CHAPTER 24

Understanding Visual Basic Fundamentals

In this chapter, you'll learn how to create, edit, and test Microsoft Visual Basic code in your Microsoft Access 2010 applications. The chapter covers the following major topics:

- The Microsoft Visual Basic Editor (VBE) and its debugging tools
- Variables and constants and how to declare them
- The primary object models defined in Access—the Access model, the Data Access Objects (DAO) model, and the ActiveX Data Objects (ADO) model. You'll need to understand these models to be able to manipulate objects such as forms, form controls, and recordsets in your code.
- Visual Basic procedural statements:
    - Function and Sub statements
    - Property Get, Property Let, and Property Set (for use in class modules) statements
    - Flow-control statements, including Call, Do, For, If, and Select Case
    - DoCmd and RunCommand statements
    - On Error statements
- A walkthrough of some example code you'll find in the sample databases

If you're new to Visual Basic, you might want to read through the chapter from beginning to end, but keep in mind that the large section in the middle of the chapter on procedural statements is designed to be used primarily as a reference. If you're already familiar with Visual Basic, you might want to review the sections on the VBE and the object models, and then use the rest of the chapter as reference material.

Note

You can find many of the code examples from this chapter in the modExamples module in the Contacts.accdb and Housing.accdb sample databases on the companion CD.

The Visual Basic Development Environment

In Access for Windows 95 (version 7.0), Visual Basic replaced the Access Basic programming language included with versions 1 and 2 of Access. The two languages are very similar because both Visual Basic and Access Basic evolved from a common design created before either product existed. (It's called Visual Basic because it was the first version of Basic designed specifically for the Windows graphical environment.) In recent years, Visual Basic has become the common programming language for Microsoft Office applications, including Access, Microsoft Excel, Microsoft Word, and Microsoft PowerPoint. Some of the Office 2010 system products (including Word 2010 and Excel 2010) can work with an even newer variant of Visual Basic—VB.NET—but Access 2010 does not.

Having a common programming language across applications provides several advantages. You have to learn only one programming language, and you can easily share objects across applications by using Visual Basic with object automation. Access 2010 uses the VBE common to all Office applications and to the Visual Basic programming product. The VBE provides color-coded syntax, an Object Browser, and other features. It also provides excellent tools for testing and confirming the proper execution of the code you write.

Modules

You save all Visual Basic code in your database in modules. Access 2010 provides two ways to create a module: as a module object or as part of a client form or client report object.

> **Note**
>
> **You cannot create a module as part of a web form or web report object in a web database. You can, however, create module objects and modules as part of client forms and client reports in a web database. Web objects do not support using Visual Basic so for the purposes of the discussions in this chapter, you can assume we are always referring to client forms and client reports.**

Module Objects

You can view the module objects in your database by clicking the top of the Navigation pane and then clicking Object Type under Navigate To Category. Click the Navigation Pane menu again, and click Modules under Filter By Group. Figure 24-1 shows the standard and class modules in the Conrad Systems Contacts sample database—Contacts.accdb. (We also right-clicked the top of the Navigation pane, clicked View By on the shortcut menu, and then Details on the submenu so you can see the descriptions we've attached to all the modules.) You should use module objects to define procedures that you need to call from queries or from several forms or reports in your application. You can call a public procedure defined in a module from anywhere in your application.

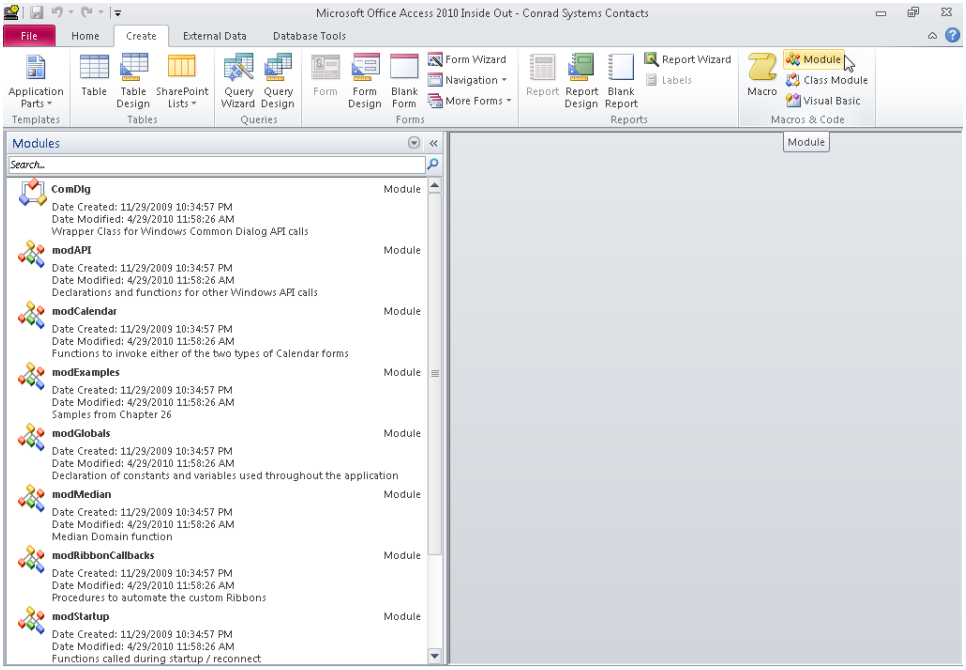

Figure 24-1 To see all the modules in your database, click Modules under Filter By Group on the Navigation Pane menu when you have Navigate To Category set to Object Type. On the Create tab, in the Macros & Code group, click the Module command to create a new standard module.

To create a new module, click the Module or Class Module command in the Macros & Code group on the Create tab, also shown in Figure 24-1. When you click Module, Access creates a new *standard module*. You use a standard module to define procedures that you can call from anywhere in your application. It's a good idea to name modules based on their purpose. For example, you might name a module that contains procedures to perform custom calculations for queries *modQueryFunctions*, and you might name a module containing procedures to work directly with Windows functions *modWindowsAPIFunctions*.

Advanced developers might want to create a special type of module object called a *class module*. A class module is a specification for a user-defined object in your application, and the Visual Basic procedures you create in a class module define the properties and methods that your object supports. You create a new class module by clicking Class Module. You'll learn more about objects, methods, properties, and class modules later in this chapter.

Form and Report Modules

To make it easy to create Visual Basic procedures that respond to events on client forms or client reports, Access 2010 supports a class module associated with each form or report. (You can design forms and reports that do not have a related class module.) A module associated with a form or report is also a class module that allows you to respond to events defined for the form or report as well as define extended properties and methods of the form or report. Within a form or report class module, you can create specially named event procedures to respond to Access-defined events, private procedures that you can call only from within the scope of the class module, and public procedures that you can call as methods of the class. See "Collections, Objects, Properties, and Methods," on page 1494, for more information about objects and methods. You can edit the module for a form or a report by opening the form or report in Design view and then clicking the View Code button in the Tools group on the Design contextual tab (located under Form Design Tools). As you'll learn later, you can also open a form or a report by setting an object equal to a new instance of the form or report's class module.

Using form and report modules offers three main advantages over module objects:

- All the code you need to automate a form or a report resides with that form or report. You don't have to remember the name of a separate form-related or report-related module object.

- Access loads module objects into memory when you first reference any procedure or variable in the module and leaves them loaded so long as the database is open. Access loads the code for a form or a report only when the form or the report is opened. Access unloads a form or a report class module when the object is closed; therefore, form and report modules consume memory only when you're using the form or the report to which they are attached.
- If you export a form or report, all the code in the form or report module is exported with it.

However, form and report modules have one disadvantage: Because the code must be loaded each time you open the form or report, a form or report with a large supporting module opens noticeably more slowly than one that has little or no code. In addition, saving a form or report design can take longer if you have also opened the associated module and changed any of the code.

One enhancement that first appeared in Access 97 (version 8.0)—the addition of the HasModule property—helps Access load forms and reports that have no code more rapidly. Access automatically sets this property to Yes if you try to view the code for a form or report, even if you don't define any event procedures. If HasModule is No, Access doesn't bother to look for an associated Visual Basic module, so the form or report loads more quickly.

CAUTION!

If you set the HasModule property to No in the property window, Access deletes the code module associated with the form or report. However, Access warns you and gives you a chance to change your mind if you set the HasModule property to No in error.

The Visual Basic Editor Window

When you open a module in Design view, Access 2010 opens the VBE and asks the editor to display your code. Open the Conrad Systems Contacts (Contacts.accdb) sample database if you haven't already, view the Modules list in the Navigation pane, and then either right-click the modExamples object and click Design View on the shortcut menu or double-click the modExamples object to see the code for this module opened in the VBE, as shown in Figure 24-2. Notice that the VBE in Access 2010 uses the older menu and toolbar technology from releases of Access before Access 2007, not the ribbon used in the main Access window.

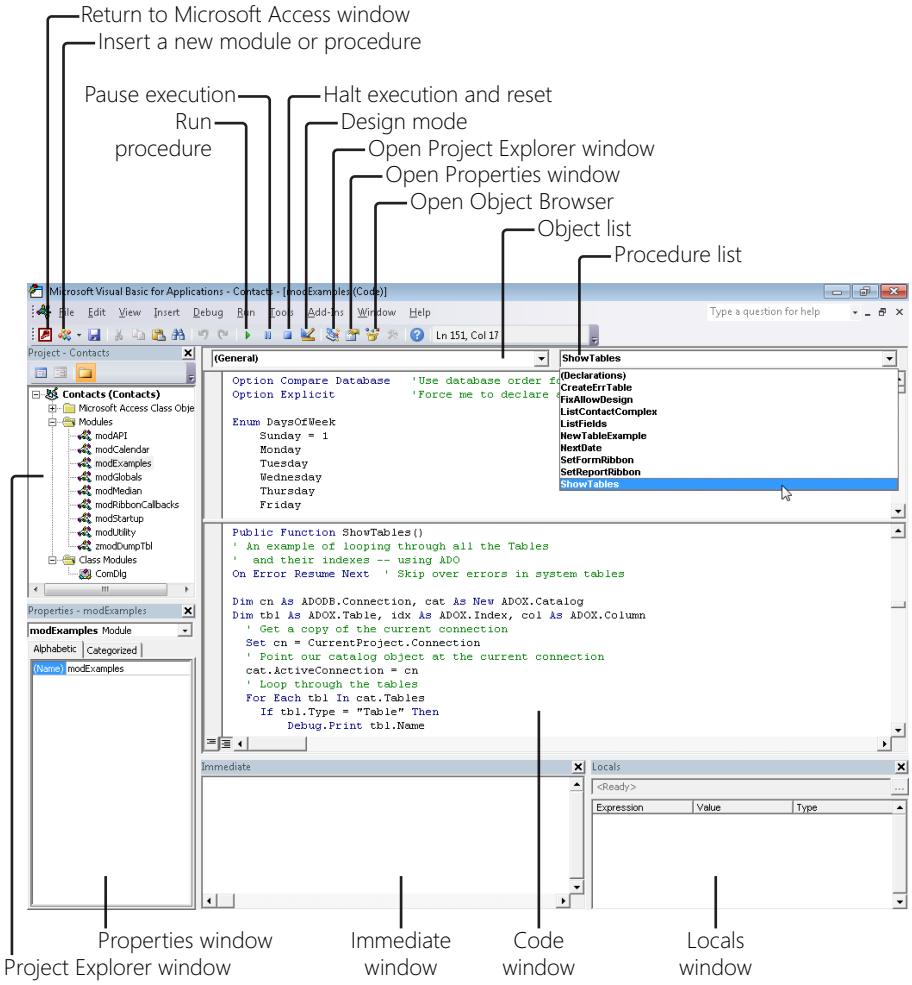

Figure 24-2 Use the VBE to view and edit all Visual Basic code in your database.

What you see on your screen might differ from Figure 24-2, particularly if you have opened the VBE previously and moved some windows around. In the upper-left corner of the figure, you can see the Visual Basic Project Explorer window docked in the workspace. (Click Project Explorer on the View menu or press Ctrl+R to see this window if it's not visible.) In this window, you can discover all module objects and form and report class modules saved

in the database. You can double-click any module to open it in the Code window, which you can see maximized in the upper-right corner.

Docked in the lower-left corner is the Properties window. (Click Properties Window on the View menu or press F4 to see this window if it's not visible.) When you have a form or report that has a Visual Basic module open in Design view in Access, you can click that object in the Project Explorer to see all its properties. If you modify a property in the Properties window, you're changing it in Access. To open a form or report that is not open, you can select it in the Project Explorer and then click Object on the View menu.

In the lower-right corner, you can see the Locals window docked. (Click Locals Window on the View menu to see this window if it's not visible.) As you will see later, this window allows you to instantly see the values of any active variables or objects when you pause execution in a procedure. In the lower center, you can see the Immediate window docked. (Click Immediate Window on the View menu or press Ctrl+G to see this window if it's not visible.) It's called the Immediate window because you can type any valid Visual Basic statement and press Enter to execute the statement immediately. You can also use a special "what is" command character (?) to find out the value of an expression or variable. For example, you can type **?5*20** and press Enter, and Visual Basic responds with the answer on the following line: *100*.

You can undock any window by grabbing its title bar and dragging it away from its docked position on the edge toward the center of the screen. You can also undock a window by right-clicking anywhere in the window and clearing the Dockable property. As you will see later, you can set the Dockable property of any window by clicking Options on the Tools menu. When a window is set as Dockable but not docked along an edge, it becomes a pop-up window that floats on top of other windows—similar to the way an Access form works when its Pop Up property is set to Yes, as you learned in Chapter 14, "Customizing a Form." When you make any window not Dockable, it shares the space occupied by the Code window.

You cannot set the Code window as Dockable. The Code window always appears in the part of the workspace that is not occupied by docked windows. You can maximize the Code window to fill this remaining space, as shown in Figure 24-2. You can also click the Restore button for this window and open multiple overlaid Code windows for different modules within the Code window space.

At the top of the Code window, just below the toolbar, you can see two list boxes:

- **Object list box** When you're editing a form or report class module, open this list on the left to select the form or the report, a section on the form or the report, or any control on the form or the report that can generate an event. The Procedure list box then shows the available event procedures for the selected object. Select General to view the Declarations section of the module, where you can set options or declare variables shared by multiple procedures. In a form or a report class module, General is also where you'll see any procedures you have coded that do not respond to events. When you're editing a standard module object, this list displays only the General option. In a class module object, you can choose General or Class.

- **Procedure list box** Open this list on the right to select a procedure in the module and display that procedure in the Code window. When you're editing a form or report module, this list shows the available event procedures for the selected object and displays in bold type the event procedures that you have coded and attached to the form or the report. When you're editing a module object, the list displays in alphabetic order all the procedures you coded in the module. In a class module when you have selected Class in the Object list box, you can choose the special Initialize or Terminate procedures for the class.

In Figure 24-2, we dragged the divider bar at the top of the scroll bar on the right of the Code window downward to open two edit windows. We clicked in the lower window and then clicked ShowTables in the Procedure list box. You might find a split window very handy when you're tracing calls from one procedure to another. The Procedure list box always shows you the name of the procedure that currently has the focus. In the Code window, you can use the arrow keys to move horizontally and vertically. When you enter a new line of code and press Enter, Visual Basic optionally verifies the syntax of the line and warns you of any problems it finds.

If you want to create a new procedure in a module, you can type either a *Function* statement, a *Sub* statement, or a *Property* statement on any blank line above or below an existing procedure and then press Enter; click anywhere in the module and click the arrow to the right of the Insert button on the toolbar and then click Procedure; or click Procedure on the Insert menu. (For details about the Function and Sub statements, see "Functions and Subroutines," on page 1525. For details about the Property statement, see "Understanding Class Modules," on page 1529.) Visual Basic creates a new procedure for you (it does not embed the new procedure in the procedure you were editing) and inserts an *End Function*, *End Sub*, or

End Property statement. When you create a new procedure using the Insert button or the Insert menu, Visual Basic opens a dialog box where you can enter the name of the new procedure, select the type of the procedure (Sub, Function, or Property), and select the scope of the procedure (Public or Private). To help you organize your procedures, Visual Basic inserts the new procedure in alphabetical order within the existing procedures.

CAUTION!

If you type a Function, Sub, or Property statement in the middle of an existing procedure, Visual Basic accepts the statement if it's syntactically correct, but your project won't compile because you cannot place a Function, Sub, or Property procedure inside another Function, Sub, or Property procedure.

If you're working in a form or report module, you can select an object in the object list box and then open the Procedure list box to see all the available events for that object. An event name displayed in bold type means you have created a procedure to handle that event. Select an event whose name isn't displayed in bold type to create a procedure to handle that event.

Visual Basic provides many options that you can set to customize how you work with modules. Click Options on the Tools menu, and then click the Editor tab to see the settings for these options, as shown in Figure 24-3.

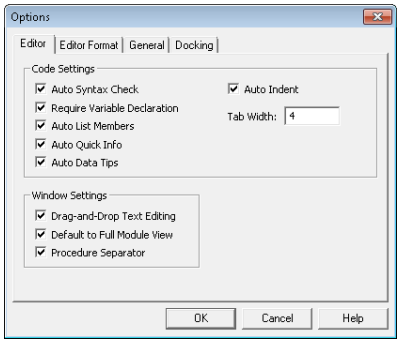

Figure 24-3 You can customize the VBE by using the settings on the Editor tab in the Options dialog box.

On the Editor tab, some important options to consider are Auto Syntax Check, to check the syntax of lines of code as you enter them; and Require Variable Declaration, which forces you to declare all your variables. (Require Variable Declaration is not selected by default—you'll see later why it's important to select it.) If you want to see required and optional parameters as you type complex function calls, select the Auto List Members check box. Auto Quick Info provides drop-down lists where appropriate built-in constants are available to complete parameters in function or subroutine calls. When you're debugging code, Auto Data Tips lets you discover the current value of a variable by pausing your mouse pointer on any usage of the variable in your code.

Drag-And-Drop Text Editing allows you to highlight code and drag it to a new location. Default To Full Module View shows all your code for multiple procedures in a module in a single scrollable view. If you clear that check box, you will see only one procedure at a time and must page up or down or select a different procedure in the Procedure list box to move to a different part of the module. When you're in full module view, selecting the Procedure Separator check box asks Visual Basic to draw a line between procedures to make it easy to see where one procedure ends and another begins.

Selecting the Auto Indent check box asks Visual Basic to leave you at the same indent as the previous line of code when you press the Enter key to insert a new line. We wrote all of the sample code you'll see in this book and in the sample databases with indents to make it easy to see related lines of code within a loop or an If...Then...Else construct. You can set the Tab Width to any value from 1 through 32. This setting tells Visual Basic how many spaces you want to indent when you press the Tab key while writing code.

On the Editor Format tab of the Options dialog box, you can set custom colors for various types of code elements and also choose a display font. We recommend using a monospaced font such as Courier New for all code editing.

On the General tab, shown in Figure 24-4, you can set some important options that dictate how Visual Basic acts as you enter new code and as you debug your code. You can ignore all the settings under Form Grid Settings because they apply to forms designed in Visual Basic, not Access.

