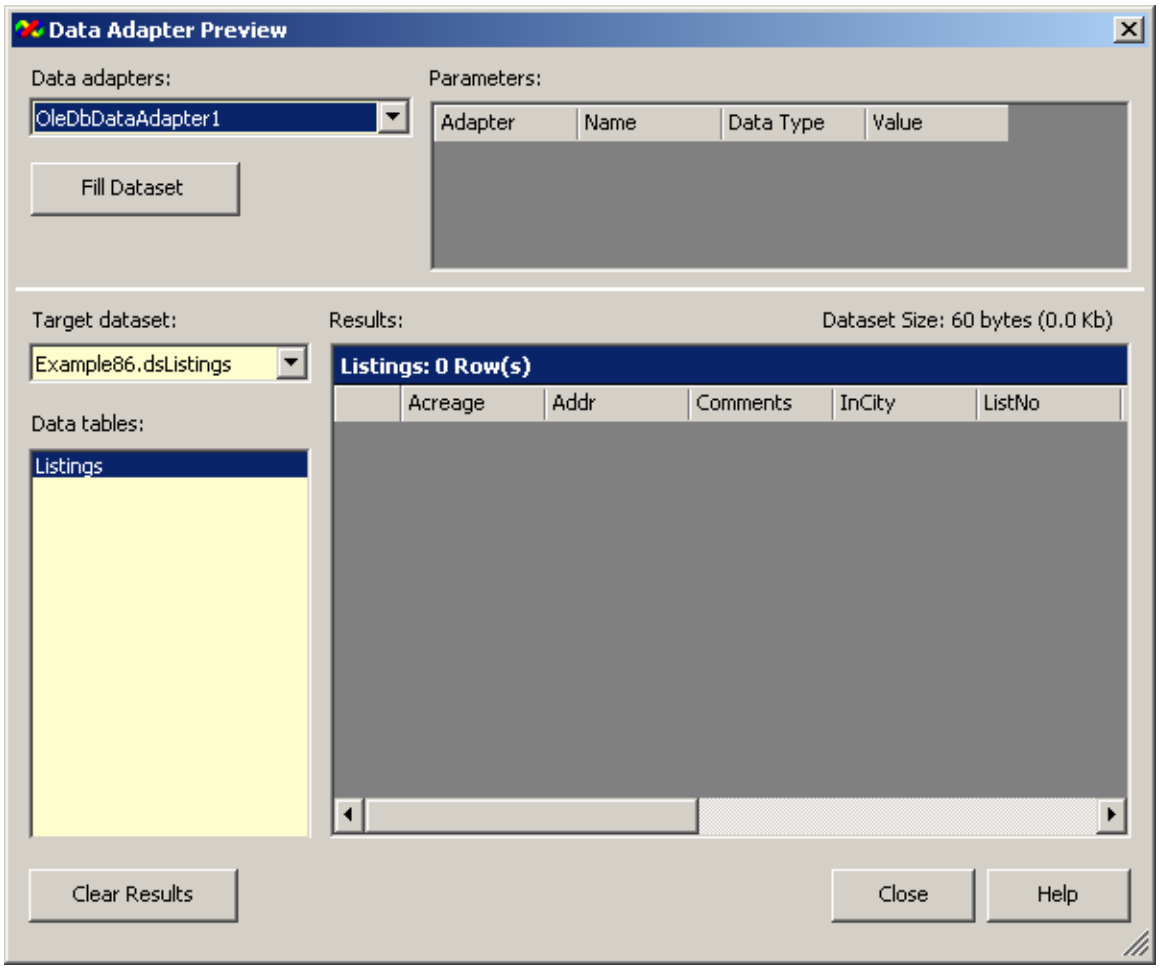


Figure 8.78 The Data Adapter Preview window to see the contents of a data set

Clicking on the Fill Dataset button then fills the data set and uses the Results grid to display the contents of the data set. Figure 8.79 shows this.