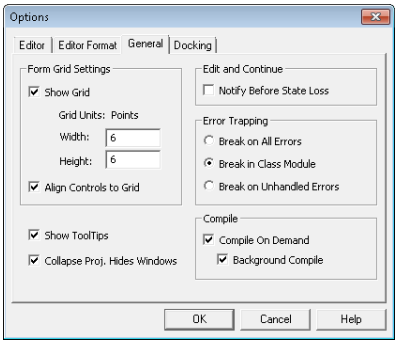

Figure 24-4 You can modify settings to help you debug your code on the General tab in the Options dialog box.

If your code has halted, in many cases you can enter new code or correct problems in code before continuing to test. Some changes you make, however, will force Visual Basic to reset rather than let you continue to run from the halted point. If you select the Notify Before State Loss check box, Visual Basic will warn you before allowing you to make code changes that would cause it to reset.

In the Error Trapping section, you can select one of three ways to tell Visual Basic how to deal with errors. As you'll discover later in this chapter, you can write statements in your code to attempt to catch errors. If you think you have a problem in your error-trapping code, you can select Break On All Errors. With this setting, Visual Basic ignores all error trapping and halts execution on any error. If you have written class modules that can be called from other modules, to catch an untrapped error that occurs within a class module, choose Break In Class Module to halt on the statement within the class module that failed. (We recommend this setting for most testing.) If you choose Break On Unhandled Errors, and an untrapped error occurs within a class module, Visual Basic halts at the statement that invoked the class module.

The last two important options on this tab are Compile On Demand and Background Compile. With the Compile On Demand check box selected, Visual Basic compiles any previously uncompiled new code whenever you run that code directly or run a procedure that calls that code. Background Compile lets Visual Basic use spare CPU cycles to compile new code as you are working in other areas.

Finally, on the Docking tab, you can specify whether the Immediate window, Locals window, Watch window, Project Explorer, Properties window, or Object Browser can be docked. We will take a look at the Immediate window and Watch window in the next section. You can use the Object Browser to discover all the supported properties and methods of any object or function defined in Access, Visual Basic, or your database application.

INSIDE OUT Understanding the Relationship Between Access and Visual Basic

Access 2010 and Visual Basic work as two separate but interlinked products in your Access application. Access handles the storage of the Visual Basic project (both the source code and the compiled code) in your desktop database (.accdb or .mdb) or project (.adp) file, and it calls Visual Basic to manage the editing and execution of your code.

Because Access tightly links your forms and reports with class modules stored in the Visual Basic project, some complex synchronization must happen between the two products. For example, when you open a form module and enter a new event procedure in the Visual Basic Code window, Access must set the appropriate event property to [Event Procedure] so that both the form and the code are correctly linked. Likewise, when you delete all the code in an event procedure, Access must clear the related form or control property. Therefore, when you open a form or report module from the VBE window, you'll notice that Access also opens the related form or report object in the Access window.

When Access first began using Visual Basic (instead of Access Basic) in version 7 (Access for Windows 95), it was possible to end up with a corrupted Visual Basic project or corrupted form or report object if you weren't careful to always compile and save both the code and the form or report definition at the same time when you made changes to either. It was particularly easy to encounter corruption if multiple developers had the database open at the same time. This corruption most often occurred when Access failed to merge a changed module back into the Visual Basic project when the developer saved changes.

Microsoft greatly improved the reliability of this process when it switched in version 9 (Access 2000) to saving the entire Visual Basic project whenever you save a change. However, this change means that two developers can no longer have the same database open and be working in the code at the same time. This also means that your Access file can grow rapidly if you're making frequent changes to the code and saving your changes.

When you're making multiple changes in an Access application, we recommend that you always compile your project when you have finished changing a section of code. (Click Compile on the Debug menu in the VBE.) You should also save all at once multiple objects that you have changed by clicking the Save button in the VBE window and always responding Yes to the Save dialog box message that Access shows you when you have multiple changed objects open.

Working with Visual Basic Debugging Tools

You might have noticed that the debugging tools for data macros and user interface macros are limited. You can't do much more than run user interface macros in Single Step mode to try to find the source of an error. The debugging tools for Visual Basic are significantly more extensive. The following sections describe many of the tools available in Visual Basic. You might want to scan these sections first and then return after you have learned more about the Visual Basic language and have begun writing procedures that you need to debug.

Setting Breakpoints

If you still have the modExamples module open, scroll down until you can see the entire ShowTables function, as shown in Figure 24-5. This sample function examines all the table definitions in the current database and displays the table name, the names of any indexes defined for the table, and the names of columns in each index by printing to a special object called Debug (another name for the Immediate window).

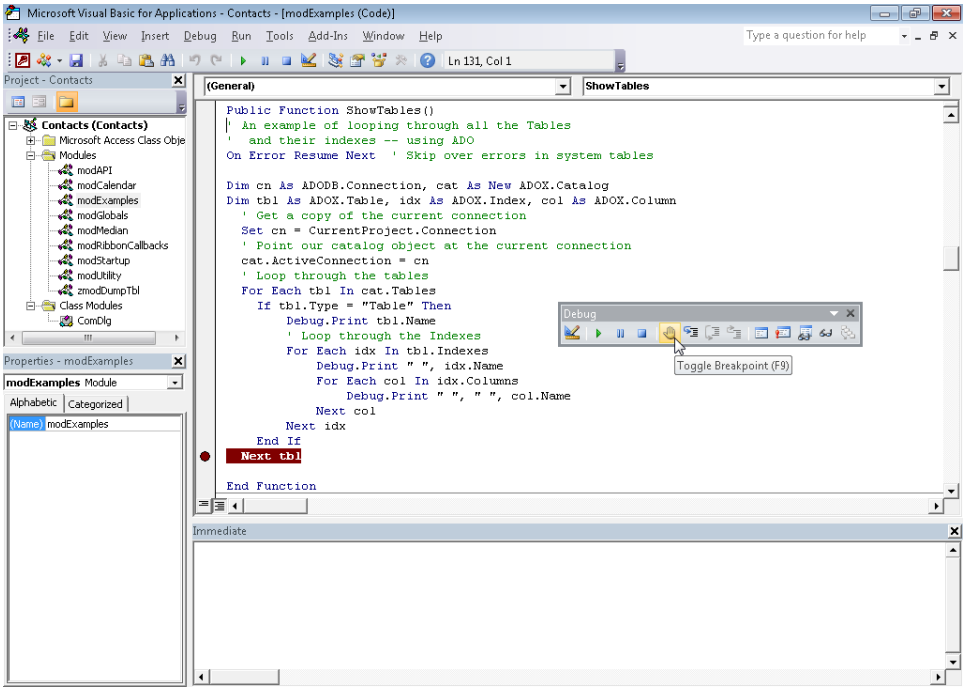

Figure 24-5 You can set a breakpoint in a Visual Basic module to help you debug your code.

One of the most common ways to test particularly complex code is to open the module you want to examine, set a stopping point in the code (called a breakpoint), and then run the code. Visual Basic halts before executing the statement on the line where you set the breakpoint. As you'll soon see, when Visual Basic stops at a breakpoint, you can examine all sorts of information to help you clean up potential problems. While a procedure is stopped, you can look at the values in variables—including all object variables you might have defined. In addition, you can also change the value of variables, single-step through the code, reset the code, or restart at a different statement.

To set a breakpoint, click anywhere on the line of code where you want Visual Basic execution to halt and either click the Toggle Breakpoint button on the Debug toolbar (open this toolbar by right-clicking any toolbar and clicking Debug on the shortcut menu), click Toggle Breakpoint on the Debug menu, or press F9 to set or clear a breakpoint. When a breakpoint is active, Access highlights the line of code (in red by default) where the breakpoint is established and displays a dot on the selection bar to the left of the line of code. Note that you can set as many breakpoints as you like, anywhere in any module. After you set a breakpoint, the breakpoint stays active until you close the current database, specifically clear the breakpoint, or click Clear All Breakpoints on the Debug menu (or press Ctrl+Shift+F9). In the example shown in Figure 24-5, we set a breakpoint to halt the procedure at the bottom of the loop that examines each table. When you run the procedure later, you'll see that Visual Basic will halt on this statement just before it executes the statement. Note that you can right-click the toolbar at the top of the VBE window and then click Debug to see the floating Debug toolbar shown in Figure 24-5.

Using the Immediate Window

"Action central" for all troubleshooting in Visual Basic is a special edit window called the Immediate window. You can open the Immediate window while editing a module by clicking the Immediate Window button on the Debug toolbar or clicking Immediate Window on the View menu. Even when you do not have a Visual Basic module open, you can open the Immediate window from anywhere in Access by pressing Ctrl+G.

Executing Visual Basic Commands in the Immediate Window In the Immediate window (shown in Figure 24-2), you can type any valid Visual Basic command and press Enter to have it executed immediately. You can also execute a procedure by typing the procedure name followed by any parameter values required by the procedure. You can ask Visual Basic to evaluate any expression by typing a question mark character (sometimes called the "what is" character) followed by the expression. Access displays the result of the evaluation on the line below. You might want to experiment by typing **?(5 * 4) / 10**. You will see the answer *2* on the line below.

Because you can type any valid Visual Basic statement, you can enter an *assignment statement* (the name of a variable, an equals sign, and the value you want to assign to the variable) to set a variable that you might have forgotten to set correctly in your code. For example, there's a public variable (you'll learn more about variables later in this chapter) called gintDontShowCompanyList that the Conrad Systems Contacts sample application uses to save whether the current user wants to see the Select Companies pop-up window when clicking Companies on the main switchboard. Some users may prefer to go directly to the Companies/Organizations form that edits all companies rather than select or filter the list. If you have been running the Conrad Systems Contacts application, you can find out the current value of the string by typing

```
?gintDontShowCompanyList
```

Visual Basic displays the value of this variable, which should be either 0 or –1. You can set the value of this string to False (0) by typing

```
gintDontShowCompanyList = 0
```

You can verify the value of the variable you just set by typing

```
?gintDontShowCompanyList
```

If you assigned 0 to the variable, you should see that value echoed in the Immediate window.

To have a sense of the power of what you're doing, go to the Database window in Access by clicking the View Microsoft Access button on the left end of the toolbar in the VBE window. Open the frmMain form in Form view. Click Companies to find out whether the Select Companies form or the Companies/Organizations form opens. If you go directly to the Select Companies form, then gintDontShowCompanyList must be False (0). Close the form that opens.

Now, go back to the VBE window. (An easy way to do this is to use the Windows Alt+Tab feature.) In the Visual Basic Immediate window, set the value to True by entering in the Immediate window

```
gintDontShowCompanyList = True
```

Go back to the main switchboard and click Companies again. Because you set the public variable to True, you should go directly to the Companies/Organizations form. Now that you have the form open to edit companies, you can set a filter directly from the Immediate window. Go back to that window and enter the expression

```
Forms!frmCompanies.Filter = "[StateOrProvince] = 'PA'"
```

If you want, you can ask what the filter property is to see if it is set correctly. Note that nothing has happened yet to the form. Next, turn on the form's FilterOn property by entering

```
Forms!frmCompanies.FilterOn = True
```

Return to the form, and you should now see the form filtered down to two rows—all the companies in the state of Pennsylvania. If you want to try another example, return to the Immediate window and enter

```
Forms!frmCompanies.Section(0).Backcolor = 255
```

The background of Section(0), the detail area of the form, should now appear red. Note that none of these changes affect the design of the form. You can close the form, and the next time you open it, the form will have a normal background color, and the records won't be filtered. Close the forms you have open to continue with the next section.

Using Breakpoints You saw earlier how to set a breakpoint within a module procedure. To see how a breakpoint works, open the modExamples module in the VBE window, find the ShowTables function, and be sure you have set a breakpoint on the `Next tbl` statement, as shown in Figure 24-6.

Because the ShowTables procedure is a function that might return a value, you have to ask Visual Basic to evaluate the function to run it. The function doesn't require any parameters, so you don't need to supply any. To run the function, type **?ShowTables()** in the Immediate window, as shown in Figure 24-6, and press Enter.

> **Note**
>
> **You can also ask Visual Basic to run any public procedure by clicking in the procedure and clicking the Run button on either the Standard or Debug toolbar.**

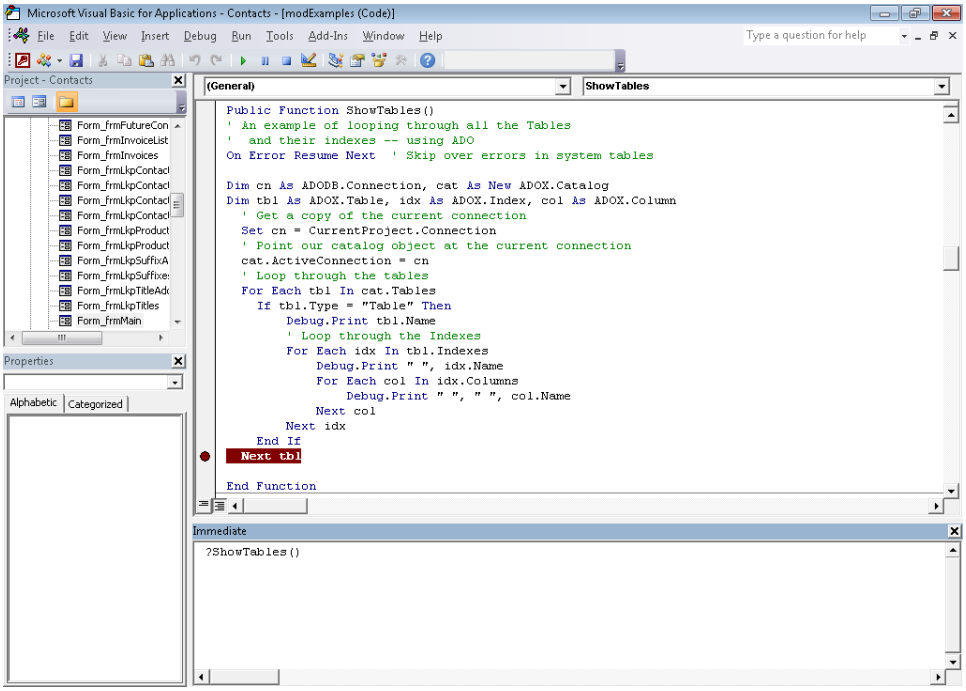

Figure 24-6 You can execute a module function from the Immediate window.

Visual Basic runs the function you requested. Because you set a breakpoint, the code stops on the statement with the breakpoint, as shown in Figure 24-7. The first table in the database is actually a linked table (an Excel spreadsheet), so you won't see any output. Click the Continue button on the toolbar to run through the loop a second time to display the first table.

Note that we clicked Locals Window on the View menu to reveal the Locals window that you can see across the bottom of Figure 24-7. (We undocked the Immediate window so you can see more of the Locals window.) In the Locals window, Visual Basic shows you all the active variables. You can, for example, click the plus sign (+) next to the word cat (a variable set to the currently opened database catalog) to browse through all the property settings for the database and all the objects within the database. You can click on the tbl variable to explore the columns and properties in the table. See "Collections, Objects, Properties, and Methods," on page 1494, for details about all the objects you see in the "tree" under the database catalog.

The Immediate window displays the output of three Debug.Print statements within the function you're running, as also shown in Figure 24-7.

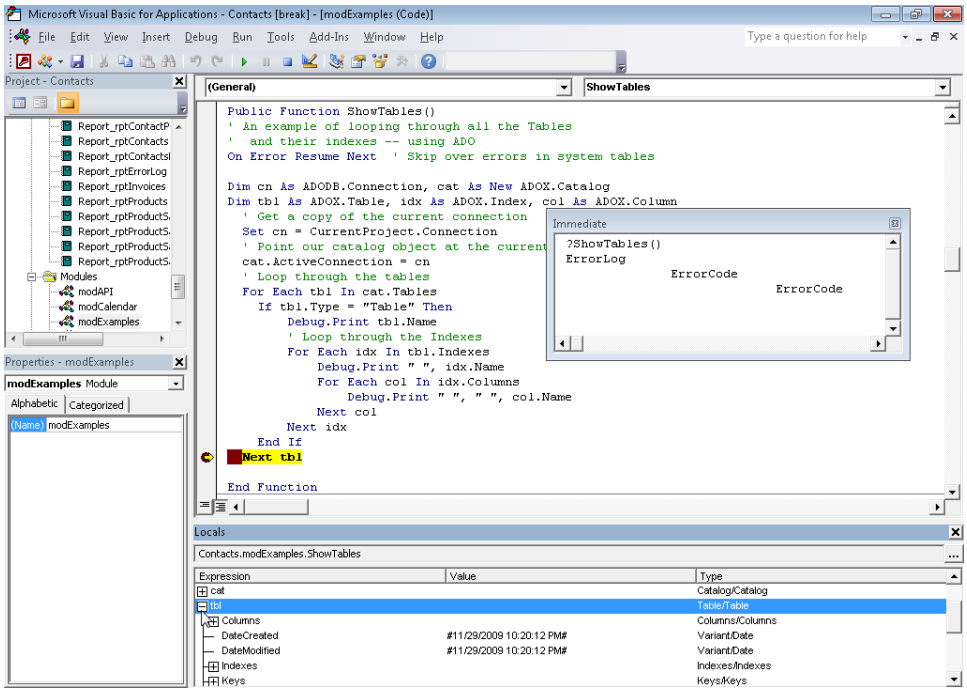

Figure 24-7 When your Visual Basic code stops at a breakpoint, you can use the Locals window to examine variable and object values.

The first line shows the name of the first table (Errorlog) that the function found in the database. The second (indented) line shows the name of the index for that table. The third line shows the name of the one column in the index.

If you want to see the results of executing the next loop in the code (examining the next table object in the catalog), click the Continue button on the toolbar. If you want to run the code a single statement at a time, click Step Into or Step Over on the Debug menu or open the Debug toolbar and click the Step Into or Step Over button. Step Into and Step Over work the same unless you're about to execute a statement that calls another procedure. If the next statement calls another procedure, Step Into literally steps into the called procedure so that you can step through the code in the called procedure one line at a time. Step Over calls the procedure without halting and stops at the next statement in the current procedure.

When you are finished studying the loop in the ShowTables function, be sure to click the Reset button on the toolbar to halt code execution.

Note

The Tables collection in the catalog includes tables, linked tables, system tables, and queries. Because the ShowTables procedure only looks for tables, you will need to loop through the code several times until the procedure finds the next object that defines a table. You should quickly find the ErrorLog, ErrTable, and ErrTableSample tables, but the code must then loop through all the queries and linked tables (more than 40 of them) before finding the SwitchboardDriver table.

Working with the Watch Window

Sometimes setting a breakpoint isn't enough to catch an error. You might have a variable that you know is being changed somewhere by your code (perhaps incorrectly). By using the Watch window, you can examine a variable as your code runs, ask Visual Basic to halt when an expression that uses the variable becomes true, or ask Visual Basic to halt when the variable changes.

An interesting set of variables in the Conrad Systems Contacts sample database are gintDontShowCompanyList, gintDontShowContactList, and gintDontShowInvoiceList (all defined in the modGlobals module). When any of these variables are set to True, the main switchboard bypasses the intermediate list/search form for companies, contacts, and invoices, respectively. You played with one of these variables earlier, but it would be interesting to trap when these are set or reset.