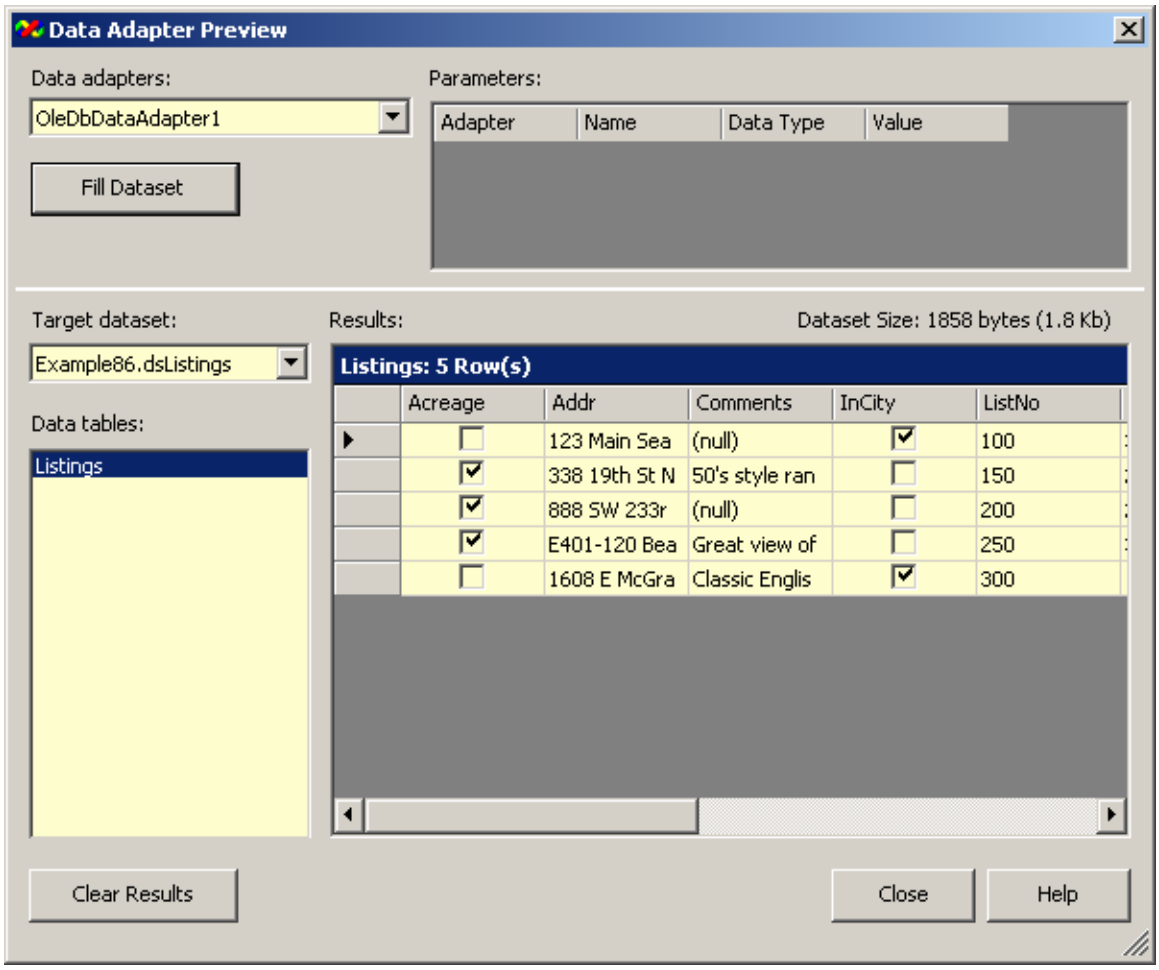


Figure 8.79 The results of clicking on the Fill Dataset button

8.5 PROJECT 9: REAL ESTATE LISTINGS DATABASE APPLICATION

Note that this project introduces new concepts related to database processing including adding new records and editing and deleting existing records. It also includes information on how a form can access data values found on another form.

Many database applications provide support for a company's operations. In this project we describe a database application that a real estate agent might use to access information on listings (properties and homes available for sale). This realistic example not only demonstrates database principles discussed in this chapter, but it also employs many of the controls and techniques covered in earlier chapters.

Description of the Application

Real estate agents require access to a large amount of data to perform their daily tasks. They often need to access this information from different locations—their office, their home, or the properties they are showing to clients. A laptop computer makes it possible for them to carry a computer-based database. What they need is an application that makes it easy to access the data.

Figure 8.80 shows such an application. It displays information about each listing (address, price, square feet, etc.) on the left side of the form. Controls on the right side allow the user to browse through the database record by record. Controls on the bottom

allow the user to perform standard database maintenance functions as well as displaying the payment calculator. Figure 8.81 shows the dialog box used to get information on a new listing and Figure 8.82 shows the calculator that is displayed when the user clicks the Compute Monthly Pmnt button.

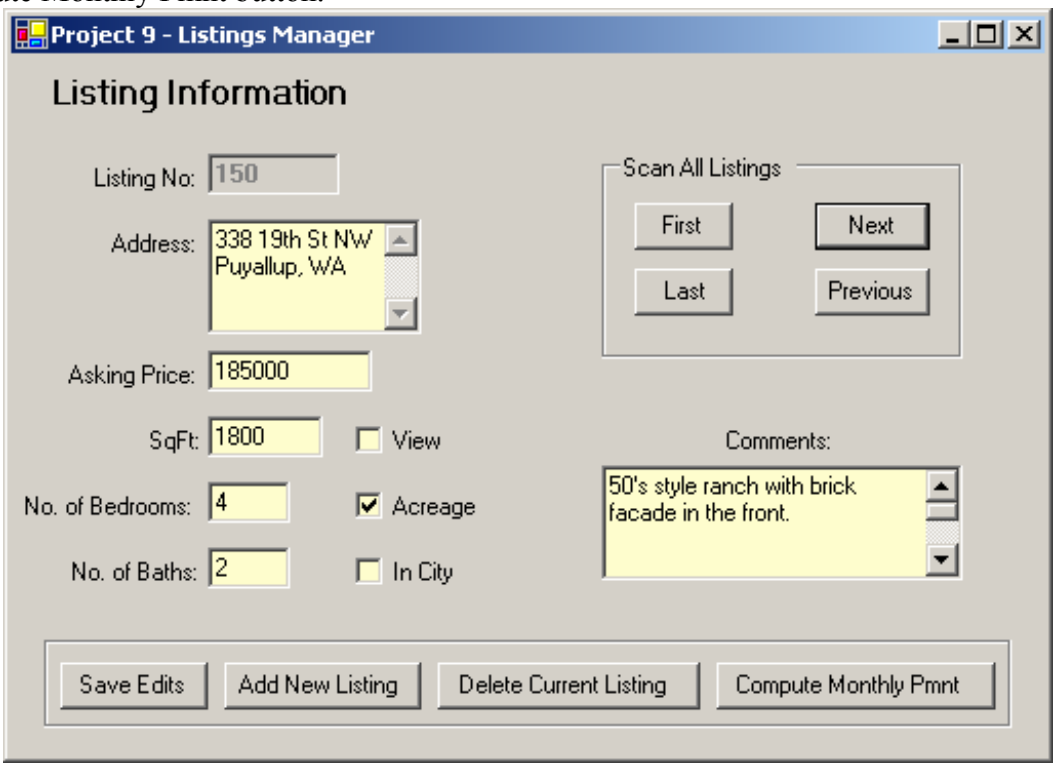


Figure 8.80 Real estate listings application at run time

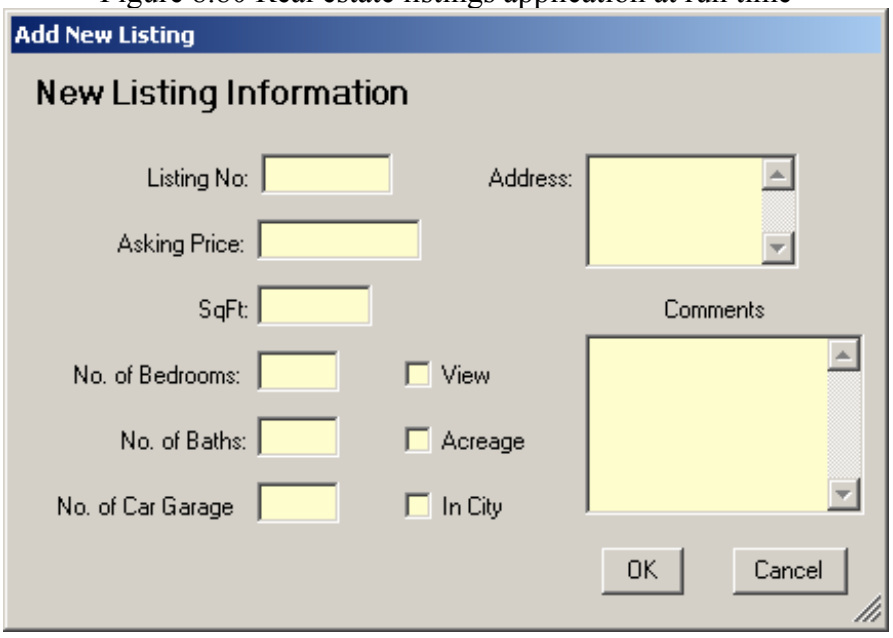


Figure 8.81 Add New Listing dialog at run time

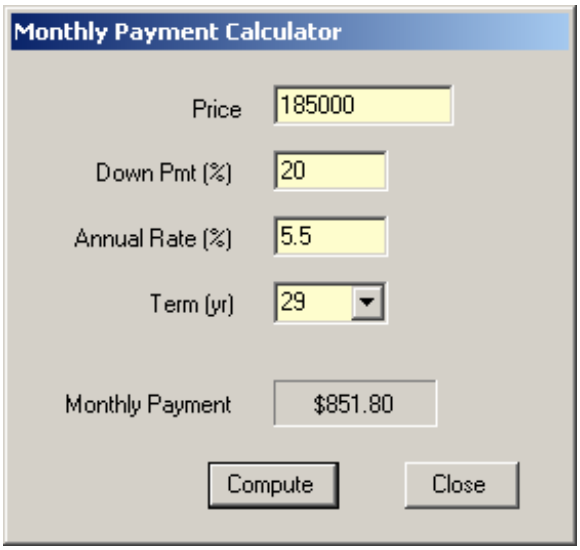


Figure 8.82 Monthly Payment Calculator at run time

In addition to the three functional forms shown previously, the application also shows a number of confirmation messages. Figure 8.83 shows message boxes with these confirmation messages.