CAUTION!

There are a couple of known issues with setting breakpoints in Access 2010. First, code will not halt if you have cleared the Use Access Special Keys check box in the Application Options section of the Current Database category of the Access Options dialog box (click the File tab on the Backstage view and then click Options). Second, the Break When Value Is True and Break When Value Changes options in the Add Watch dialog box will not work if the value or expression you're watching is changed in a form or report module that is not already open in the VBE. For this example to work, the form modules for frmMain, frmSignon, and frmUsers must be open. You can verify that these modules are open by opening the Windows menu in the VBE window. The Contacts.accdb sample file should have modules open, but these modules might not be open in your copy if you have closed them and compiled and saved the project. You can find these modules in the Project Explorer window. Open the list of objects in the Microsoft Class Objects category and then double-click the form modules that you need to open them.

To set a watch for when the value changes, open the Watch window by clicking it on the View menu, right-click in the Watch window, and click Add Watch on the shortcut menu. You can also click Add Watch on the Debug menu. You should see the Add Watch dialog box, as shown in Figure 24-8.

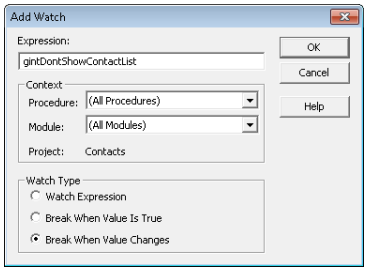

Figure 24-8 You can set a watch for when a variable's value changes.

In the Expression box, enter the name of the variable you want the code to watch. In this case, you want to watch when the gintDontShowContactList variable changes. You don't know where the variable is set, so set the Procedure and Module selections to (All Procedures) and (All Modules), respectively. Under Watch Type, select the Break When Value Changes option, and click OK to set the watch. Go to the Immediate window and set gintDontShowContactList to True by entering **gintDontShowContactList = True** and pressing Enter. Now return to the Navigation pane and start the application by opening the frmSplash form. (Code in the Load event of this form hides the Navigation pane and then opens the Conrad Systems Contacts Sign On form.) Because you set a watch to halt when gintDontShowContactList changes, the code execution should halt in the module for the frmSignOn form, as shown in Figure 24-9.

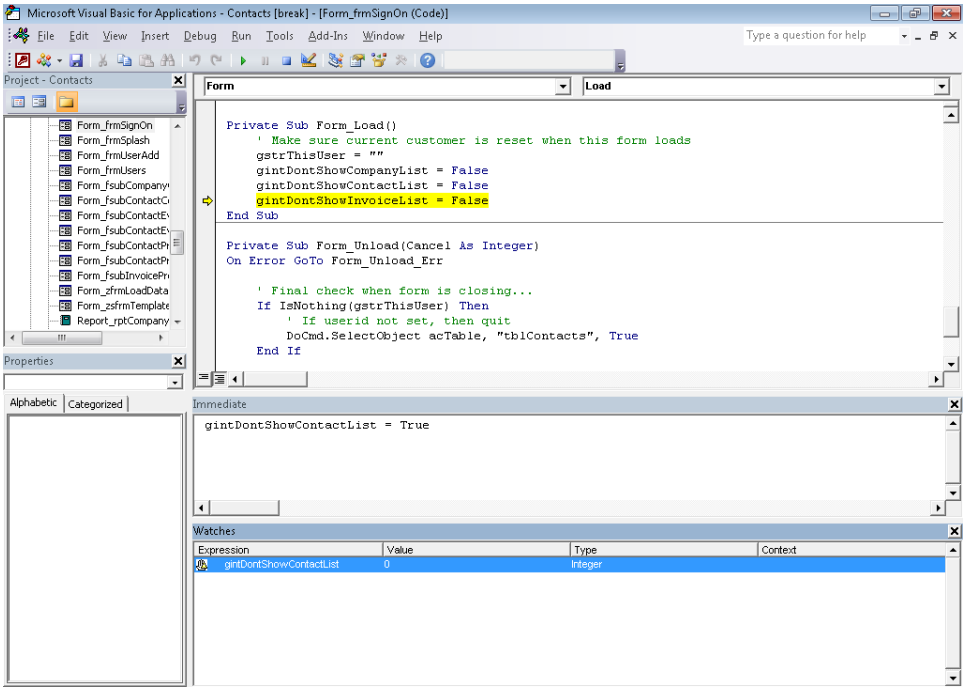

Figure 24-9 Visual Basic code halts immediately after a watch variable has changed.

Note that the code halts at the statement immediately after the one that reset the watched variable. If you didn't set the variable to True before you started the application, Visual Basic won't halt because the value won't be changing.

Click Continue (or press F5) to let the code continue executing. Return to the Access window, and in the Conrad Systems Contacts Sign On dialog box, select my name (Jeff Conrad), type **password** in the Password text box, and press Enter or click Sign On. The dialog box will close, and the main switchboard form opens. In the main switchboard, click Users to open the user edit form. The first record should be my record unless you've created other users. Select the Don't Show Contact List check box in my record and click Save. The procedure halts again, as shown in Figure 24-10.

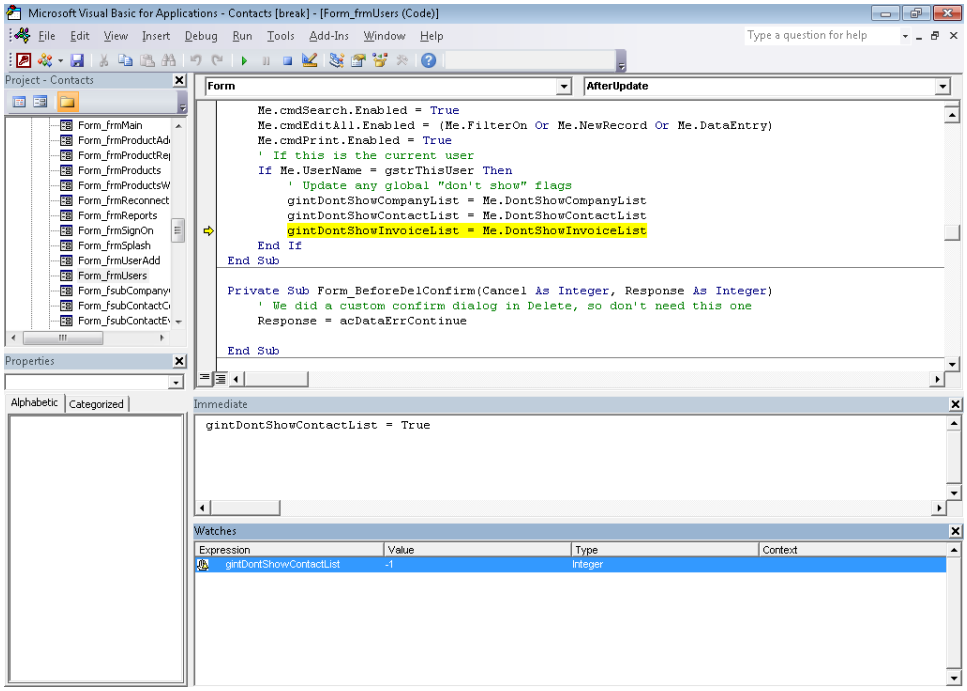

Figure 24-10 The gintDontShowContactList variable is set to the value of a form control.

It appears that this code is setting the gintDontShowContactList variable to some value on the user edit form. (As you'll learn later, Me is a shorthand way to reference the form object where your code is running, so Me.DontShowContactList references a control on the form.) Click Continue again to let the code finish execution. Return to the Access window and click the Close button on the Users form to return to the main switchboard. (The Users form might be showing behind the main switchboard form.)

If you open frmUsers in Design view (you can't do this while the procedure is still halted) and examine the names of the check box controls on the form, you'll find that the check box you selected is named DontShowContactList. When the code behind frmUsers detects a change to the options for the currently signed-on user, it makes sure the option variables in modGlobals get changed as well. Be sure to close the frmUsers form when you're finished looking at it.

Examining the Procedure Call Sequence (Call Stack)

After stopping code that you're trying to debug, it's useful sometimes to find out what started the current sequence of code execution and what procedures have been called by Visual Basic. For this example, you can continue with the watch on the gintDontShowContactList variable.

You should now be at the main switchboard form (frmMain) in the application. Click Exit to close the application and return to the Navigation pane. (You'll see a prompt asking you if you're sure you want to exit—click Yes. You might also see a prompt offering to back up the data file—click No.) The code should halt again at the Close event of the frmMain form. Click the Call Stack button on the toolbar or click Call Stack on the View menu to see the call sequence shown in Figure 24-11.

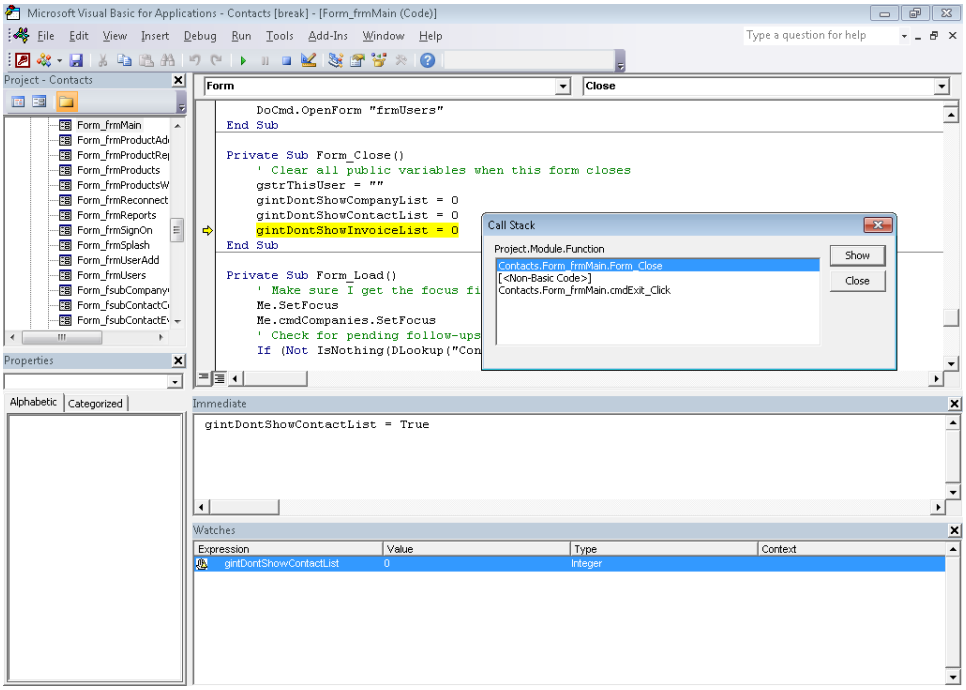

Figure 24-11 When your code is halted, you can see the chain of code executed to the point of the halt in the Call Stack dialog box.

The Call Stack dialog box shows the procedures that have executed, with the most recent procedure at the top of the list, and the first procedure at the bottom. You can see that the code started executing in the cmdExit_Click procedure of the frmMain form. This happens to be the Visual Basic event procedure that runs when you click Exit. If you click that line and then click Show, you should see the cmdExit_Click procedure in the module for the frmMain form (the switchboard) with the cursor on the line that executes the DoCmd.Close command to close the form. This line calls the Access built-in Close command (the <Non-Basic Code> you see in the call stack list), which in turn triggered the Close event procedure for the form. It's the Close event procedure code that sets the gintDontShowContactList variable back to False (0). Be sure that the Call Stack dialog box is closed and click Continue on the toolbar to let the code finish running.

> **Note**
>
> **Be sure to delete the watch after you are finished seeing how it works by right-clicking it in the Watch window and clicking Delete on the shortcut menu.**

Variables and Constants

In addition to using Visual Basic code to work with the controls on any open forms or reports (as you can with user interface macros), you can declare and use named variables in Visual Basic code for storing values temporarily, calculating a result, or manipulating any of the objects in your database. To create a value available anywhere in your code, you can define a global variable, as you can find in the modGlobals module in the Conrad Systems Contacts sample database.

Another way to store data in Visual Basic is with a *constant*. A constant is a data object with a fixed value that you cannot change while your application is running. You've already encountered some of the built-in constants in Access 2010—Null, True, and False. Visual Basic also has a large number of intrinsic constants—built-in constants that have meaningful names—that you can use to test for data types and other attributes or that you can use as fixed arguments in functions and expressions. You can view the list of intrinsic constants by searching for the Visual Basic Constants topic in Help. You can also declare your own constant values to use in code that you write.

In the following sections, you'll learn about using variables to store and calculate data and to work with database objects.

Data Types

Visual Basic supports data types for variables and constants that are similar to the data types you use to define fields in tables. It also allows you to define a variable that is a pointer to an object (such as a form or a recordset). The data types are described in Table 24-1.

Table 24-1 Visual Basic Data Types

Data Type	Size	Data-Typing Character	Can Contain
Boolean	2 bytes	(none)	True (–1) or False (0)
Byte	1 byte	(none)	Binary data ranging in value from 0 through 255
Integer	2 bytes	%	Integers from –32,768 through 32,767
Long	4 bytes	&	Integers from –2,147,483,648 through 2,147,483,647
LongLong	8 bytes	^	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (Valid on 64-bit platforms only.)
LongPtr	4 bytes on 32-bit systems and 8 bytes on 64-bit systems	(none)	–2,147,483,648 to 2,147,483,647 on 32-bit systems –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 on 64-bit systems
Single	4 bytes	!	Floating-point (imprecise) numbers from approximately -3.4×10^{38} through 3.4×10^{38}
Double	8 bytes	#	Floating-point (imprecise) numbers from approximately -1.79×10^{308} through 1.79×10^{308}
Currency	8 bytes	@	A scaled integer with four decimal places from –922,337,203,685,477.5808 through 922,337,203,685,477.5807
Decimal	14 bytes	(none)	A precise number with up to 29 digits and up to 28 decimal places from -79.228×10^{27} to 79.228×10^{27} (Visual Basic in Access supports the Decimal data type only as a type within the Variant data type.)
String	10 bytes plus 2 bytes per character	$	Any text or binary string up to approximately 2 billion bytes in length, including text, hyperlinks, memo data, and "chunks" from an ActiveX object; a fixed-length string can be up to 65,400 characters long

Data Type	Size	Data-Typing Character	Can Contain
Date	8 bytes	(none)	Date/time values ranging from January 1, 100, to December 31, 9999
Object	4 bytes	(none)	A pointer to an object—you can also define a variable that contains a specific type of object, such as the Database object
Variant	16 bytes through approximately 2 billion bytes	(none)	Any data, including Empty, Null, and date/time data (Use the VarType function to determine the current data type of the data in the variable. A Variant can also contain an array of Variants. Use the andto determine whether a Variant is an array.)
User-defined	Depends on elements defined	(none)	Any number of variables of any of the above data types

You can implicitly define the data type of a variable by appending a data-typing character, as noted in Table 24-1, the first time you use the variable. For example, a variable named *MyInt%* is an integer variable. If you do not explicitly declare a data variable that you reference in your code and do not supply a data-typing character, Visual Basic assigns the Variant data type to the variable. (See "Declaring Constants and Variables," on page 1479, to learn how to explicitly declare data variables.) Note that although the Variant data type is the most flexible (and, in fact, is the data type for all controls on forms and reports), it is also the least efficient because Visual Basic must do extra work to determine the current data type of the data in the variable before working with it in your code. Variant is also the only data type that can contain the Null value.

The Object data type lets you define variables that can contain a pointer to an object. See "Collections, Objects, Properties, and Methods," on page 1494, for details about objects that you can work with in Visual Basic. You can declare a variable as the generic Object data type, or you can specify that a variable contains a specific type of object. The major object types are AccessObject, Application, Catalog, Column, Command, Connection, Container, Control, Database, Document, Error, Field, Form, Group, Index, Key, Parameter, Procedure, Property, QueryDef, Recordset, Relation, Report, Table, TableDef, User, View, and Workspace.

INSIDE OUT

Using Option Explicit Is a Good Idea

You can request that Visual Basic generate all new modules with an Option Explicit statement by selecting the Require Variable Declaration check box on the Editor tab of the Options dialog box, as shown in Figure 24-3. If you set this option, Visual Basic includes an Option Explicit statement in the Declarations section of every new module. This helps you avoid errors that can occur when you use a variable in your code that you haven't properly declared in a Dim, Public, Static, or Type statement or as part of the parameter list in a Function statement or a Sub statement. (See "Functions and Subroutines," on page 1525.) When you specify this option in a module, Visual Basic flags any undeclared variables it finds when you ask it to compile your code. Using an Option Explicit statement helps you find variables that you might have misspelled when you entered your code.

Variable and Constant Scope

The scope of a variable or a constant determines whether the variable or the constant is known to only one procedure, all procedures in a module, or all procedures in your database. You can create variables or constants that can be used by any procedure in your database (public scope). You can also create variables or constants that apply only to the procedures in a module or only to a single procedure (private scope). A variable declared inside a procedure is always private to that procedure (available only within the procedure). A variable declared in the Declarations section of a module can be private (available only to the procedures in the module) or public. You can pass values from one procedure to another using a parameter list, but the values might be held in variables having different names in the two procedures. See the sections on the Function, Sub, and Call statements later in this chapter.

To declare a public *variable*, use the *Public* statement in the Declarations section of a standard module or a class module. All modules attached to forms or reports are class modules. To declare a public *constant*, use the Public keyword with a *Const* statement in the Declarations section of a standard module. You cannot declare a public constant in a class module. To declare a variable or a constant that all procedures in a module can reference, define that variable or constant in the Declarations section of the module. (A variable defined in a Declarations section is private to the module unless you use the Public statement.) To declare a variable or a constant used only in a particular procedure, define that variable or constant as part of the procedure.

Visual Basic in Access 2010 allows you to use the same name for variables or constants in different module objects or at different levels of scope. In addition, you can declare constants and public variables in form and report modules as well as public variables and constants in standard modules. (You can declare a constant in a form or report module, but it cannot be public.)

To use the same name for public variables and constants in different module objects or form or report modules, specify the name of the module to which it belongs when you refer to it. For example, you can declare a public variable named intX in a module object with the name modMyModule and then declare another public variable named intX in a second module object, named modMyOtherModule. If you want to reference the intX variable in modMyModule from a procedure in modMyOtherModule (or any module other than modMyModule), you must use the following code:

```
modMyModule.intX
```

You can also declare variables or constants with the same name at different levels of scope within a module object or a form or report module. For example, you can declare a public variable named intX and then declare a local variable named intX within a procedure. (You can't declare a public variable within a procedure.) References to intX within the procedure refer to the local variable, while references to intX outside the procedure refer to the public variable. To refer to the public variable from within the procedure, qualify it with the name of the module, just as you would refer to a public variable from within a different module.

Declaring a public variable in a form or report module can be useful for variables that are logically associated with a particular form or report but that you might also want to use elsewhere. Like the looser naming restrictions, however, this feature can sometimes create confusion. In general, it's still a good idea to keep common public variables and constants in standard modules and to give public variables and constants names that are unique across all variable names in your application.

Note

For information on the syntax conventions used in the remainder of this chapter, refer to "Syntax Conventions," in the "Conventions Used in This Book" section at the beginning of this book.

Declaring Constants and Variables

The following sections show the syntax of the statements you can use to define constants and variables in your modules and procedures.

Const Statement

Use a Const statement to define a constant.

Syntax

```
[Public | Private] Const {constantname [As datatype]
  = <const expression>},...
```

Notes

Include the Public keyword in the Declarations section of a standard module to define a constant that is available to all procedures in all modules in your database. Include the Private keyword to declare constants that are available only within the module where the declaration is made. Constants are private by default, and a constant defined within a procedure is always private. You cannot define a Public constant in a class module. (All constants in a class module are private.)

Valid *datatype* entries can be Byte, Boolean, Integer, Long, Currency, Single, Double, Date, String, or Variant. You cannot declare a constant as an object. Use a separate As *datatype* clause for each constant being declared. If you don't declare a type, Visual Basic assigns the data type that is most appropriate for the expression provided. (You should always explicitly declare the data type of your constants.)

The <*const expression*> item cannot include variables, user-defined functions, or Visual Basic built-in functions (such as Chr). You can include simple literals and other previously defined constants.

Example

To define the constant PI to be available to all procedures in all modules, enter the following in the Declarations section of any standard module:

```
Public Const PI As Double = 3.14159
```

INSIDE OUT

Use Variable Naming Conventions

It's a good idea to give all variable names you create a prefix notation that indicates the data type of the variable, particularly if you create complex procedures. This helps ensure that you aren't attempting to assign or calculate incompatible data types. (For example, the names will make it obvious that you're creating a potential error if you try to assign the contents of a long integer variable to an integer variable.) It also helps ensure that you pass variables of the correct data type to procedures. Finally, including a prefix helps ensure that you do not create a variable name that is the same as an Access or Visual Basic reserved word. The following table suggests data type prefixes that you can use for many of the most common data types.

Data Type	Prefix	Data Type	Prefix
Boolean	bol	Document	doc
Byte	byt	Field	fld
Currency	cur	Form	frm
Double	dbl	Index	idx
Integer	int	Key	key
Long	lng	Parameter	prm
Single	sgl	Procedure	prc
String	str	Property	prp
User-defined (using the Type statement)	usr	QueryDef	qdf
Variant	var	Recordset	rst
Catalog	cat	Report	rpt
Column	col	Table	tbl
Command	cmd	TableDef	tbl
Connection	cn	View	vew
Control	ctl	Workspace	wks
Database	db		

Dim Statement

Use a Dim statement in the Declarations section of a module to declare a variable or a variable array that can be used in all procedures in the module. Use a Dim statement within a procedure to declare a variable used only in that procedure.

Syntax

```
Dim {[WithEvents] variablename
  [([<array dimension>],... )] [As [New]
  datatype]},...
```

where <*array dimension*> is

```
[lowerbound To ] upperbound
```

Notes

If you do not include an <*array dimension*> specification but you do include the parentheses, you must include a ReDim statement in eac procedure that uses the array to dynamically allocate the array at run time. You can define an array with as many as 60 dimensions. If you do not include a *lowerbound* value in an <*array dimension*> specification, the default lower bound is 0. You can reset the default lower bound to 1 by including an *Option Base 1* statement in the module Declarations section. The *lowerbound* and *upperbound* values must be integers, and *upperbound* must be greater than or equal to *lowerbound*. The number of members of an array is limited only by the amount of memory on your computer.

Valid *datatype* entries are Byte, Boolean, Integer, Long, Currency, Single, Double, Date, String (for variable-length strings), String * *length* (for fixed-length strings), Object, Variant, or one of the object types described earlier in this chapter. You can also declare a user-defined variable structure using the Type statement and then use the user type name as a data type. You should always explicitly declare the data type of your variables. If you do not include the As *datatype* clause, Visual Basic assigns the Variant data type.

Use the New keyword to indicate that a declared object variable is a new instance of an object that doesn't have to be set before you use it. You can use the New keyword only with object variables to create a new instance of that class of object without requiring a Set statement. You can't use New to declare dependent objects. If you do not use the New keyword, you cannot reference the object or any of its properties or methods until you set the variable to an object using a Set statement.

Use the WithEvents keyword to indicate an object variable within a class module that responds to events triggered by an ActiveX object. Form and report modules that respond to events on the related form and report objects are class modules. You can also define custom class modules to create custom objects. If you use the WithEvents keyword, you cannot use the New keyword.

Visual Basic initializes declared variables at compile time. Numeric variables are initialized to zero (0), variant variables are initialized to empty, variable-length string variables are initialized as zero-length strings, and fixed-length string variables are filled with American National Standards Institute (ANSI) zeros (Chr(0)). If you use a Dim statement within a procedure to declare variables, Visual Basic reinitializes the variables each time you run the procedure.

Examples

To declare a variable named intMyInteger as an integer, enter the following:

```
Dim intMyInteger As Integer
```

To declare a variable named dbMyDatabase as a database object, enter the following:

```
Dim dbMyDatabase As Database
```

To declare an array named strMyString that contains fixed-length strings that are 20 characters long and contains 50 entries from 51 through 100, enter the following:

```
Dim strMyString(51 To 100) As String * 20
```

To declare a database variable, a new table variable, and two new field variables for the table; set up the objects; and append the new table to the Tabledefs collection, enter the following:

```
Public Sub NewTableExample()
    Dim db As DAO.Database
    Dim tdf As New DAO.TableDef, _
        fld1 As New DAO.Field, _
        fld2 As New DAO.Field
    ' Initialize the table name
    tdf.Name = "MyTable"
    ' Set the name of the first field
    fld1.Name = "MyField1"
    ' Set its data type
    fld1.Type = dbLong
    ' Append the first field to the Fields
    ' collection of the table
    tdf.Fields.Append fld1
    ' Set up the second field
    fld2.Name = "MyField2"
    fld2.Type = dbText
    fld2.Size = 20
    ' Append the second field to the table
```

```
    tdf.Fields.Append fld2
    ' Establish an object on the current database
    Set db = CurrentDb
    ' Create a new table by appending tdf to
    ' the Tabledefs collection of the database
    db.TableDefs.Append tdf
End Sub
```

See "Collections, Objects, Properties, and Methods," on page 1494, for details about working with DAO objects. See "Functions and Subroutines," on page 1525, for details about the Sub statement.

To declare an object variable to respond to events in another class module, enter the following:

```
Option Explicit
Dim WithEvents objOtherClass As MyClass

Sub LoadClass ()
    Set objOtherClass = New MyClass
End Sub

Sub objOtherClass_Signal(ByVal strMsg As string)
    MsgBox "MyClass Signal event sent this " & _
      "message: " & strMsg
End Sub
```

In class module MyClass, enter the following:

```
Option Explicit
Public Event Signal(ByVal strMsg As String)

Public Sub RaiseSignal(ByVal strText As String)
    RaiseEvent Signal(strText)
End Sub
```

In any other module, execute the following statement:

```
    MyClass.RaiseSignal "Hello"
```

Enum Statement

Use an Enum statement in a module Declarations section to assign long integer values to named members of an enumeration. You can use an enumeration name as a restricted Long data type.

Syntax

```
[Public | Private] Enum enumerationname
    <member> [= <long integer expression>]
    ...
End Enum
```

Notes

Enumerations are constant values that you cannot change when your code is running. Include the Public keyword to define an enumeration that is available to all procedures in all modules in your database. Include the Private keyword to declare an enumeration that is available only within the module where the declaration is made. Enumerations are public by default.

You must declare at least one member within an enumeration. If you do not provide a *<long integer expression>* assignment, Visual Basic adds 1 to the previous value or assigns 0 if the member is the first member of the enumeration. The *<long integer expression>* cannot include variables, user-defined functions, or Visual Basic built-in functions (such as CLng). You can include simple literals and other previously defined constants or enumerations.

Enumerations are most useful as a replacement for the Long data type in a Function or Sub statement. When you call the function or sub procedure in code, you can use one of the enumeration names in place of a variable, constant, or literal. If you select the Auto List Members option (see Figure 24-3), Visual Basic displays the available names in a drop-down list as you type the sub or function call in your code.

Example

To declare a public enumeration for days of the week and use the enumeration in a procedure, enter the following:

```
Option Explicit
Public Enum DaysOfWeek
    Sunday = 1
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
End Enum
```

```
Public Function NextDate(lngDay As DaysOfWeek) As Date
' This function returns the next date
' that matches the day of week requested
Dim intThisDay As Integer, datDate As Date
    ' Get today
    datDate = Date
    ' Figure out today's day of week
    intThisDay = WeekDay(datDate)
    ' Calculate next day depending on
    ' whether date requested is higher or lower
    If intThisDay < lngDay Then
        NextDate = datDate + (lngDay - intThisDay)
    Else
        NextDate = datDate + (lngDay + 7) - intThisDay
    End If
End Function
```

You can test the function from the Immediate window by entering the following:

```
?NextDate(Monday)
```

Event Statement

Use the Event statement in the Declarations section of a class module to declare an event that can be raised within the module. In another module, you can define an object variable using the WithEvents keyword, set the variable to an instance of this class module, and then code procedures that respond to the events declared and triggered within this class module.

Syntax

[Public] Event *eventname* ([*<arguments>*])

where *<arguments>* is

{[**ByVal** | **ByRef**] *argumentname* [**As** *datatype*]},...

Notes

An Event must be public, which makes the event available to all other procedures in all modules. If you want, you can include the Public keyword when coding this statement.

You should declare the data type of any arguments in the event's argument list. Note that the names of the variables passed by the triggering procedure can be different from the names of the variables known by this event. If you use the ByVal keyword to declare an

argument, Visual Basic passes a copy of the argument to your event. Any change you make to a ByVal argument does not change the original variable in the triggering procedure. If you use the ByRef keyword, Visual Basic passes the actual memory address of the variable, allowing the event to change the variable's value in the triggering procedure. (If the argument passed by the triggering procedure is an expression, Visual Basic treats it as if you had declared it by using ByVal.) Visual Basic always passes arrays by reference (ByRef).

Example

To declare an event that can be triggered from other modules, enter the following in the class module MyClass:

```
Option Explicit
Public Event Signal(ByVal strMsg As String)

Public Sub RaiseSignal(ByVal strText As String)
    RaiseEvent Signal(strText)
End Sub
```

To respond to the event from another module, enter the following:

```
Option Explicit
Dim WithEvents objOtherClass As MyClass

Sub LoadClass ()
    Set objOtherClass = New MyClass
End Sub

Sub objOtherClass_Signal(ByVal strMsg As string)
    MsgBox "MyClass Signal event sent this " & _
      "message: " & strMsg
End Sub
```

To trigger the event in any other module, execute the following:

```
    MyClass.RaiseSignal "Hello"
```

Private Statement

Use a Private statement in the Declarations section of a standard module or a class module to declare variables that you can use in any procedure within the module. Procedures in other modules cannot reference these variables.

Syntax

```
Private {[WithEvents] variablename
  [([<array dimension>],... )]
  [As [New] datatype]},...
```

where <*array dimension*> is

```
[lowerbound To ] upperbound
```

Notes

If you do not include an <*array dimension*> specification but you do include the parentheses, you must include a ReDim statement in each procedure that uses the array to dynamically allocate the array at run time. You can define an array with up to 60 dimensions. If you do not include a *lowerbound* value in an <*array dimension*> specification, the default lower bound is 0. You can reset the default lower bound to 1 by including an *Option Base 1* statement in the module Declarations section. The *lowerbound* and *upperbound* values must be integers, and *upperbound* must be greater than or equal to *lowerbound*. The number of members of an array is limited only by the amount of memory on your computer.

Valid *datatype* entries are Byte, Boolean, Integer, Long, Currency, Single, Double, Date, String (for variable-length strings), String * *length* (for fixed-length strings), Object, Variant, or one of the object types described earlier in this chapter. You can also declare a user-defined variable structure using the Type statement and then use the user type name as a data type. You should always explicitly declare the data type of your variables. If you do not include the As *datatype* clause, Visual Basic assigns the Variant data type.

Use the New keyword to indicate that a declared object variable is a new instance of an object that doesn't have to be set before you use it. You can use the New keyword only with object variables to create a new instance of that class of object without requiring a Set statement. You can't use New to declare dependent objects. If you do not use the New keyword, you cannot reference the object or any of its properties or methods until you set the variable to an object using a Set statement.

Use the WithEvents keyword to indicate an object variable within a class module that responds to events triggered by an ActiveX object. Form and report modules that respond to events on the related form and report objects are class modules. You can also define custom class modules to create custom objects. If you use the WithEvents keyword, you cannot use the New keyword.

Visual Basic initializes declared variables at compile time. Numeric variables are initialized to zero (0), variant variables are initialized to empty, variable-length string variables are initialized as zero-length strings, and fixed-length string variables are filled with ANSI zeros (Chr(0)).

Example

To declare a long variable named lngMyNumber that can be used in any procedure within this module, enter the following:

```
Private lngMyNumber As Long
```

Public Statement

Use a Public statement in the Declarations section of a standard module or a class module to declare variables that you can use in any procedure anywhere in your database.

Syntax

```
Public {[WithEvents] variablename
  [([<array dimension>],... )]
  [As [New] datatype]},...
```

where <*array dimension*> is

```
[lowerbound To ] upperbound
```

Notes

If you do not include an <*array dimension*> specification but you do include the parentheses, you must include a ReDim statement in each procedure that uses the array to dynamically allocate the array at run time. You can define an array with up to 60 dimensions. If you do not include a *lowerbound* value in an <*array dimension*> specification, the default lower bound is 0. You can reset the default lower bound to 1 by including an *Option Base 1* statement in the module Declarations section. The *lowerbound* and *upperbound* values must be integers, and *upperbound* must be greater than or equal to *lowerbound*. The number of members of an array is limited only by the amount of memory on your computer.

Valid *datatype* entries are Byte, Boolean, Integer, Long, Currency, Single, Double, Date, String (for variable-length strings), String * *length* (for fixed-length strings), Object, Variant, or one of the object types described earlier in this chapter. Note, however, that you cannot declare a Public fixed-length string within a class module. You can also declare a user-defined variable structure using the Type statement and then use the user type name as a data type. You should always explicitly declare the data type of your variables. If you do not include the As *datatype* clause, Visual Basic assigns the Variant data type.

Use the New keyword to indicate that a declared object variable is a new instance of an object that doesn't have to be set before you use it. You can use the New keyword only with object variables to create a new instance of that class of object without requiring a Set statement. You can't use New to declare dependent objects. If you do not use the New keyword, you cannot reference the object or any of its properties or methods until you set the variable to an object using a Set statement.

Use the WithEvents keyword to indicate an object variable within a class module that responds to events triggered by an ActiveX object. Form and report modules that respond to events on the related form and report objects are class modules. You can also define custom class modules to create custom objects. If you use the WithEvents keyword, you cannot use the New keyword.

Visual Basic initializes declared variables at compile time. Numeric variables are initialized to zero (0), variant variables are initialized to empty, variable-length string variables are initialized as zero-length strings, and fixed-length string variables are filled with ANSI zeros (Chr(0)).

Example

To declare a long variable named lngMyNumber that can be used in any procedure in the database, enter the following:

```
Public lngMyNumber As Long
```

ReDim Statement

Use a ReDim statement to dynamically declare an array within a procedure or to redimension a declared array within a procedure at run time.

Syntax

```
ReDim [Preserve] {variablename
  (<array dimension>,...) [As datatype]},...
```

where *<array dimension>* is

```
[lowerbound To ] upperbound
```

Notes

If you're dynamically allocating an array that you previously defined with no *<array dimension>* specification in a Dim, Public, or Private statement, your array can have up to 60 dimensions. You cannot dynamically reallocate an array that you previously defined with

an <*array dimension*> specification in a Dim, Public, or Private statement. If you declare the array only within a procedure, your array can have up to 60 dimensions. If you do not include a *lowerbound* value in an <*array dimension*> specification, the default lower bound is 0. You can reset the default lower bound to 1 by including an *Option Base 1* statement in the module Declarations section. The *lowerbound* and *upperbound* values must be integers, and *upperbound* must be greater than or equal to *lowerbound*. The number of members of an array is limited only by the amount of memory on your computer. If you previously specified dimensions in a Public, Private, or Dim statement or in another ReDim statement within the same procedure, you cannot change the number of dimensions.

Include the Preserve keyword to ask Visual Basic not to reinitialize existing values in the array. When you use Preserve, you can change the bounds of only the last dimension in the array.

Valid *datatype* entries are Byte, Boolean, Integer, Long, Currency, Single, Double, Date, String (for variable-length strings), String * *length* (for fixed-length strings), Object, Variant, or one of the object types described earlier in this chapter. You can also declare a user-defined variable structure using the Type statement and then use the user type name as a data type. You should always explicitly declare the data type of your variables. If you do not include the As *datatype* clause, Visual Basic assigns the Variant data type. You cannot change the data type of an array that you previously declared with a Dim, Public, or Private statement. After you establish the number of dimensions for an array that has module or global scope, you cannot change the number of its dimensions using a ReDim statement.

Visual Basic initializes declared variables at compile time. Numeric variables are initialized to zero (0), variant variables are initialized to empty, variable-length string variables are initialized as zero-length strings, and fixed-length string variables are filled with ANSI zeros (Chr(0)). When you use the Preserve keyword, Visual Basic initializes only additional variables in the array. If you use a ReDim statement within a procedure to both declare and allocate an array (and you have not previously defined the array with a Dim, Public, or Private statement), Visual Basic reinitializes the array each time you run the procedure.

Example

To dynamically allocate an array named strProductNames that contains 20 strings, each with a fixed length of 25, enter the following:

```
ReDim strProductNames(20) As String * 25
```

Static Statement

Use a Static statement within a procedure to declare a variable used only in that procedure and that Visual Basic does not reinitialize while the module containing the procedure is open. Visual Basic opens all standard and class modules (objects you can see in the Modules list in the Navigation pane) when you open the database containing those objects. Visual Basic keeps form or report class modules open only while the form or the report is open.

Syntax

```
Static {variablename [({<array dimension>},...)]
  [As [New] datatype]},...
```

where <*array dimension*> is

```
[lowerbound To ] upperbound
```

Notes

If you do not include an <*array dimension*> specification but you do include the parentheses, you must include a ReDim statement in each procedure that uses the array to dynamically allocate the array at run time. You can define an array with up to 60 dimensions. If you do not include a *lowerbound* value in an <*array dimension*> specification, the default lower bound is 0. You can reset the default lower bound to 1 by including an *Option Base 1* statement in the module Declarations section. The *lowerbound* and *upperbound* values must be integers, and *upperbound* must be greater than or equal to *lowerbound*. The number of members of an array is limited only by the amount of memory on your computer.

Valid *datatype* entries are Byte, Boolean, Integer, Long, Currency, Single, Double, Date, String (for variable-length strings), String * *length* (for fixed-length strings), Object, Variant, or one of the object types described in this chapter. You can also declare a user-defined variable structure using the Type statement and then use the user type name as a data type. You should always explicitly declare the data type of your variables. If you do not include the As *datatype* clause, Visual Basic assigns the Variant data type.