<combine following images into one figure>

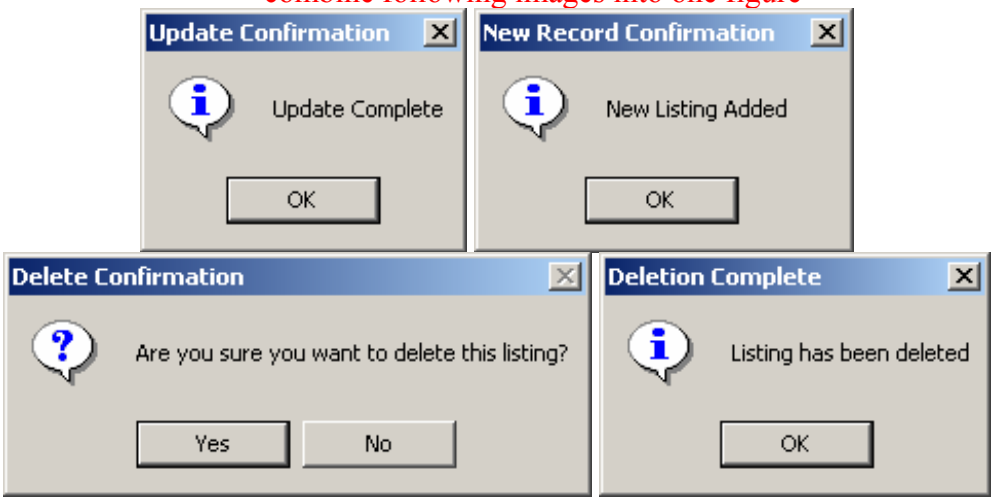


Figure 8.83 Various confirmation messages

Design of the Application

The application is built around a single-table database created with Microsoft Access. This database is called Listings.mdb and its single table is called Listings. Table 8.x shows the fields in the Listings table. Note that the "Type" column indicates the type as defined within Microsoft Access, not Visual Basic .NET. You will need to create a connection for this database (see Section 8.4). The main form for the application has an OleDbDataAdapter associated with the Listings.mdb connection and an SQL Select statement that accesses all the fields from the Listings table. It also has a data set that is associated with the data adapter. All the data are presented via bound controls (see Example 8.3). The check boxes (View, Acreage, and In City) are bound to fields that are

type Yes/No in the Microsoft Access database. The values Yes and No in Access are the same as the Boolean values True and False in VB.

TABLE 8.x Fields in real estate Listings table

Database Field	**Access Type**	**Description**
ListNo	Text	Listing number
Addr	Text	Address
Price	Currency	Asking price
SqFt	Integer	Number of square feet
NoBed	Integer	Number of bedrooms
NoBath	Single	Number of bathrooms
NoCarGarg	Integer	Garage space for cars
View	Yes/No	Is there a view
Acreage	Yes/No	Is there acreage
InCity	Yes/No	Is house in the city
Comments	Text	Misc. comments

Let us turn our attention to the "Scan All Listings" command buttons on the right side of the form. These controls move through the records of the database as indicated by the button captions.

- **First**: Sets the Position property of the dataset to the first record (a value of 0).
- **Last**: Sets the Position property of the dataset to the last record (a value equal to the Count property minus 1).
- **Next**: Increases the value of the Position property by the value 1 (see Example 8.3).
- **Previous**: Decreases the value of the Position property by the value 1 (see Example 8.3).

In addition to the main form, you should create a second form (dialog box) that enables the user to add a new listing (see Figure 8.81). This form needs to include a way for the information entered by the user to be available to the main form where that database management activities take place. We will facilitate this by adding some public functions that return the values. We also need to be able to tell which of the two buttons (OK or Cancel) was clicked from the main form so we know what to do when the user is finished with the Add New Listing dialog box. These tasks are discussed later.

Finally, you should create a third form that enables the user to calculate the monthly payments for a loan (see Figure 8.82). The Price field should be taken from the Asking Price text box on the main form. That is, the Text property of the Price text box should automatically be set before the form is shown. Recall from Chapter 3 that to access a control on another form you must qualify the control name with the name of the form on which it resides. For example, if you are writing code for frmMain that has a TextBox component named txtPrice, and you want code in frmMain to it store the value of txtPrice's Text property in the component txtCost that is on the form referenced by ObjCalc, then you could enter following:

```
ObjCalc.txtCost.Text = txtPrice.Text
```

The Close button hides the form. The Compute button takes the purchase price, down payment percentage, annual interest rate, and the term from the TextBox

components and dropdown ComboBox component, then computes and displays the monthly payment. The interest rate and down payment percent are entered as whole numbers, e.g., 20 is entered for 20%. This means that your code needs to divide these two values by 100 to move the decimal point in the right location (20/100 = .20).Compute the loan amount by subtracting the down payment from the purchase price. Then use the Pmt() function to compute the monthly payment.

```
Payment = -Pmt(IntRate / 12, Term * 12, LoanAmt, 0, 0)
```

The Term combo box should include the values 10, 15, 20, 25, 29, and 30 set up at design time.

Construction of the Application

Start by creating a Windows Application. You then need to add the connection, data adapter, and dataset. The database, Listings.mdb, is stored in directory C08/Proj9 in the code package for this text. See Example 8.6 for details on this step. Next add a few of the bound text boxes like the ones for the listing number and address. Set their (DataBindings) Text property to the appropriate field from the dataset (see Example 8.3). Using the form's load event, add the code to fill the data adpter and then run the application. You should see the values from the first record in the database displayed in these bound controls.

When you get the first few bound controls working, add a few more and get them working. Continue by setting up the remainder of the controls that are bound directly to the database.

Now turn your attention to the navigation controls on the right side of the form. Again, add one button (the First button in the Scan All Listings section would be a good place to start), then code and test it. Continue, one button at a time, to add and test. Use the descriptions provided earlier to guide you in writing the code.

Remember, limit what you do at each step and thoroughly test what you've done before you proceed. The strategy of doing a little coding and then testing will limit the number and complexity of the problems you encounter. It also increases the chances you'll be able to correct the problem quickly.