Use the New keyword to indicate that a declared object variable is a new instance of an object that doesn't have to be set before you use it. You can use the New keyword only with object variables to create a new instance of that class of object without requiring a Set statement. You can't use New to declare dependent objects. If you do not use the New keyword, you cannot reference the object or any of its properties or methods until you set the variable to an object using a Set statement.

Visual Basic initializes declared variables at compile time. Numeric variables are initialized to zero (0), variant variables are initialized to empty, variable-length string variables are initialized as zero-length strings, and fixed-length string variables are filled with ANSI zeros (Chr(0)).

Examples

To declare a static variable named intMyInteger as an integer, enter the following:

```
Static intMyInteger As Integer
```

To declare a static array named strMyString that contains fixed-length strings that are 20 characters long and contains 50 entries from 51 through 100, enter the following:

```
Static strMyString(51 To 100) As String * 20
```

Type Statement

Use a Type statement in a Declarations section to create a user-defined data structure containing one or more variables.

Syntax

```
[Public | Private] Type typename
    {variablename [({<array dimension>},...)]
      As datatype}
    ...
End Type
```

where <*array dimension*> is

```
[lowerbound To ] upperbound
```

Notes

A Type statement is most useful for declaring sets of variables that can be passed to procedures, including Windows application programming interface (API) functions, as a single variable. You can also use the Type statement to declare a record structure. After you declare a user-defined data structure, you can use *typename* in any subsequent Dim, Public, Private, or Static statement to create a variable of that type. You can reference variables in a user-defined data structure variable by entering the variable name, a period, and the name of the variable within the structure. (See the second part of the example that follows.)

Include the Public keyword to declare a user-defined type that is available to all procedures in all modules in your database. Include the Private keyword to declare a user-defined type that is available only within the module in which the declaration is made. You must enter each *variablename* entry on a new line. You must indicate the end of your user-defined data structure using an End Type statement.

Valid *datatype* entries are Byte, Boolean, Integer, Long, Currency, Single, Double, Date, String (for variable-length strings), String * *length* (for fixed-length strings), Object, Variant, or one of the object types described earlier in this chapter. You can also declare a user-defined variable structure using the Type statement and then use the user type name as a data type. You should always explicitly declare the data type of your variables. If you do not include the As *datatype* clause, Visual Basic assigns the Variant data type.

If you do not include an <*array dimension*> specification but you do include the parentheses, you must include a ReDim statement in each procedure that uses the array to dynamically allocate the array at run time in any variable that you declare as this *Type*. You can define an array with as many as 60 dimensions. If you do not include a *lowerbound* value in an <*array dimension*> specification, the default lower bound is 0. You can reset the default lower bound to 1 by including an *Option Base 1* statement in the module Declarations section. The *lowerbound* and *upperbound* values must be integers, and *upperbound* must be greater than or equal to *lowerbound*. The number of members of an array is limited only by the amount of memory on your computer.

Note that a Type declaration does not reserve any memory. Visual Basic allocates the memory required by the Type statement when you use *typename* as a data type in a Dim, Public, Private, or Static statement.

Example

To define a user type structure named MyRecord containing a long integer and three string fields, declare a variable named usrContacts using that user type, and then set the first string to "Jones", first enter the following:

```
Type MyRecord
    lngID As Long
    strLast As String
    strFirst As String
    strMid As String
End Type
```

Then, within a procedure, enter the following:

```
Dim usrContacts As MyRecord
usrContacts.strLast = "Jones"
```

Collections, Objects, Properties, and Methods

You've already dealt with two of the main collections supported by Access 2010—Forms and Reports. The Forms collection contains all the form objects that are open in your application, and the Reports collection contains all the open report objects.

As you'll learn in more detail later in this section, collections, objects, properties, and methods are organized in several object model hierarchies. An object has *properties*, which describe the object, and *methods*, which are actions that you can ask the object to execute. For example, a Form object has a Name property (the name of the form) and a Requery method (to ask the form to requery its record source). Many objects also have *collections* that define sets of other objects within the object. For example, a Form object has a Controls collection, which is the set of all control objects (text boxes, labels, and so on) defined on the form.

You don't need a thorough understanding of collections, objects, properties, and methods to perform most application tasks. It's useful, however, for you to know how Access and Visual Basic organize these items so that you can better understand how Access works. If you want to study advanced code examples available in the many sample databases that you can download from public forums, you'll need to understand collections, objects, properties, and methods and how to correctly reference them.

The Access Application Architecture

An Access 2010 desktop application (.accdb or .mdb) has two major components—the application engine, which controls the programming and the user interface, and the Access Database Engine (DBEngine), which controls the storage of data and the definition of all the objects in your database. An Access project (.adp) also uses the application engine, but it depends on its Connection object to define a link to the Microsoft SQL Server database that contains the tables, views, functions, and stored procedures used by the application.

As you'll see in the next section, Visual Basic supports two distinct object models (Data Access Objects–DAO, and ActiveX Data Objects–ADO) for manipulating objects stored by the database engine. Figure 24-12 shows the application architecture of Access.

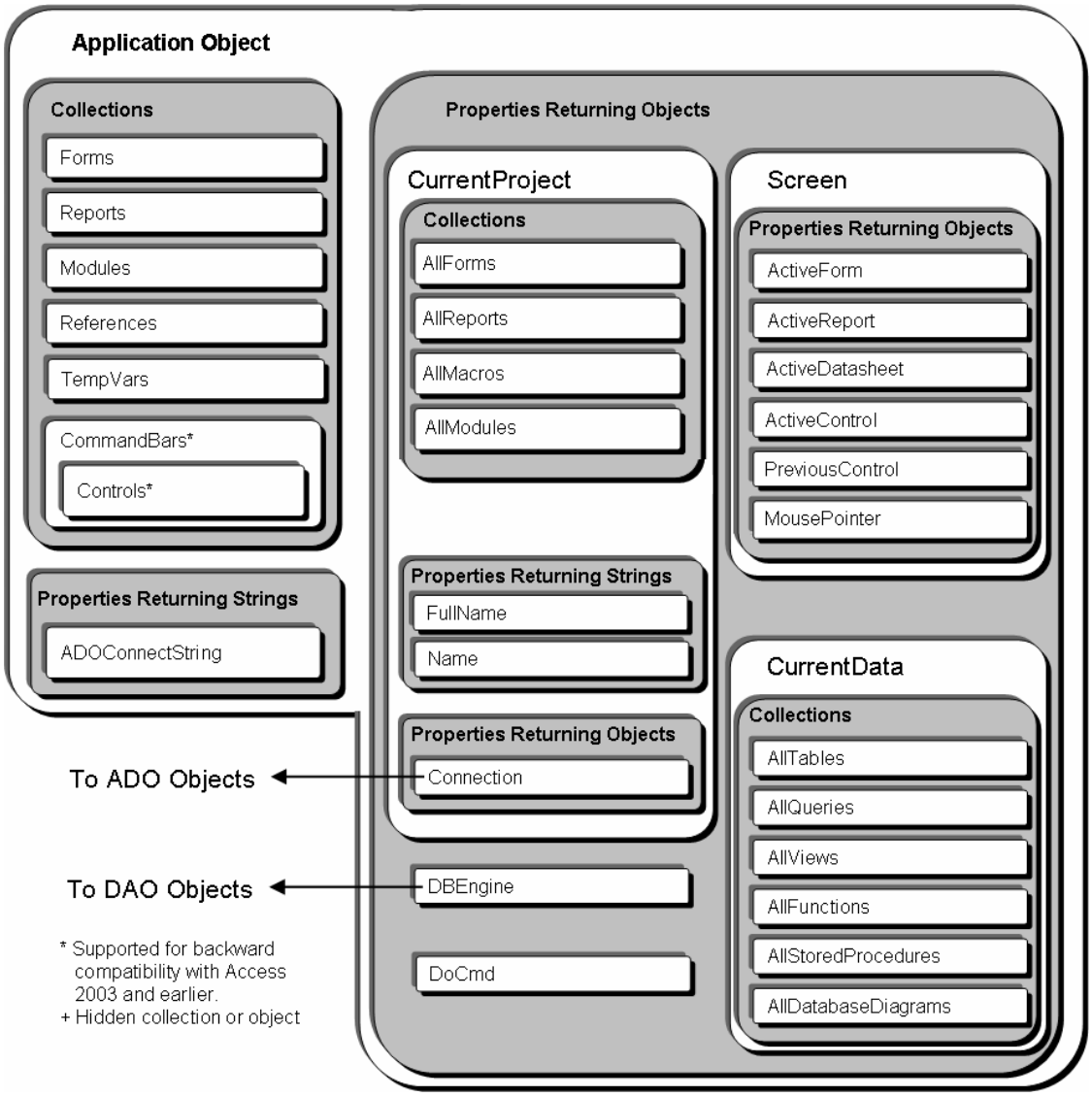

Figure 24-12 You can explore objects in the Access application architecture from the Application object.

When you open a database, the application engine loads the appropriate object collections from the database and application files to enable it to list the names of all the tables, queries, views, database diagrams, stored procedures, forms, reports, macros, and modules to display in the Navigation pane. The application engine establishes the top-level Application object, which contains a Forms collection (all the open forms), a Reports collection (all the open reports), a Modules collection (all the open modules, including form and report modules), and a References collection (all Visual Basic library references). Each form and report, in turn, contains a Controls collection (all of the controls on the form or report). Among some of the more interesting properties of the Application object is the ADOConnectString property that contains the information you can use to connect to this database from another database.

> **Note**
>
> **For backward compatibility with earlier versions and database files in the .mdb format, the Access object architecture continues to support obsolete collections, objects, and properties. For example, the Application object continues to support a CommandBars collection to allow you to manipulate any custom menus or toolbars that might have been defined using Access 2003 or earlier.**

The Application object also contains two special objects, the Screen object and the DoCmd object. The Screen object has six very useful properties: ActiveForm, ActiveReport, ActiveDatasheet, ActiveControl, PreviousControl, and MousePointer. Without knowing the actual names, you can reference the control (if any) that currently has the focus, the datasheet (if any) that has the focus, the form (if any) that has the focus, the report (if any) that has the focus, or the name of the control that previously had the focus. You can use the MousePointer property to examine the current status of the mouse pointer (arrow, I-beam, hourglass, and so on) and set the pointer. (Additional details about referencing properties of objects appear later in this chapter.) The DoCmd object lets you execute most macro actions within Visual Basic. For more information, see "Running Macro Actions and Menu Commands," on page 1549. If your application is an Access desktop database (.accdb), the DBEngine object under the Application object connects you to the Access Database Engine (ACE) to manipulate its objects using the DAO model.

Two properties allow you to directly find out the names of all objects stored in your database without having to call the database engine. In an Access desktop database (.accdb), you can find out the names of all your tables and queries via the CurrentData property. In an Access project file (.adp) that is connected to SQL Server, you also can learn the names of database diagrams, stored procedures, functions, and views via this same property. In either type of Access file, you can discover the names of all your forms, reports, macros, and modules via the CurrentProject property. Finally, the FullName property of the CurrentProject object tells you the full path and file name of your application file, and the Name property tells you the file name only.

The DAO Architecture

The first (and older) of the two models you can use to fetch data and examine or create new data objects is the DAO model. This model is best suited for use within Access desktop applications (.accdb and .mdb) because it provides objects, methods, and properties specifically tailored to the way Access and the ACE work together. To use this model, you must ask Visual Basic to load a reference to the Microsoft Office 14.0 Access Database Engine Object Library. To verify that your project includes this reference, open any module in Design view and click References on the Tools menu. If you don't see the check box for this library selected at the top of the References dialog box, scroll down the alphabetical list until you find the library, select its check box, and click OK to add the reference. Access 2010 creates this reference for you in any new database that you create.

INSIDE OUT

Is the Rumor That "DAO Is Dead" Really True?

Absolutely not! First, you need to know a bit of history. Beginning with version 9 (Access 2000), the Access development team introduced ADO to make it easier to work with SQL Server or other server databases as the data store for Access applications. ADO was touted as the "new direction" for data engine object models because it was designed to be more generic to work with different databases. Access 2000 also introduced the project file format (.adp) that lets you create an Access application linked directly to a database in SQL Server. Both Access 2000 and Access XP (2002) provided a default reference to the ADO library in new database, and you had to add the DAO library if you wanted to use it. Microsoft also declared DAO "stable" (read: no new enhancements) and began distributing the Access JET database engine as part of Microsoft Data Access Components (MDAC) that you install with your operating system—Microsoft Windows 98, Windows 2000, Windows XP, Windows Vista, or Windows 7. And so, the developer community began to think that DAO was "dead."

But DAO in many cases really works better if you're building a desktop application. DAO gives you direct access not only to all your table and query definitions but also forms, reports, macros, and modules. Also, the record source for all forms and reports creates a DAO recordset, so it doesn't make sense to try to use the entirely different ADO recordset object in your code. As of Access 2002, you can assign a recordset object you open in code directly to the Recordset property of a form. But if you're using an ADO recordset, features that you expect to work—such as updating across a join or autolookup when you set a foreign key—don't work correctly. In short, DAO was designed to work best with Access desktop applications.

When Microsoft stopped providing DAO as a default reference in new databases, many in the developer community pointed out to Microsoft that this really wasn't a good idea for desktop applications. Microsoft listened to its users and changed the default library back to DAO in Access 2003. However, the Access development team couldn't plan any major enhancements because the JET engine had become part of Windows.

For Access 2007, the development team created its own new version of the JET engine—now called the ACE. ACE includes the new features to support the Attachment and the Calculated data types, as well as multi-value fields, and it also supports all the features of the old JET engine, but uses an enhanced version of DAO. So no, DAO is not dead—it in a sense has been reborn in the new database engine for Access 2007 and Access 2010.

The Application object's DBEngine property serves as a bridge between the application engine and the Access Database Engine. The DBEngine property represents the DBEngine object, which is the top-level object in the DAO hierarchy. Figure 24-13 shows you a diagram of the hierarchy of collections defined in the DAO model.

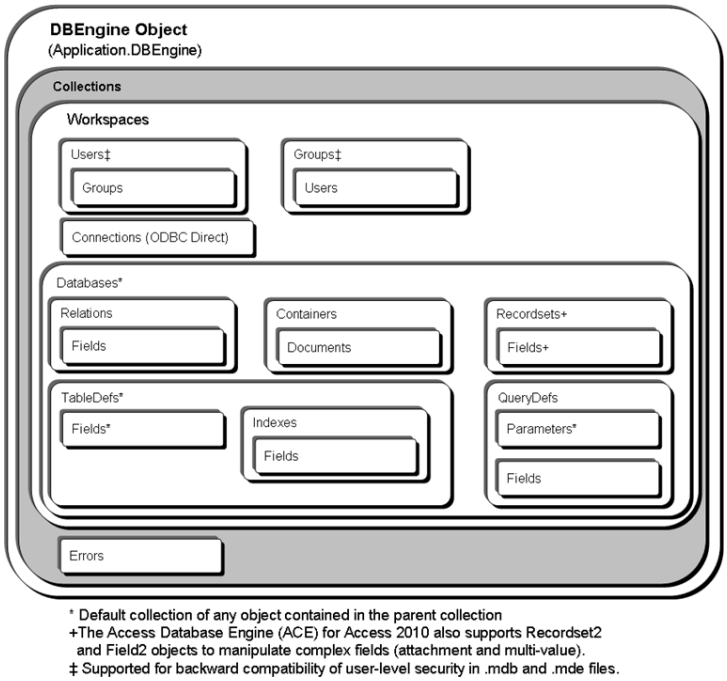

Figure 24-13 The DAO model is specifically designed to manipulate data objects in an Access desktop database.

The DBEngine object controls all the database objects in your database through a hierarchy of collections, objects, and properties. When you open an Access database, the DBEngine object first establishes a Workspaces collection and a default Workspace object (the first object in the Workspaces collection). If you are opening a secured database created in the prior version format (.mdb, .mde) and your workgroup is secured, Access prompts you for a password and a user ID so that the DBEngine can create a User object in the Users collection and a Group object in the Groups collection within the default workspace. If your workgroup is not secured, the DBEngine signs you in as a default user called Admin.

Finally, the DBEngine creates a Database object within the Databases collection of the default Workspace object. If your prior version format file is secured, the DBEngine uses the current User and/or Group object information to determine whether you're authorized to access any of the objects within the database.

After the DBEngine creates a Database object, the application engine determines whether the database contains any potentially untrustworthy objects. Any database containing tables, queries, macros, or Visual Basic code is deemed potentially untrustworthy. If the database is signed with a certificate that you have accepted as trustworthy or the database resides in a trusted location, the application engine enables all code. If the database is not trusted, the application engine displays a security warning message and provides the option to enable the content in the database.

Next, the application engine checks the database's application options to find out whether to open a display form, load an application icon, and display a title or to use one or more of the other application options. You can set these options when you have your database open by clicking the File tab on the Backstage view, clicking Options, and clicking the Current Database category in the Access Options dialog box. After checking the application options, the application engine checks to see whether a macro group named Autoexec exists in the database. If it finds Autoexec, the application engine runs this macro. In versions 1 and 2 of Access, you'd often use the Autoexec macro to open a startup form and run startup routines. In Access 2010, however, you should use the application options to specify a display form, and then use the event procedures or embedded macros of the startup form to run your startup routines.

See Chapter 26, "The Finishing Touches," on the companion CD, for details on creating startup properties and custom ribbons.

You can code Visual Basic procedures that can create additional Database objects in the Databases collection by opening additional .accdb files. Each open Database object has a Containers collection that the DBEngine uses to store the definition (using the Documents collection) of all your tables, queries, forms, reports, macros, and modules.

You can use the TableDefs collection to examine and modify existing tables. You can also create new TableDef objects within this collection. Each TableDef object within the TableDefs collection has a Fields collection that describes all the fields in the table, and an Indexes collection (with a Fields collection for each Index object) that describes any indexes that you created on the table. Likewise, the Relations collection contains Relation objects that describe how tables are related and what integrity rules apply between tables, and each Relation object has a Fields collection that describes the fields that participate in the relation.

The QueryDefs collection contains QueryDef objects that describe all the queries in your database. You can modify existing queries or create new ones. Each QueryDef object has a Parameters collection for any parameters required to run the query and a Fields collection that describes the Fields returned by the query. Finally, the Recordsets collection contains a Recordset object for each open recordset in your database, and the Fields collection of each Recordset object tells you the Fields in the recordset.