Once you have the navigation buttons working, it's time to work on the functionality of the database maintenance buttons and the monthly payment compute button at the bottom of the form. We start with the "Save Edits" button. The user can make changes to any of the fields except the Listing Number (that text box should not be enabled). As these changes are made, the underlying dataset is also changed because of the controls that are bound to it. However, to cause these changes to be made to the actual database, the data adapter must execute its Update method. For example, if the data adapter is named OleDbDataAdapter1 and the data set is named DsListings1, then the statement:

```
OleDbDataAdapter1.Update(DsListings1)
```

should cause the database to be updated. However, before this works, the system must move to a new different record. Instead of having the user move to a different record, we add that logic to the code for the Save button's click event. Figure 8.84 shows this navigation code as well as the Update method.

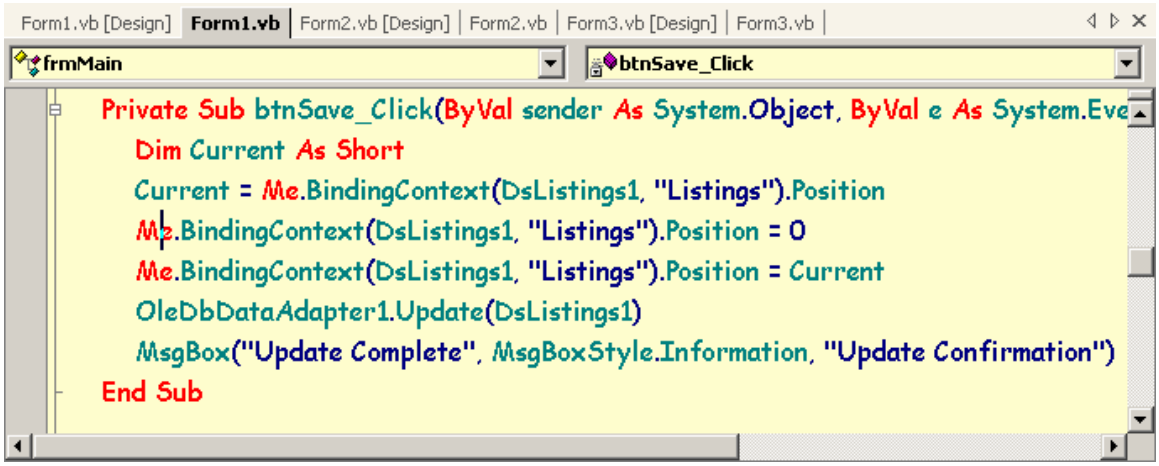


Figure 8.84 Code to update an existing record.

Next we turn our attention to the Add New Listing button. This button causes the Add New Listing dialog box, shown in Figure 8.81, to be displayed. The tricky part about this task is providing the mechanism for the main form to be able to get the data entered by the user on the dialog box. One strategy is to add a set of public functions (methods) to the class definition for the Add New Listing form that provide the access to the TextBox components. For example, consider the code in Figure 8.85. This function returns the value the user entered in the TextBox txtListNo on the Add New Listing form.

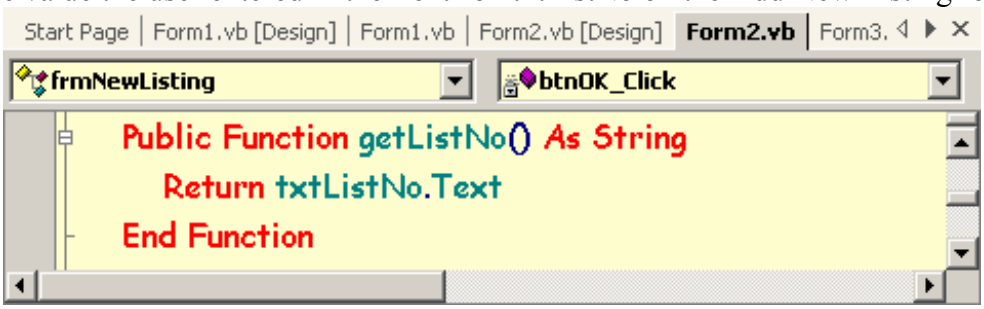


Figure 8.85 Public function that returns the user-entered value in txtListNo

You need to add a function like this for all the components used by the user to define a new listing such as the asking price, square footage, etc. In the main form you can use these functions such as:

```
ListNo = ObjNewListing.getListNo
```

to get the value (assuming the Add New Listing form is referenced by the object reference ObjNewListing).

To determine which button the user clicked, you can use a variable and set it equal to one of two values depending on which button was pressed. Consider the code in Figure 8.86. In this code the variable `ButtonClicked` is set to either the constant OK (=1) or CANCEL (=2) in the btnOK_Click or btnCancel_Click events. The function getButton can then be used to return the value of the variable.

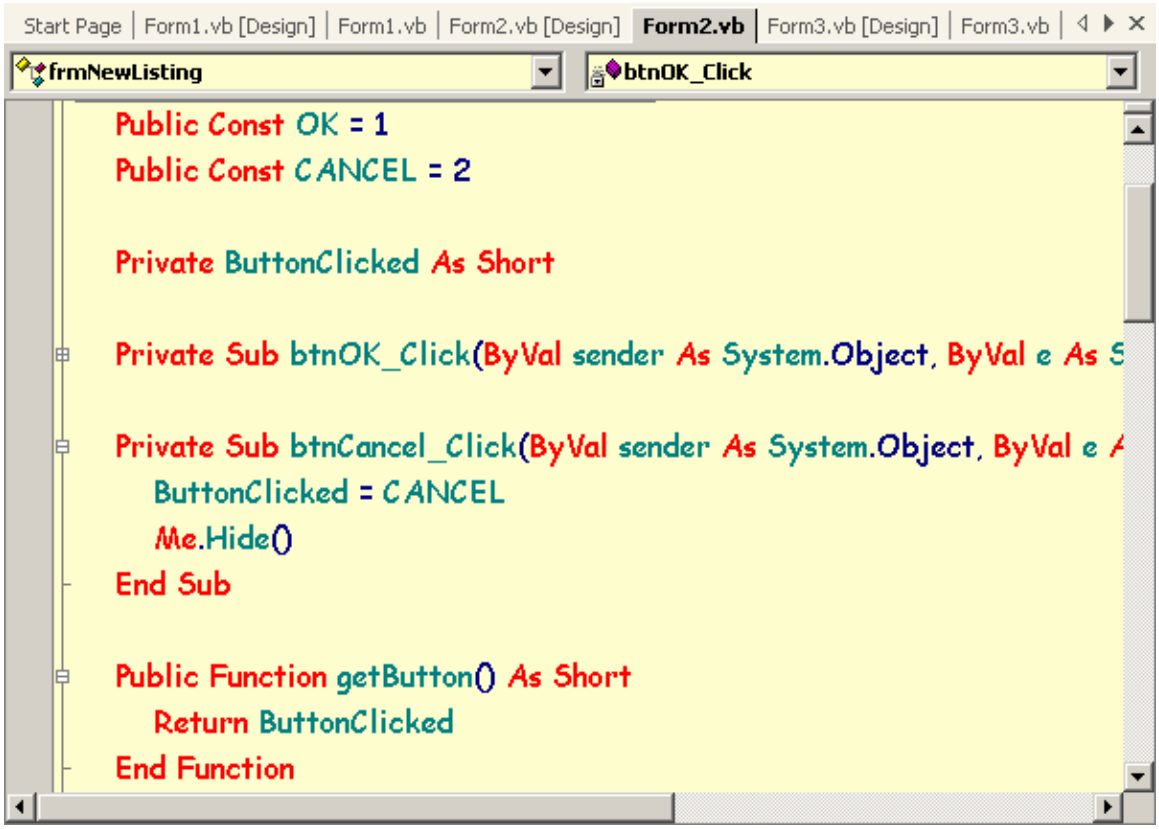


Figure 8.86 Determining which button was pressed

In the main form the getButton function can be used to determine which button was clicked as the segment of code below shows.

```
If (ObjNewListing.getButton() = ObjNewListing.CANCEL) Then Exit Sub
```

The btnOK_Click event should verify that the data are complete and that the numeric fields contain valid numbers. Use message boxes to notify the user if an error is detected (see Figure 8.87).

<combine images below into one figure>

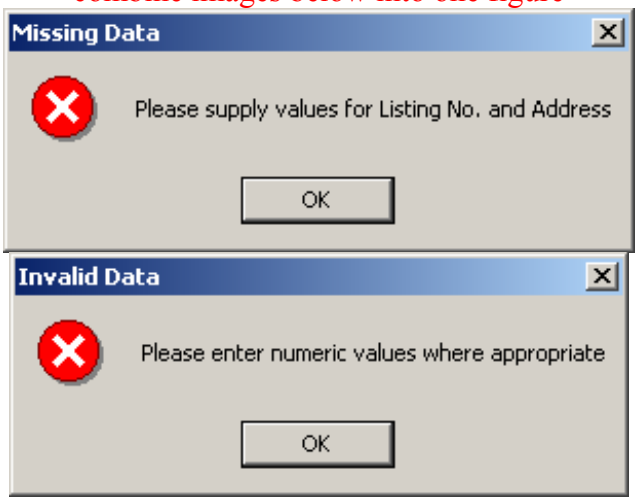


Figure 8.87 Message boxes indicating missing data or non-numeric values on the New Listing form

Once the user has correctly filled out the Add New Listing form and clicked on the OK button, a new record needs to be added to the database. Using the dataset's Add method followed by the data adapter's Update method does this. Figure 8.88 shows the code that performs these two steps.

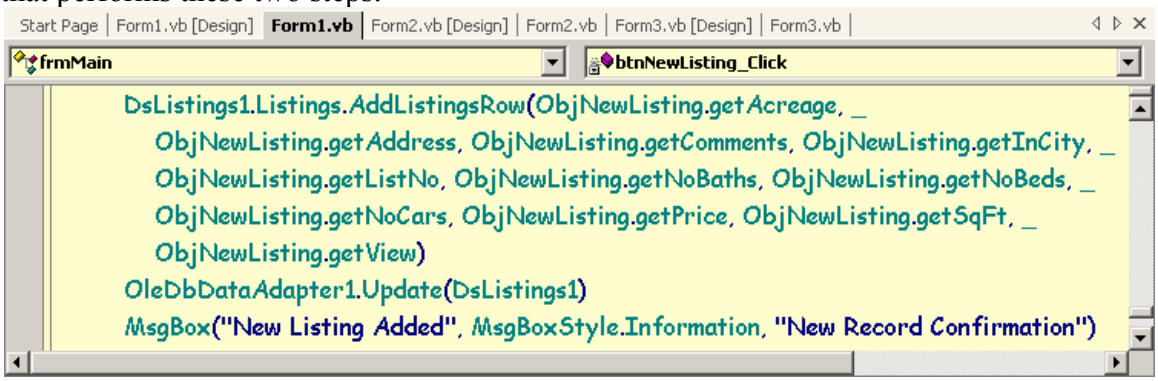


Figure 8.88 Statements used to add a new listing to the database

Be aware that the code

```
DsListings1.Listings.AddListingsRow( …
```

is formed using the name of the database table associated to the dataset. In the case above, the table is named "listings". If the table had been called "stuff", the statement would have been:

```
DsListings1.Stuff.AddStuffRow( …
```