To reference any object within the DAO model, you can always start with the DBEngine object. If you want to work in the current database, that Database object is always the first database in the Databases collection of the first Workspace object. For example:

```
Dim dbMyDB As DAO.Database
Set dbMyDB = DBEngine.Workspaces(0).Databases(0)
```

Access also provides a handy shortcut object to the current database called CurrentDb. Therefore, you can also establish a pointer to the current database as follows:

```
Set dbMyDB = CurrentDb
```

> **Note**
>
> **In one of the examples at the end of this chapter, you'll learn how to create a new TableDef object and then open a Recordset object on the new table to insert rows. You can find code examples in the Conrad Systems Contacts application that manipulate objects using both DAO and ADO.**

The ADO Architecture

With Access 2000, Microsoft introduced a more generic set of data engine object models to provide references not only to objects stored by the ACE but also to data objects stored in other database products such as SQL Server. These models are called the ADO architecture. With Access 97 (version 8.0), you could download the Microsoft Data Access Components from the Microsoft website to be able to use the ADO model. Access 2000 and Access XP (2002) provided direct support for ADO with built-in libraries and direct references to key objects in the model from the Access Application object. As noted earlier, Access 2003, Access 2007, and Access 2010 provide a default reference to the DAO library, not ADO.

Because these models are designed to provide a common set of objects across any data engine that supports the ADO, they do not necessarily support all the features you can find in the DAO architecture that was specifically designed for the ACE. For this reason, if you are designing an application that will always run with the ACE, you are better off using the DAO model. If, however, you expect that your application might one day "upsize" to an ActiveX data engine such as SQL Server, you should consider using the ADO architecture as much as possible. If you create your Access application as an Access project (.adp) linked to SQL Server, you should use only the ADO models.

Figure 24-14 shows you the two major models available in the ADO architecture. The basic ActiveX Data Objects (ADODB) model lets you open and manipulate recordsets via the Recordset object and execute action or parameter queries via the Command object. The ADO Extensions for DDL and Security (ADOX) model allows you to create, open, and manipulate tables, views (non-parameter unordered queries), and procedures (action queries, parameter queries, ordered queries, functions, triggers, or procedures) within the data engine Catalog object (the object that describes the definition of objects in your database). You can also examine and define Users and Groups collections defined in the Catalog object with ADOX.

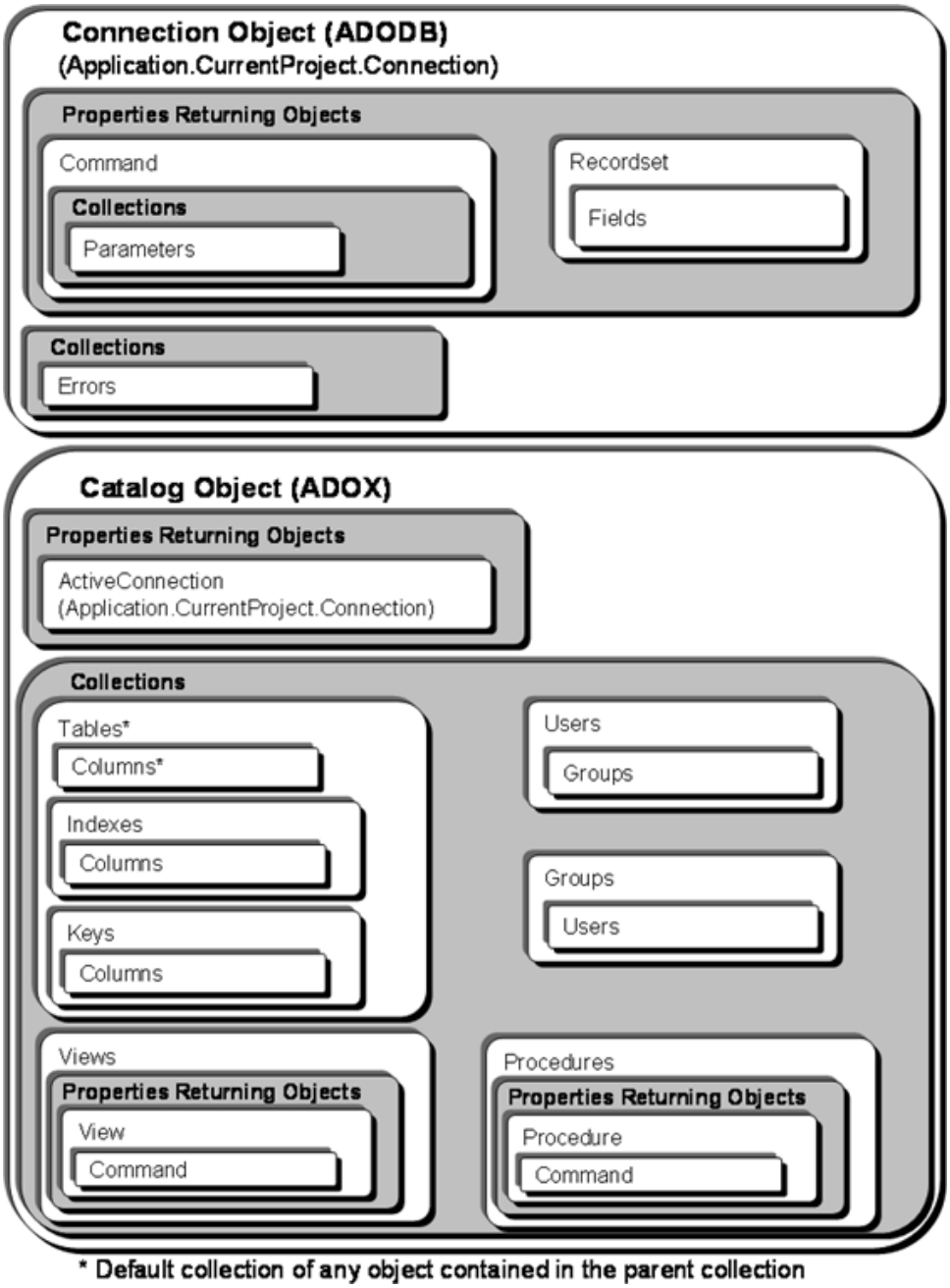

Figure 24-14 The ADODB and ADOX models provide another way to work with the data and objects in your database.

To use the ADODB model, you must instruct Visual Basic to load a reference to the Microsoft ActiveX Data Objects Library. For objects in the ADOX model, you need the Microsoft ADO Extensions for DDL and Security Library. (You should normally find only one version on your computer. If you find multiple versions in the list, select the latest one.) To verify that your project includes these references, open any module in Design view and click References on the Tools menu. If you don't see the check boxes for these libraries selected at the top of the References dialog box, scroll down the alphabetical list until you find the library you need, select its check box, and click OK to add the reference. Access 2010 does not automatically create a reference to the ADODB library for you in any new database that you create.

Note that there are some objects in common between DAO, ADODB, and ADOX. If you use multiple models in an application, you must be careful to qualify object declarations. For example, a Recordset object type in the DAO model is DAO.Recordset, whereas a Recordset in the ADODB model is ADODB.Recordset. You cannot freely interchange a DAO recordset with an ADODB recordset—they are completely different objects.

The link to ADODB and ADOX is via the CurrentProject.Connection property. After you open an ADODB.Connection object, you can work with other collections, objects, and properties within the ADODB model. Likewise, by establishing an ADOX.Catalog object and setting its Connection property, you can work with any collection, object, or property within the ADOX model.

For all objects within either ADODB or ADOX, you must first establish a base object (connection or catalog, respectively). For example:

```
Dim cn As ADODB.Connection, rst As New ADODB.Recordset
Set cn = CurrentProject.Connection
rst.Open = "tblContacts", cn
```

or

```
Dim catThisDB As New ADOX.Catalog, tbl As ADOX.Table
Set catThisDB.ActiveConnection = CurrentProject.Connection
Set tbl = catThisDB.Tables("tblContacts")
```

Note

One of the extensive examples at the end of this chapter uses ADO exclusively to manipulate recordsets in the Conrad Systems Contacts sample database.

Referencing Collections, Objects, and Properties

In Chapter 23, "Using Business Connectivity Services," you were introduced to the most common way to reference objects in the Forms and Reports collections, controls on open forms and reports, and properties of controls. There are two alternative ways to reference an object within a collection. The three ways to reference an object within a collection are as follows:

- **CollectionName![Object Name]** This is the method you used in Chapter 20, "Automating a Client Application Using Macros," and Chapter 21, "Automating a Web Application Using Macros." For example: Forms![frmContacts].
- **CollectionName("Object Name")** This method is similar to the first method but uses a string constant (or a string variable) to supply the object name. For example: Forms("frmContacts") and Forms(strFormName).
- **CollectionName(RelativeObjectNumber)** Visual Basic numbers objects within most collections from zero (0) to CollectionName.Count minus 1. You can determine the number of open forms by referring to the Count property of the Forms collection. For example: Forms.Count. You can refer to the second open form in the Forms collection as Forms(1).

Forms and Reports are relatively simple because they are top-level collections within the application engine. As you saw in Figure 24-13, when you reference a collection or an object maintained by the DBEngine, the hierarchy of collections and objects is quite complex. If you want to find out the number of Workspace objects that exist in the Workspaces collection, for example, you need to reference the Count property of the Workspaces collection like this:

```
DBEngine.Workspaces.Count
```

(You can create additional workspaces from Visual Basic code.)

Using the third technique described earlier to reference an object, you can reference the default (first) Workspace object by entering the following:

```
DBEngine.Workspaces(0)
```

Likewise, you can refer to the currently open database in a desktop application (.accdb) by entering the following:

```
DBEngine.Workspaces(0).Databases(0)
```

When you want to refer to an object that exists in an object's default (or only) collection (see Figures 22-13 and 22-14), you do not need to include the collection name. Therefore, because the Databases collection is the default collection for the Workspaces collection, you can also refer to the currently open database by entering the following:

```
DBEngine.Workspaces(0)(0)
```

As you can see, even with this shorthand syntax, object names can become quite cumbersome if you want to refer, for example, to a particular field within an index definition for a table within the current database in the default Workspace object—or a column within an index definition for a table within the current catalog. For example, using this full syntax, you can reference the name of the first field in the tblContacts table in Contacts.accdb like this:

```
DBEngine(0)(0).TableDefs("tblContacts").Fields(0).Name
```

(Whew!) If for no other reason, object variables are quite handy to help minimize name complexity.

In particular, you can reduce name complexity by using an object variable to represent the current database. When you set the variable to the current database, you can call the CurrentDb function rather than use the database's full qualifier. For example, you can declare a Database object variable, set it to the current database by using the CurrentDb function, and then use the Database object variable name as a starting point to reference the TableDefs, QueryDefs, and Recordsets collections that it contains. (See "Assigning an Object Variable—Set Statement," on page 1509, for the syntax of the Set statement.) Likewise, if you are going to work extensively with fields in a TableDef object or columns in a Table object, you are better off establishing an object variable that points directly to the TableDef or Table object. For example, you can simplify the complex expression to reference the name of the first field in the tblContacts table in Contacts.accdb like this:

```
Dim db As DAO.Database, tdf As DAO.TableDef
Set db = CurrentDb
Set tdf = db.Tabledefs![tblContacts]
Debug.Print tdf.Fields(0).Name
```

INSIDE OUT

Should I Use CurrentDb or DBEngine.Workspaces(0).Databases(0)?

When you use DBEngine.Workspaces(0).Databases(0) (or DBEngine(0)(0)) to set a database object, Visual Basic establishes a pointer to the current database. You can have only one object variable set to the actual copy of the current database, and you must never close this copy. A safer technique is to set your database variable using the CurrentDb function. Using this technique opens a new database object that is based on the same database as the current one. You can have as many copies of the current database as you like, and you can close them when you finish using them. When you use CurrentDb to establish a pointer to your database, Visual Basic refreshes all the collections and keeps them current. If you want to ensure that the collections are current (for example, to be aware of any added or deleted tables or queries), you must refresh the collections yourself when you use DBEngine(0)(0). The one small advantage to DBEngine(0)(0) is that it is more efficient because it does not refresh all collections when you establish a pointer to it.

When to Use "!" and "."

You've probably noticed that a complex, fully qualified name of an object or a property in Access 2010 or Visual Basic contains exclamation points (!) and periods (.) that separate the parts of the name.

Use an exclamation point preceding a name when the name refers to an object that is in the preceding object or collection of objects. A name following an exclamation point is generally the name of an object you created (such as a form or a table). Names following an exclamation point must be enclosed in brackets ([ ]) if they contain embedded blank spaces or a special character, such as an underscore (_). You must also enclose the name of an object you created in brackets if the name is also an Access or SQL reserved word. For example, most objects have a Name property—if you name a control or field "Name," you must use brackets when you reference your object.

To make this distinction clear, you might want to get into the habit of always enclosing in brackets names that follow an exclamation point, even though brackets are not required for names that don't use blank spaces or special characters. Access automatically inserts brackets around names in property sheets, design grids, and action arguments.

Use a period preceding a name that refers to a collection name, a property name, or the name of a method that you can perform against the preceding object. (Names following a period should never contain blank spaces.) In other words, use a period when the following name is *of* the preceding name (as in the TableDefs collection *of* the Databases(0) object, the Count property of the TableDefs collection, or the MoveLast method of the DAO Recordset object). This distinction is particularly important when referencing something that has the same name as the name of a property. For example, the reference

```
DBEngine.Workspaces(0).Databases(0).TableDefs(18).Name
```

refers to the name of the 19th TableDef object in the current database. In the Contacts.accdb database, if you use Debug.Print or the Immediate window to display this reference, Visual Basic returns the value *tblCompanies*. However, the reference

```
DBEngine.Workspaces(0).Databases(0).TableDefs(18)![Name]
```

refers to the contents of a field called Name (if one exists) in the 19th TableDef object in the current database. In the Conrad Systems Contacts database, this reference returns an error because there is no Name field in the tblCompanies table.

INSIDE OUT What About Me?

If you spend some time looking at any of the code behind forms and reports in the sample databases, you'll notice many references such as Me.Name or Me.ProductName. Whenever you write code in a form or report module, you'll likely need to reference some of the controls on the form or report or some of the properties of the form or report. You already know that you can reference an open form by using, for example,

```
Forms![frmProducts]
```

and to reference a control on the open frmProducts form, you could use

```
Forms![frmProducts]![ProductName]
```

Rather than type the collection name (Forms) and the form name (frmProducts) each time, you can use a shortcut—Me. This special keyword is a reference to the object where your code is running. Also, when Access opens a form, it loads the names of all controls you defined on the form as properties of the form—which are also properties of the Me object. (It also does the same for controls on open reports.) Therefore, you can reference the ProductName control in code behind the frmProducts form by entering

```
Me.ProductName
```

This can certainly make entering code faster. Also, because Me is an object, your code executes more quickly.

INSIDE OUT

Is It Possible to Reference in Visual Basic Variables Created by Macros?

You bet! As you learned in Chapter 20, you can use SetTempVar, RemoveTempVar, and RemoveAllTempVars actions to create, modify, and inspect values that you can pass from one macro to another. If you create an application that uses both macros and Visual Basic, you can also create, modify, and inspect these variables by using the TempVars collection. Unlike most collections in Access where you must first create an object before you can reference it, you can both create and set a macro temporary variable by simply assigning a value to a name in the TempVars collection. For example, to create and set a temporary variable called MyTempVar, use the following:

```
TempVars!MyTempVar = "Value to pass to a macro"
```

Temporary variables are the Variant data type, so you assign a string, a number, or a date/time value to a member of the TempVars collection. To delete a temporary variable, use the Removemethod as follows:

```
TempVars.Remove MyTempVar
```

To remove all temporary variables, use the RemoveAll method as follows:

```
TempVars.RemoveAll
```

However, be careful. If you reference a temporary variable that does not exist yet, you won't get any error. If you misspell a temporary variable name, Access temporarily creates the variable and returns the value Null.

Assigning an Object Variable—Set Statement

Use the Set statement to assign an object or object reference to an object variable.

Syntax

Set *objectvariablename* = [**New**] *objectreference*

Notes

As noted earlier, you can use object variables to simplify name references. Also, using an object variable is less time-consuming than using a fully qualified name. At run time, Visual Basic must always parse a qualified name to first determine the type of object and then determine which object or property you want. If you use an object variable, you have already defined the type of object and established a direct pointer to it, so Visual Basic can quickly go to that object. This is especially important if you plan to reference, for example,

many controls on a form. If you create a form variable first and then assign the variable to point to the form, referencing controls on the form via the form variable is much simpler and faster than using a fully qualified name for each control.

You must first declare *objectvariablename* using a Dim, Private, Public, or Static statement. The object types you can declare include AccessObject, Application, ADOX.Catalog, ADOX.Column, ADODB.Command, ADOX.Command, ADODB.Connection, DAO.Container, Control, DAO.Database, DAO.Document, ADODB.Error, DAO.Error, ADODB.Field, DAO.Field, DAO.Field2, Form, ADOX.Group, DAO.Group, ADOX.Index, DAO.Index, ADOX.Key, ADODB.Parameter, DAO.Parameter, ADOX.Procedure, ADODB.Property, ADOX.Property, DAO.Property, DAO.QueryDef, ADODB.Recordset, DAO.Recordset, DAO.Recordset2, DAO.Relation, Report, ADOX.Table, DAO.TableDef, ADOX.User, DAO.User, ADOX.View, and DAO.Workspace object. You can also declare a variable as the generic Object data type and set it to any object (similar to the Variant data type). In addition, you can declare a variable as an instance of the class defined by a class module. The object type must be compatible with the object type of *objectreference*. You can use another object variable in an *objectreference* statement to qualify an object at a lower level. (See the examples that follow.) You can also use an object *method* to create a new object in a collection and assign that object to an object variable. For example, it's common to use the OpenRecordset method of a QueryDef or TableDef object to create a new Recordset object. See the example in the next section, "Object Methods."

An object variable is a reference to an object, not a copy of the object. You can assign more than one object variable to point to the same object and change a property of the object. When you do that, all variables referencing the object will reflect the change as well. The one exception is that several Recordset variables can refer to the same recordset, but each can have its own Bookmark property pointing to different rows in the recordset. If you want to create a new instance of an object, include the New keyword.

Examples

To create a variable reference to the current database, enter the following:

```
Dim dbMyDB As DAO.Database
Set dbMyDB = CurrentDb
```

To create a variable reference to the tblContacts table in the current database using the dbMyDB variable defined earlier, enter the following:

```
Dim tblMyTable As DAO.TableDef
Set tblMyTable = dbMyDB![tblContacts]
```

Notice that you do not need to explicitly reference the TableDefs collection of the database, as in dbMyDB.TableDefs![tblContacts] or dbMyDB.TableDefs("tblContacts"), because TableDefs is the default collection of the database. Visual Basic assumes that [tblContacts] refers to the name of an object in the default collection of the database.

To create a variable reference to the Notes field in the tblContacts table using the tblMyTable variable defined earlier, enter the following:

```
Dim fldMyField As DAO.Field
Set fldMyField = tblMyTable![Notes]
```

Again, you do not need to include a specific reference to the Fields collection of the TableDef object, as in tblMyTable.Fields![Notes], because Fields is the default collection.