The final button we need to add is the "Delete Current Record" button. To delete the current record, you first need to determine the index of the current record by accessing the Position property of the dataset. This value is then used to delete the record from the dataset. Finally, the database is updated using the data adapter's Update method. Figure 8.89 shows the code to perform these tasks.

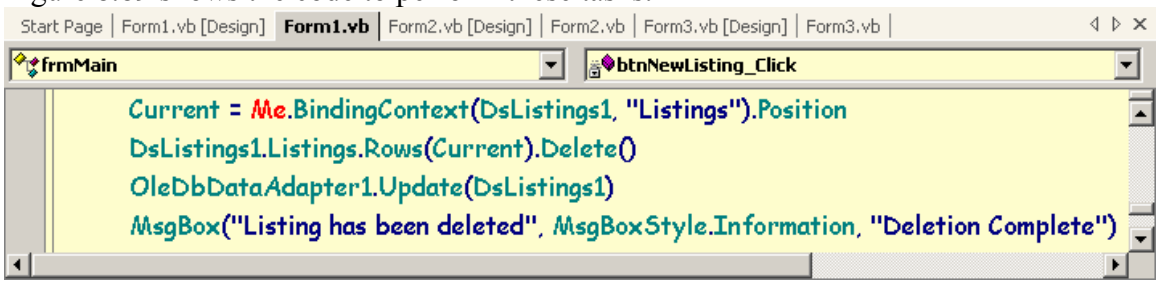


Figure 8.89 Code used to delete the current record

Chapter Summary

1. A database is an organized collection of data and relationships that describe entities of interest to a business. An entity is a thing, such as an employee, a customer, a product, or a part. Examples of relationships are (a) the parts used to assemble a product and (b) the set of customers an employee is responsible for. Databases are a very important part of most business data processing, and Visual Basic .NET is a very important tool used by businesses to access the data in their databases.

2. An entity-relationship diagram (ERD) documents entities and their relationships. This diagram indicates cardinality—the number of one entity that can be related to another entity. For example, an ERD might indicate that a customer is related to one employee and an employee is related to many customers. This would be an example of a one-to-many relationship.
3. Visual Basic .NET supports the relational database approach for organizing data about entities and their relationships. In a relational database, the data for each entity are stored in their own two-dimensional table. Each table may include a key field that has unique values from one record to another. Relational databases use foreign keys and correlation tables to link tables according to the relationships between the entities.
4. Database designers use a process called normalization to place data items into tables in a way that minimizes the problems of redundant data while maintaining the correct relationships between the entities. The data stored in several related tables of a relational database can be used to answer a question using an SQL (Structured Query Language) query.
5. Visual Basic .NET uses ADO.NET as the technology to access and manage databases. The ADO.NET object model includes the data provider classes for DataAdapters and Connections. It also provides the DataSet class that contains a set of "disconnected" tables that are associated with data providers. These classes provide the ability to select existing records for display, deletion, and modification as well as adding new records to a database.
6. The DataGrid control is a control that can be bound to a specific data set. It provides a row and column display of records and fields from a dataset and also provides a rich set of properties that control the appearance of the data displayed in the grid.
7. Parameterized queries provide for the ability to determine how records will be selected at runtime. The use of parameterized queries allows values entered by the user as the program executes to be used to find a set of records from the database.
8. Many of Visual Basic .NET's controls can be bound to a dataset and its fields. Simple data binding allows the property of a control to be bound to a single data element such as a single field within a DataSet. Complex data binding allows the property of a control to be bound to more than one data element in a DataSet.
9. ADO.NET supports master/detail record relationships where one master record is associated with zero or more detail records. The DataGrid supports this type of relationship by including an expander symbol "+" to expand and collapse detail records.
10. The Server Explorer provides access and management tools for connections and servers.

Key Terms

ADO.NET	database management system	foreign key
associative object	DataGrid	intersection table
BindingContext	DataProvider	key field
cardinality	DataRelation	master/detail DataSet
complex data binding	DataSet	normalization

compound key	DBMS	parameter
Connection	Distinct	relational database
correlation table	entities	relationships
data binding	entity-relationship diagram	simple data binding
DataAdaptor	expander	SQL
database	filter	structured query language
		third normal form

End-of-Chapter Problems

1. Describe the association between entities and relationships. Give examples of each.
2. A table consists of rows and columns. What type of column is used in a source table to provide a one-to-many relation? What type of column in a destination table to reference a value in the source column?
3. Name and describe the statement that is used to retrieve current data and place it in a DataSet. What statement is used to send modifications made to the DataSet back to the data source?
4. Classify the following controls as simple bound controls or complex bound controls: TextBox, ComboBox, ListBox, CheckBox, RadioButton, and DataGrid. Explain the difference between the two types of controls.
5. Name the two properties that must be set on a DataGrid control to establish a relationship to the data source and describe the function of each.
6. How is an intersection table used in a complex query?
7. Can the Query Builder be used to create a query that contains a relationship between two or more tables? If so, does it provide the SQL code that represents this relationship that may be edited?
8. The DataGrid control can be used to display the Master and Details of a data set. Explain what a Master/Detail relation represents. Provide an example.
9. What does the acronym SQL stand for? Write out the SQL Select statement generated by the Query Builder used to return all of the data from the publishers table.
10. Would use of the "Distinct" keyword affect the DataSet returned by a query? Explain.