To create a variable reference to the catalog for the current database, enter the following:

```
Dim catThisDB As New ADOX.Catalog
catThisDB.ActiveConnection = CurrentProject.Connection
```

Note that you must use the New keyword because there's no way to open an existing catalog without first establishing a connection to it. You open a catalog by declaring it as a new object and assigning a Connection object to its ActiveConnection property. The example earlier takes advantage of the existence of the Application.CurrentProject.Connection property rather than first setting a Connection object. If you already have another Catalog object open, you can create a copy of it by using

```
Dim catCopy As ADOX.Catalog
Set catCopy = catThisDB
```

To create a variable reference to the tblContacts table in the current database using the catThisDB variable defined earlier, enter the following:

```
Dim tblMyTable As ADOX.Table
Set tblMyTable = catThisDB![tblContacts]
```

Notice that you do not need to explicitly reference the Tables collection of the database, as in catThisDB.Tables![tblContacts] or catThisDB.Tables("tblContacts"), because Tables is the default collection of the catalog. Visual Basic assumes that [tblContacts] refers to the name of an object in the default collection of the catalog.

To create a variable reference to the Notes column in the tblContacts table using the tblMyTable variable defined earlier, enter the following:

```
Dim colMyColumn As ADOX.Column
Set colMyColumn = tblMyTable![Notes]
```

Again, you do not need to explicitly reference the Columns collection of the Table object, as in tblMyTable.Columns![Notes], because the Columns collection is the default collection of a Table object.

Object Methods

When you want to apply an action to an object in your database (such as open a query as a recordset or go to the next row in a recordset), you apply a *method* of either the object or an object variable that you have assigned to point to the object. In some cases, you'll use a method to create a new object. Many methods accept parameters that you can use to further refine how the method acts on the object. For example, you can tell the DAO OpenRecordset method whether you're opening a recordset on a local table, a dynaset (a query-based recordset), or a read-only snapshot.

Visual Basic supports many different object methods—far more than there's room to properly document in this book. Perhaps one of the most useful groups of methods is the group you can use to create a recordset and then read, update, insert, and delete rows in the recordset.

Working with DAO Recordsets

To create a recordset, you must first declare a Recordset object variable. Then open the recordset using the DAO OpenRecordset method of the current database (specifying a table name, a query name, or an SQL statement to create the recordset) or the OpenRecordset method of a DAO.QueryDef, DAO.TableDef, or other DAO.Recordset object. (As you'll learn in "Working with ADO Recordsets," on page 1520, if you're working in ADO, you use the Open method of a New ADODB.Recordset object.)

In DAO, you can specify options to indicate whether you're opening the recordset as a local table (which means you can use the Seek method to quickly locate rows based on a match with an available index), as a dynaset, or as a read-only snapshot. For updateable recordsets, you can also specify that you want to deny other updates, deny other reads, open a read-only recordset, open the recordset for append only, or open a read-only forward scroll recordset (which allows you to move only forward through the records and only once).

The syntax to use the OpenRecordset method of a Database object is as follows:

```
Set RecordSetObject =  DatabaseObject.OpenRecordset(source,
    [type], [options], [lockoptions])
```

RecordSetObject is a variable you have declared as DAO.Recordset, and *DatabaseObject* is a variable you have declared as DAO.Database. Source is a string variable or literal containing the name of a table, the name of a query, or a valid SQL statement. Table 24-2 describes the settings you can supply for *type*, *options*, and *lockoptions*.

Table 24-2 OpenRecordset Parameter Settings

Setting	Description
TYPE (SELECT ONE)	
dbOpenTable	Returns a table recordset. You can use this option only when *source* is a table local to the database described by the Database object. *Source* cannot be a linked table. You can establish a current index in a table recordset and use the Seek method to find rows using the index. If you do not specify a *type*, OpenRecordset returns a table if *source* is a local table name.
dbOpenDynaset	Returns a dynaset recordset. *Source* can be a local table, a linked table, a query, or an SQL statement. You can use the Find methods to search for rows in a dynaset recordset. If you do not specify a *type*, OpenRecordset returns a dynaset if *source* is a linked table, a query, or an SQL statement.
dbOpenSnapshot	Returns a read-only snapshot recordset. You won't see any changes made by other users after you open the recordset. You can use the Find methods to search for rows in a snapshot recordset.
dbOpenForwardOnly	Returns a read-only snapshot recordset that you can move forward through only once. You can use the MoveNext method to access successive rows.
OPTIONS (YOU CAN SELECT MULTIPLE OPTIONS, PLACING A PLUS SIGN BETWEEN OPTION NAMES TO ADD THEM TOGETHER)	
dbAppendOnly	Returns a table or dynaset recordset that allows inserting new rows only. You can use this option only with the dbOpenTable and dbOpenDynaset types.
dbSeeChanges	Asks Access to generate a run-time error in your code if another user changes data while you are editing it in the recordset.
dbDenyWrite	Prevents other users from modifying or inserting records while your recordset is open.
dbDenyRead	Prevents other users from reading records in your open recordset.
dbInconsistent	Allows you to make changes to all fields in a multiple table recordset (based on a query or an SQL statement), including changes that would be inconsistent with any join defined in the query. For example, you could change the customer identifier field (foreign key) of an orders table so that it no longer matches the primary key in an included customers table–unless referential integrity constraints otherwise prevent you from doing so. You cannot include both dbInconsistent and dbConsistent.
dbConsistent	Allows you to only make changes in a multiple table recordset (based on a query or an SQL statement) that are consistent with the join definitions in the query. For example, you cannot change the customer identifier field (foreign key) of an orders table so that its value does not match the value of any customer row in the query. You cannot include both dbInconsistent and dbConsistent.

Setting	Description
LOCKOPTIONS (SELECT ONE)	
dbPessimistic	Asks Access to lock a row as soon as you place the row in an editable state by executing an Edit method. This is the default if you do not specify a lock option.
dbOptimistic	Asks Access to not attempt to lock a row until you try to write it to the database with an Update method. This generates a run-time error if another user has changed the row after you executed the Edit method.

For example, to declare a recordset for the tblFacilities table in the Housing Reservations (Housing.accdb) database and open the recordset as a table so that you can use its indexes, enter the following:

```
Dim dbHousing As DAO.Database
Dim rcdFacilities As DAO.RecordSet
Set dbHousing = CurrentDb
Set rcdFacilities = dbHousing.OpenRecordSet("tblFacilities", _
  dbOpenTable)
```

To open the qryContactProducts query in the Conrad Systems Contacts database (Contacts.accdb) as a dynaset, enter the following:

```
Dim dbContacts As DAO.Database
Dim rcdContactProducts As DAO.RecordSet
Set dbContacts = CurrentDb
Set rcdContactProducts = _
  dbContacts.OpenRecordSet("qryContactProducts")
```

(Note that opening a recordset as a dynaset is the default when the source is a query.)

> **Note**
>
> **Any table recordset or dynaset recordset based on a table is updateable. When you ask Access to open a dynaset on a table, Access internally builds a query that selects all columns from the table. A dynaset recordset based on a query will be updateable if the query is updateable. See "Limitations on Using Select Queries to Update Data," on page 680, for details.**

After you open a recordset, you can use one of the Move methods to move to a specific record. Use *recordset*.MoveFirst to move to the first row in the recordset. Other Move methods include MoveLast, MoveNext, and MovePrevious. If you want to move to a specific row in a dynaset recordset, use one of the Find methods. You must supply a string variable

containing the criteria for finding the records you want. The criteria string looks exactly like a WHERE clause in SQL, but without the WHERE keyword. (See Article 2, "Understanding SQL," on the companion CD, for more details.) For example, to find the first row in the qryContactProducts query's recordset whose SoldPrice field is greater than $200, enter the following:

```
rcdContactProducts.FindFirst "SoldPrice > 200"
```

To delete a row in an updateable recordset, move to the row you want to delete and then use the Delete method. For example, to delete the first row in the qryContactProducts query's recordset that hasn't been invoiced yet (the Invoiced field is false), enter the following:

```
Dim dbContacts As DAO.Database
Dim rcdContactProducts As DAO.RecordSet
Set dbContacts = CurrentDb
Set rcdContactProducts = _
  dbContacts.OpenRecordSet("qryContactProducts")
rcdContactProducts.FindFirst "Invoiced = 0"
' Test the recordset NoMatch property for "not found"
If Not rcdContactProducts.NoMatch Then
    rcdContactProducts.Delete
End If
```

If you want to update rows in a recordset, move to the first row you want to update and then use the Edit method to lock the row and make it updateable. You can then refer to any of the fields in the row by name to change their values. Use the Update method on the recordset to save your changes before moving to another row. If you do not use the Update method before you move to a new row or close the recordset, the database discards your changes.

For example, to increase by 10 percent the SoldPrice entry of the first row in the rcdContactProducts query's recordset whose SoldPrice value is greater than $200, enter the following:

```
Dim dbContacts As DAO.Database
Dim rcdContactProducts As DAO.RecordSet
Set dbContacts = CurrentDb
Set rcdContactProducts = _
  dbContacts.OpenRecordSet("qryContactProducts")
rcdContactProducts.FindFirst "SoldPrice > 200"
 ' Test the recordset NoMatch property for "not found"
If Not rcdContactProducts.NoMatch Then
  rcdContactProducts.Edit
  rcdContactProducts![SoldPrice] = _
    rcdContactProducts![SoldPrice] * 1.1
  rcdContactProducts.Update
End If
```

To insert a new row in a recordset, use the AddNew method to start a new row. Set the values of all required fields in the row, and then use the Update method to save the new row. For example, to insert a new company in the Conrad Systems Contacts tblCompanies table, enter the following:

```
Dim dbContacts As DAO.Database
Dim rcdCompanies As DAO.RecordSet
Set dbContacts = CurrentDb
Set rcdCompanies = _
  dbContacts.OpenRecordSet("tblCompanies")
rcdCompanies.AddNew
rcdCompanies![CompanyName] = "Winthrop Brewing Co."
rcdCompanies![Address] = "155 Riverside Ave."
rcdCompanies![City] = "Winthrop"
rcdCompanies![StateOrProvince] = "WA"
rcdCompanies![PostalCode] = "98862"
rcdCompanies![PhoneNumber] = "(509) 555-8100"
rcdCompanies.Update
```

Note that because all the main data tables in Contacts.accdb are linked tables, rcdCompanies is a dynaset recordset, not a table recordset. If you want to position the recordset on the newly created record, you could add an rcdCompanies.Bookmark=rcdCompanies.LastModified line at the end of the preceding code example.

Manipulating Complex Data Types Using DAO

Access 2010 supports complex data types—the Attachment data type or any field defined as multi-value. A complex data type lets you store multiple values or objects in a field within a single record. Access 20107 accomplishes this by building hidden tables that contain one row per multiple value stored. You can manipulate these rows in a recordset in code, but only using DAO.

To work with data in a complex data type field, you must first open a recordset on the table containing the field. You can either open the table directly or open a query that includes the table and its complex field(s). The secret to dealing with complex fields is the Value property of the field in the recordset returns a DAO.Recordset2 object. Therefore, you can set a declared DAO.Recordset2 variable to the Value property to open a recordset on the hidden table. You can manipulate this recordset exactly as you can any other DAO recordset, including using the Find, Move, Edit, AddNew, Update, and Delete methods.

When the complex field is a multi-value field, the recordset returned from the Value property of the parent field contains a single field called Value. You'll find one row per multiple value stored in the complex field. When the complex field is an Attachment data type,

the recordset returned from the Value property of the parent field contains three fields—FileData, FileName, and FileType. The FileData field in an attachment complex recordset supports one method, LoadFromFile, that lets you insert the complex Object Linking and Embedding (OLE) data into the record by supplying a file location and name.

The tblContacts table in the Contacts sample database contains both a multi-value field (ContactType) and an attachment field (Photo). In the modExamples module in the Contacts.accdb database, you can find the following code, which displays in the Immediate window the values from both fields for all contact records:

```
Public Sub ListContactComplex()
' An example of listing all the complex values in the Contacts table
Dim db As DAO.Database, rst As DAO.Recordset, rstComplex As DAO.Recordset2
Dim fld As DAO.Field2
  ' Point to this database
  Set db = CurrentDb
  ' Open a recordset on tblContacts
  Set rst = db.OpenRecordset("SELECT * FROM tblContacts")
  ' Loop through all the records
  Do Until rst.EOF
    ' Dump out the ID and name
    Debug.Print rst!ContactID, rst!LastName, rst!FirstName
    ' Get the contact type complex field
    Set rstComplex = rst!ContactType.Value
    ' Loop through them all
    Do Until rstComplex.EOF
      ' Dump out each value
      Debug.Print " ", "Contact Type: ", rstComplex!Value
      ' Get the next
      rstComplex.MoveNext
    Loop
    ' Get the Photo Attachment recordset
    Set rstComplex = rst!Photo.Value
    ' Loop though them all
    Do Until rstComplex.EOF
      ' Dump out the data
      Debug.Print " ", "Photo FileName: ", rstComplex!FileName, _
        "  File Type: ", rstComplex!FileType
      ' Get the next
      rstComplex.MoveNext
    Loop
    ' Get the next contact
    rst.MoveNext
  Loop
  ' Close out
  rst.Close
  Set rst = Nothing
  Set rstComplex = Nothing
  Set db = Nothing
End Sub
```

If you want to find the record for John Viescas and add the Trainer value to the Contact-Type field, use the following code:

```
Public Sub AddContactTypeViescas()
Dim db As DAO.Database, rst As DAO.Recordset, rstComplex As DAO.Recordset2
  ' Set a pointer to the current database
  Set db = CurrentDb
  ' Open the contacts table
  Set rst = db.OpenRecordset("tblContacts", dbOpenDynaset)
  ' Find the record for Viescas
  rst.FindFirst "LastName = 'Viescas'"
  ' Make sure we found it
  If Not rst.NoMatch Then
    ' Put parent record in Edit
    rst.Edit
    ' Get the ContactType recordset
    Set rstComplex = rst!ContactType.Value
    ' Add a new row
    rstComplex.AddNew
    ' Insert the new value
    rstComplex.Value = "Trainer"
    ' Save the new value
    rstComplex.Update
    ' Now save the parent
    rst.Update
  End If
  ' Close out
  rst.Close
  Set rst = Nothing
  Set rstComplex = Nothing
  Set db = Nothing
```

To find the contact record for John Viescas, check for the Trainer value in the ContactType field, and delete it if it exists, use the following code:

```
Public Sub DeleteTrainerFromViescas()
Dim db As DAO.Database, rst As DAO.Recordset, rstComplex As DAO.Recordset2
  ' Set a pointer to the current database
  Set db = CurrentDb
  ' Open the contacts table
  Set rst = db.OpenRecordset("tblContacts", dbOpenDynaset)
  ' Find the record for Viescas
  rst.FindFirst "LastName = 'Viescas'"
  ' Make sure we found it
  If Not rst.NoMatch Then
    ' Get the ContactType recordset
    Set rstComplex = rst!ContactType.Value
    ' See if Trainer exists
    rstComplex.FindFirst "Value = 'Trainer '"
    ' If it exists,
```

```
    If Not rstComplex.NoMatch Then
      ' Delete it
      rstComplex.Delete
    End If
  End If
  ' Close out
  rst.Close
  Set rst = Nothing
  Set rstComplex = Nothing
  Set db = Nothing
```

To check whether the Photo field for contact Jeff Conrad contains a file named JeffConrad.docx and add it if it does not, use the following code:

```
Public Sub AddDocumentToConradPhotoField()
Dim db As DAO.Database, rst As DAO.Recordset, rstComplex As DAO.Recordset2
  ' Set a pointer to the current database
  Set db = CurrentDb
  ' Open the contacts table
  Set rst = db.OpenRecordset("tblContacts", dbOpenDynaset)
  ' Find the record for Conrad
  rst.FindFirst "LastName = 'Conrad'"
  ' Make sure we found it
  If Not rst.NoMatch Then
    ' Get the Photo recordset
    Set rstComplex = rst!Photo.Value
    ' See if the JeffConrad.docx file exists
    rstComplex.FindFirst "FileName = 'JeffConrad.docx'"
    ' If it does not exist,
    If rstComplex.NoMatch Then
      ' Put parent record in Edit
      rst.Edit
      ' Start a new attachment record
      rstComplex.Addnew
      ' Load the file
      rstComplex!FileData.LoadFromFile _
        "C:\Microsoft Press\Access 2010 Inside Out\Documents\Jeff Conrad.docx"
      ' Save the new row
      rstComplex.Update
      ' Save the parent row
      rst.Update
    End If
  End If
  ' Close out
  rst.Close
  Set rst = Nothing
  Set rstComplex = Nothing
  Set db = Nothing
```

Working with ADO Recordsets

Recordsets in ADO offer many of the same capabilities and options as recordsets in DAO, but the terminology is somewhat different. Because you will most often use ADO with data stored in a server database such as SQL Server, the options for an ADO recordset are geared toward server-based data. For example, ADO uses the term *cursor* to refer to the set of rows returned by the server. Fundamentally, a cursor is a pointer to each row you need to work with in code. Depending on the options you choose (and the options supported by the particular database server), a cursor might also be read-only, updateable, or forward-only. A cursor might also be able to reflect changes made by other users of the database (a key-set or dynamic cursor), or it might present only a snapshot of the data (a static cursor).

To open an ADO recordset, you must use the Open method of a new ADO Recordset object. The syntax to use the Open method of a Recordset object is as follows:

```
RecordSetObject.Open [source], [connection],
  [cursortype], [locktype], [options]
```

RecordSetObject is a variable you have declared as a New ADO.Recordset. *Source* is a Command object, a string variable, or string literal containing the name of a table, the name of a view (the SQL Server term for a query), the name of a stored procedure, the name of a function that returns a table, or a valid SQL statement. A stored procedure might be a parameter query or a query that specifies the sorting of rows from a table or view. A function might also accept parameters. If you supply a Command object as the source, you do not need to supply a *connection* (you define the connection in the Command object). Otherwise, *connection* must be the name of a Connection object that points to the target database.

Table 24-3 describes the settings you can supply for *cursortype*, *lockoptions*, and *options*.

Table 24-3 RecordSetObject.Open Parameter Settings

Setting	Description
CURSORTYPE (SELECT ONE)	
adOpenForwardOnly	Returns a read-only snapshot cursor (recordset) that you can move forward through only once. You can use the MoveNext method to access successive rows. If you do not supply a CursorType setting, adOpenForwardOnly is the default.
adOpenKeyset	Returns a Keyset cursor. This is roughly analogous to a DAO dynaset. If you are using ADO to open a recordset against a source in an Access .accdb file, you should use this option to obtain a recordset that behaves most like a DAO recordset. In this type of cursor, you will see changes to rows made by other users, but you will not see new rows added by other users after you have opened the cursor.