Programming Problems

1. Create a simple Visual Basic.NET database application that displays the titles of books available in the pubs database using a DataGrid control. From the "titles" table, display the columns for title id, title, price, notes, and publication date in the grid. You will need to create a connection, a SQL data adapter, and generate a DataSet. Your form load event should initialize the data set and fill the data adapter.
2. The flexible display nature of the DataGrid control provides the perfect method of browsing several tables on a data source. Create a form that contains a ComboBox control with selections for the Authors, Employee, and Publishers tables. The user selection in the ComboBox should fill the DataGrid with the appropriate information.

   You will need to create three DataAdapters and three DataSets, one for each table reference. The SelectedIndexChanged event of the ComboBox control can be

used to re-assign the properties of the DataGrid to display the proper table. Setting the DataGrid to a specific table requires modification of the same properties that you've set manually with the Properties window in this chapter. For example, to view the authors table, the program would set the DataSource property of the grid to equal DsAuthors1, the DataMember property to equal "authors", clear the data set, and then fill the data adapter.

3. In this chapter, you learned how to display a master/detail relationship using a DataGrid. With a little extra work, it is possible to show the master records in a list box and allow the user to select a ListBox item which displays the detail records in a DataGrid on another part of the form. Create an application that displays the titles of all of the books in the pubs database in a ListBox and allows the user to select a title to display all of the sales records associated with that title in a grid.

   You will need a form that holds a ListBox control and a DataGrid control below it. Data adapter for the sales and titles tables will be needed. When you generate a DataSet that includes both of these tables, an XSD file will be created. You will need to set a relation between these tables as demonstrated in the chapter except that when the relation dialog is displayed, set the Parent element combo box to titles and the Child element to sales.

   The DataSource property of the ListBox control should be set to the data set and the DisplayMember property to the title column of the data (titles.title). To connect the DataGrid to the current record selected in the ListBox, set the DataSource property of the grid to the DataSet (DsTitleSales1) and the DataMember to the name of the relation between the two source tables (titles.titlessales). Make sure that the form load event clears the data set and then fills both data adapters from it.

4. Many database applications, particularly financial, use information from a data source to generate a report that includes calculations and summary data. Create an application that displays fields for book title, price, royalty, advance, and year-to-date sales from the titles table that are bound to read-only TextBox controls.

   From the information in these TextBoxes, calculate the Total Author Earnings on the book so far this year and the Amount of Credit that remains to be paid to the author and display these figures in label controls. To obtain the total earnings, you will need to multiply the royalty percentage by the price of each book and then multiply that number by the number of books sold in the year to date. From the earnings figure, subtract the advance already paid to the author to determine the amount currently owed. Some of the books in the pubs database do not have values entered for price, royalty, etc. so be sure to include error trapping so your calculations don't fault the program. Also add Next and Previous buttons to the form to allow the user to page through the available title records.

5. Create an application that allows the user to edit name, address, city, state, zip, and phone of records from the authors table displayed in TextBox controls. Include buttons for Update, Discard Changes, Next, and Previous. By default, the Update and the Discard buttons should have their Enabled properties set to False. The activation of the TextChanged event on any of the TextBox controls should enable these buttons.

The Update button should use the EndCurrentEdit method (Me.BindingContext(DsAuthors1, "authors").EndCurrentEdit()) to store the changes to the dataset, execute the Update method on the data adapter, and disable this button and the Discard button. The Discard button can use the CancelCurrentEdit method on the binding context and then disable itself and the Update button. Both the Next and the Previous buttons should cancel the current edit (just in case changes were made), change the position of the current record, and then disable the two modification buttons (since the position change will execute the TextChanged events).

6. When a search of records fails (no records meet the search criteria), the application usually warns the user that no records were found. Create an application that uses a parameterized query on the publishers table to allow the user to determine which publishers are located in a specified state. Accept the two-letter state abbreviation input from a TextBox control and use the parameterized query to fill a DataGrid control. If no records were returned by the query, make the DataGrid invisible (using the Visible property) and display a message box alerting the user to the search failure. If records have been returned, make the grid visible to display the records.

   To determine the number of records generated by a query, the Fill method returns an Integer value of the number generated by the fill when it completes execution. You can write a simple statement to catch the result of this method (num = daPublishers.Fill(DsPublishers1)) and then check if that result is zero. Note: The parameterized query example in the chapter works perfectly for an OleDB data source, but if you use the SqlAdapter, you will need to give the query parameter a specific name (such as "@state") in the Select statement. Set the named parameter in the same manner as shown in the chapter (such as .Parameters("@state").Value)