Setting	Description
adOpenDynamic	Returns a dynamic cursor. This type of cursor lets you see not only changes made by other users, but also added rows. Note, however, that certain key properties you might depend on in a DAO recordset such as RecordCount might not exist or might always be zero.
adOpenStatic	Returns a read-only snapshot cursor. You won't be able to see changes made by other users after you've opened the cursor.
LOCKTYPE (SELECT ONE)	
adLockReadOnly	Provides no locks. The cursor is read-only. If you do not provide a lock setting, this is the default.
adLockPessimistic	Asks the target database to lock a row as soon as you place the row in an editable state by executing an Edit method.
adLockOptimistic	Asks the target database not to attempt to lock a row until you try to write it to the database with an Update method. This generates a run-time error in your code if another user has changed the row after you executed the Edit method. You should use this option when accessing rows in an Access .accdb file.
OPTIONS (YOU CAN COMBINE ONE CMD SETTING WITH ONE ASYNC SETTING WITH A PLUS SIGN)	
adCmdText	Indicates that *source* is an SQL statement.
adCmdTable	Indicates that *source* is a table name (or a query name in a desktop database). In DAO, this is analogous to opening a dynaset recordset on a table.
adCmdTableDirect	Indicates that *source* is a table name. This is analogous to a DAO dbOpenTable.
adCmdStoredProc	Indicates that *source* is a stored procedure. In DAO, this is analogous to opening a dynaset on a sorted query.
adAsyncFetch	After fetching the initial rows to populate the cursor, additional fetching occurs in the background. If you try to access a row that has not been fetched yet, your code will wait until the row is fetched.
adAsyncFetchNonBlocking	After fetching the initial rows to populate the cursor, additional fetching occurs in the background. If you try to access a row that has not been fetched yet, your code will receive an end-of-file indication.

For example, to declare a recordset for the tblFacilities table in the Housing Reservation database (Housing.accdb) and open the recordset as a table so you can use its indexes, enter the following:

```
Dim cnThisConnect As ADODB.Connection
Dim rcdFacilities As New ADODB.RecordSet
Dim rcdBooks As New ADODB.Recordset
Set cnThisConnect = CurrentProject.Connection
rcdFacilities.Index = "PrimaryKey"
rcdBooks.Open "tblFacilities", cnThisConnect, adOpenKeyset, _
    adLockOptimistic, adCmdTableDirect
```

Note that you must establish the index you want to use before you open the recordset. (If you want to try this in the Housing Reservation database, Housing.accdb, you'll need to add a reference to the Microsoft ActiveX Data Objects Library.)

To open the qryContactProducts query in the Conrad Systems Contacts database as a keyset, enter the following:

```
Dim cnThisConnect As ADODB.Connection
Dim rcdContactProducts As New ADODB.RecordSet
Set cnThisConnect = CurrentProject.Connection
rcdContactProducts.Open "qryContactProducts", _
  cnThisConnect, adOpenKeyset, adLockOptimistic, _
  adCmdTable
```

After you open a recordset, you can use one of the Move methods to move to a specific record. Use *recordset*.MoveFirst to move to the first row in the recordset. Other Move methods include MoveLast, MoveNext, and MovePrevious. If you want to search for a specific row in the recordset, use the Find method or set the recordset's Filter property. Unlike the Find methods in DAO, the Find method in ADO is limited to a single simple test on a column in the form "<*column-name*> <*comparison*> <*comparison-value*>". Note that to search for a Null value, you must say: "[SomeColumn] = Null", not "[SomeColumn] Is Null" as you would in DAO. Also, <*comparison*> can be only <, >, <=, >=, <>, =, or LIKE. Note that if you want to use the LIKE keyword, you can use either the ANSI wildcards "%" and "_" or the Access ACE/JET wildcards "*" and "?", but the wildcard can appear only at the end of the <*comparison-value*> string.

If you want to search for rows using a more complex filter, you must assign a string variable or an expression containing the criteria for finding the records you want to the Filter property of the recordset. This limits the rows in the recordset to only those that meet the filter criteria. The criteria string must be made of the simple comparisons that you can use with Find, but you can include multiple comparisons with the AND or OR Boolean operator.

For example, to find the first row in the qryContactProducts query's recordset whose SoldPrice field is greater than $200, enter the following:

```
rcdContactProducts.MoveFirst
rcdContactProducts.Find "SoldPrice > 200"
' EOF property will be true if nothing found
If Not rcdContactProducts.EOF Then
' Found a record!
```

To find all rows in qryContactProducts where the product was sold after November 1, 2010, and SoldPrice is greater than $200, enter the following:

```
rcdContactProducts.Filter = &
  "DateSold > #11/1/2010# AND SoldPrice > 200"
' EOF property will be true if filter produces no rows
If Not rcdODetails.EOF Then
' Found some rows!
```

To delete a row in a keyset, simply move to the row you want to delete and then use the Delete method. For example, to delete the first row in the qryContactProducts query's recordset that hasn't been invoiced yet (the Invoiced field is false), enter the following:

```
Dim cnThisConnect As ADODB.Connection
Dim rcdContactProducts As New ADODB.RecordSet
Set cnThisConnect = CurrentProject.Connection
rcdContactProducts.Open "qryContactProducts", _
  cnThisConnect, adOpenKeyset, adLockOptimistic, _
  adCmdTable
rcdContactProducts.MoveFirst
rcdContactProducts.Find "Invoiced = 0"
' Test the recordset EOF property for "not found"
If Not rcdContactProducts.EOF Then
    rcdContactProducts.Delete
End If
```

Note that in this example, if tblContactRelatedProducts includes related records, Access prevents the deletion. If you want to update rows in a recordset, move to the first row you want to update. You can refer to any of the updateable fields in the row by name to change their values. You can use the Update method on the recordset to explicitly save your changes before moving to another row. ADO automatically saves your changed row when you move to a new row. If you need to discard an update, you must use the CancelUpdate method of the recordset object.

For example, to increase by 10 percent the SoldPrice entry of the first row in the rcdContactProducts query's recordset whose SoldPrice value is greater than $200, enter the following:

```
Public Sub UpdateFirstSoldPrice10Percent()
Dim cnThisConnect As ADODB.Connection
Dim rcdContactProducts As New ADODB.RecordSet
Set cnThisConnect = CurrentProject.Connection
rcdContactProducts.Open "qryContactProducts", _
  cnThisConnect, adOpenKeyset, adLockOptimistic, _
  adCmdTable
rcdContactProducts.Filter = "SoldPrice > 200"
' Test the recordset EOF property for "not found"
If Not rcdContactProducts.EOF Then
  rcdContactProducts![SoldPrice] = _
    rcdContactProducts![SoldPrice] * 1.1
  rcdContactProducts.Update
  rcdContactProducts.MoveNext
End If
```

To insert a new row in a recordset, use the AddNew method to start a new row. Set the values of all required fields in the row and then use the Update method to save the new row. For example, to insert a new company in the Conrad Systems Contacts tblCompanies table, enter the following:

```
Dim cnThisConnect As ADODB.Connection
Dim rcdCompanies As New ADODB.RecordSet
Set cnThisConnect = CurrentProject.Connection
rcdCompanies.Open "tblCompanies", cnThisConnect, _
  adOpenKeyset, adLockOptimistic, adCmdTable
rcdCompanies.AddNew
rcdCompanies![CompanyName] = "Winthrop Brewing Co."
rcdCompanies![Address] = "155 Riverside Ave."
rcdCompanies![City] = "Winthrop"
rcdCompanies![StateOrProvince] = "WA"
rcdCompanies![PostalCode] = "98862"
rcdCompanies![PhoneNumber] = "(509) 555-8100"
rcdCompanies.Update
```

Other Uses for Object Methods

As you'll learn later in this chapter in more detail, you must use a method of the DoCmd object to execute the equivalent of most macro actions within Visual Basic. You must use the RunCommand method of either the Application or DoCmd object to execute commands you can find on any of the Access menus.

You can also define a public function or subroutine (see the next section) within the module associated with a Form or Report object and execute that procedure as a method of the form or report. If your public procedure is a function, you must assign the result of the execution of the method to a variable of the appropriate type. If the public procedure is a subroutine, you can execute the form or report object method as a Visual Basic statement. For more information about object methods, find the topic about the object of interest in Help, and then click the Methods hyperlink.

Functions and Subroutines

You can create two types of procedures in Visual Basic—*functions* and *subroutines*—which are also known as Function procedures and Sub procedures. (As you'll learn in "Understanding Class Modules," on page 1529, class modules also support a special type of function, Property Get, and special subroutines, Property Let and Property Set, that let you manage properties of the class.) Each type of procedure can accept *parameters*—data variables that you pass to the procedure that can determine how the procedure operates. Functions can return a single data value, but subroutines cannot. In addition, you can execute a public function from anywhere in Access, including from expressions in queries and from macros. You can execute a subroutine only from a function, from another subroutine, or as an event procedure in a form or a report.

Function Statement

Use a Function statement to declare a new function, the parameters it accepts, the variable type it returns, and the code that performs the function procedure.

Syntax

```
[Public | Private | Friend] [Static] Function functionname
  ([<arguments>]) [As datatype]
    [<function statements>]
    [functionname = <expression>]
    [Exit Function]
    [<function statements>]
    [functionname = <expression>]
End Function
```

where <*arguments*> is

```
{[Optional][ByVal | ByRef][ParamArray] argumentname[()]
  [As datatype][= default]},...
```

Notes

Use the Public keyword to make this function available to all other procedures in all modules. Use the Private keyword to make this function available only to other procedures in the same module. When you declare a function as private in a module, you cannot call that function from a query or a macro or from a function in another module. Use the Friend keyword in a class module to declare a function that is public to all other code in your application but is not visible to outside code that activates your project via automation.

Include the Static keyword to preserve the value of all variables declared within the procedure, whether explicitly or implicitly, so long as the module containing the procedure is open. This is the same as using the Static statement (discussed earlier in this chapter) to explicitly declare all variables created in this function.

You can use a type declaration character at the end of the *functionname* entry or use the As *datatype* clause to declare the data type returned by this function. Valid *datatype* entries are Byte, Boolean, Integer, Long, Currency, Single, Double, Date, String (for variable-length strings), String * *length* (for fixed-length strings), Object, Variant, or one of the object types described earlier in this chapter. If you do not declare a data type, Visual Basic assumes that the function returns a variant result. You can set the return value in code by assigning an expression of a compatible data type to the function name.

You should declare the data type of any arguments in the function's parameter list. Note that the names of the variables passed by the calling procedure can be different from the names of the variables known by this procedure. If you use the ByVal keyword to declare an argument, Visual Basic passes a copy of the argument to your function. Any change you make to a ByVal argument does not change the original variable in the calling procedure. If you use the ByRef keyword, Visual Basic passes the actual memory address of the variable, allowing the procedure to change the variable's value in the calling procedure. (If the argument passed by the calling procedure is an expression, Visual Basic treats it as if you had declared it by using ByVal.) Visual Basic always passes arrays by reference (using ByRef).

Use the Optional keyword to declare an argument that isn't required. All optional arguments must be the Variant data type. If you declare an optional argument, all arguments that follow in the argument list must also be declared as optional. You can specify a *default* value only for optional parameters. Use the IsMissing built-in function to test for the absence of optional parameters. You can also use the ParamArray argument to declare an array of optional elements of the Variant data type. When you call the function, you can then pass it an arbitrary number of arguments. The ParamArray argument must be the last argument in the argument list.

Use the Exit Function statement anywhere in your function to clear any error conditions and exit your function normally, returning to the calling procedure. If Visual Basic runs your

code until it encounters the End Function statement, control is passed to the calling procedure, but any errors are not cleared. If this function causes an error and terminates with the End Function statement, Visual Basic passes the error to the calling procedure. See "Trapping Errors," on page 1551, for details.

Example

To create a function named MyFunction that accepts an integer argument and a string argument and returns a double value, enter the following:

```
Function MyFunction (intArg1 As Integer, strArg2 As _
  String) As Double
    If strArg2 = "Square" Then
      MyFunction = intArg1 * intArg1
    Else
      MyFunction = Sqr(intArg1)
    End If
End Function
```

Sub Statement

Use a Sub statement to declare a new subroutine, the parameters it accepts, and the code in the subroutine.

Syntax

```
[Public | Private | Friend] [Static] Sub subroutinename
  ([<arguments>])
    [ <subroutine statements> ]
    [Exit Sub]
    [ <subroutine statements> ]
End Sub
```

where <*arguments*> is

```
{[Optional][ByVal | ByRef][ParamArray]
   argumentname[()] [As datatype][ = default]},...
```

Notes

Use the Public keyword to make this subroutine available to all other procedures in all modules. Use the Private keyword to make this procedure available only to other procedures in the same module. When you declare a sub as private in a module, you cannot call that sub from a function or sub in another module. Use the Friend keyword in a class module to declare a sub that is public to all other code in your application but is not visible to outside code that activates your project via automation.

Include the Static keyword to preserve the value of all variables declared within the procedure, whether explicitly or implicitly, so long as the module containing the procedure is open. This is the same as using the Static statement (discussed earlier in this chapter) to explicitly declare all variables created in this subroutine.

You should declare the data type of all arguments that the subroutine accepts in its argument list. Valid *datatype* entries are Byte, Boolean, Integer, Long, Currency, Single, Double, Date, String (for variable-length strings), String * *length* (for fixed-length strings), Object, Variant, or one of the object types described earlier in this chapter. Note that the names of the variables passed by the calling procedure can be different from the names of the variables as known by this procedure. If you use the ByVal keyword to declare an argument, Visual Basic passes a copy of the argument to your subroutine. Any change you make to a ByVal argument does not change the original variable in the calling procedure. If you use the ByRef keyword, Visual Basic passes the actual memory address of the variable, allowing the procedure to change the variable's value in the calling procedure. (If the argument passed by the calling procedure is an expression, Visual Basic treats it as if you had declared it by using ByVal.) Visual Basic always passes arrays by reference (using ByRef).

Use the Optional keyword to declare an argument that isn't required. All optional arguments must be the Variant data type. If you declare an optional argument, all arguments that follow in the argument list must also be declared optional. You can specify a *default* value only for optional parameters. Use the IsMissing built-in function to test for the absence of optional parameters. You can also use the ParamArray argument to declare an array of optional elements of the Variant data type. When you call the subroutine, you can then pass it an arbitrary number of arguments. The ParamArray argument must be the last argument in the argument list.

Use the Exit Sub statement anywhere in your subroutine to clear any error conditions and exit your subroutine normally, returning to the calling procedure. If Visual Basic runs your code until it encounters the End Sub statement, control is passed to the calling procedure but any errors are not cleared. If this subroutine causes an error and terminates with the End Sub statement, Visual Basic passes the error to the calling procedure. See "Trapping Errors," on page 1551, for details.

Example

To create a subroutine named MySub that accepts two string arguments but can modify only the second argument, enter the following:

```
Sub MySub (ByVal strArg1 As String, ByRef strArg2 _
  As String)
    <subroutine statements>
End Sub
```

Understanding Class Modules

Whenever you create event procedures behind a form or report, you're creating a class module. A *class module* is the specification for a user-defined object in your database, and the code you write in the module defines the methods and properties of the object. Of course, forms and reports already have dozens of methods and properties already defined by Access, but you can create extended properties and methods when you write code in the class module attached to a form or report.

You can also create a class module as an independent object by clicking the Class Module button in the Macros & Code group on the Create tab or by clicking Class Module on the Insert menu in the VBE. In the Conrad Systems Contacts sample database (Contacts.accdb), you can find a class module called ComDlg that provides a simple way to call the Open File dialog box in Windows from your Visual Basic code.

As previously discussed, you define a method in a class module by declaring a procedure (either a function or a sub) public. When you create an active instance of the object defined by the class module, either by opening it or by setting it to an object variable, you can execute the public functions or subs you have defined by referencing the function or sub name as a method of the object. For example, when the frmContacts form is open, you can execute the cmdCancel_Click sub by referencing it as a method of the form's class. (The cmdCancel_Click sub is public in all forms in the sample database so that the Exit button on the main switchboard can use it to command the form to clear edits and close itself.) The name of any form's class is in the form *Form_formname*, so you execute this method in your code like this:

```
Form_frmContacts.cmdCancel_Click
```

When you create a class module that you see in the Modules list in the Navigation pane, you can create a special sub that Visual Basic runs whenever code in your application creates a new instance of the object defined by your class. For example, you can create a private Class_Initialize sub to run code that sets up your object whenever other code in your application creates a new instance of your class object. You might use this event to open recordsets or initialize variables required by the object. You can also create a private Class_Terminate sub to run code that cleans up any variables or objects (perhaps closing open recordsets) when your object goes out of scope or the code that created an instance of your object sets it to Nothing. (Your object goes out of scope if a procedure activates your class by setting it to a nonstatic local object variable and then the procedure exits.)

Although you can define properties of a class by declaring public variables in the Declarations section of the class module, you can also define specific procedures to handle fetching and setting properties. When you do this, you can write special processing code that runs whenever a caller fetches or sets one of the properties defined by these procedures. To create special property processing procedures in a class module, you need to write Property Get, Property Let, and Property Set procedures as described in the following sections.

Property Get

Use a Property Get procedure to return a property value for the object defined by your class module. When other code in your application attempts to fetch the value of this property of your object, Visual Basic executes your Property Get procedure to return the value. Your code can return a data value or an object.

Syntax

```
[Public | Private | Friend] [Static] Property Get propertyname
  ([<arguments>]) [As datatype]
    [<property statements>]
    [propertyname = <expression>]
    [Exit Property]
    [<property statements>]
    [propertyname = <expression>]
End Property
```

where <*arguments*> is

```
{[Optional][ByVal | ByRef][ParamArray] argumentname[()]
  [As datatype][= default]},...
```

Notes

Use the Public keyword to make this property available to all other procedures in all modules. Use the Private keyword to make this property available only to other procedures in the same module. When you declare a property as private in a class module, you cannot reference that property from another module. Use the Friend keyword to declare a property that is public to all other code in your application but is not visible to outside code that activates your project via automation.

Include the Static keyword to preserve the value of all variables declared within the property procedure, whether explicitly or implicitly, so long as the module containing the procedure is open. This is the same as using the Static statement (discussed earlier in this chapter) to explicitly declare all variables created in this property procedure.

You can use a type declaration character at the end of the *propertyname* entry or use the As *datatype* clause to declare the data type returned by this property. Valid *datatype* entries are Byte, Boolean, Integer, Long, Currency, Single, Double, Date, String (for variable-length strings), String * *length* (for fixed-length strings), Object, Variant, or one of the object types described earlier in this chapter. If you do not declare a data type, Visual Basic assumes that the property returns a variant result. The data type of the returned value must match the data type of the *propvalue* variable you declare in any companion Property Let or Property Set procedure. You can set the return value in code by assigning an expression of a compatible data type to the property name.

You should declare the data type of all arguments in the property procedure's parameter list. Note that the names of the variables passed by the calling procedure can be different from the names of the variables known by this procedure. If you use the ByVal keyword to declare an argument, Visual Basic passes a copy of the argument to your procedure. Any change you make to a ByVal argument does not change the original variable in the calling procedure. If you use the ByRef keyword, Visual Basic passes the actual memory address of the variable, allowing the procedure to change the variable's value in the calling procedure. (If the argument passed by the calling procedure is an expression, Visual Basic treats it as if you had declared it by using ByVal.) Visual Basic always passes arrays by reference (using ByRef).

Use the Optional keyword to declare an argument that isn't required. All optional arguments must be the Variant data type. If you declare an optional argument, all arguments that follow in the argument list must also be declared as optional. You can specify a *default* value only for optional parameters. Use the IsMissing built-in function to test for the absence of optional parameters. You can also use the ParamArray argument to declare an array of optional elements of the Variant data type. When you attempt to access this property in an object set to the class, you can then pass it an arbitrary number of arguments. The ParamArray argument must be the last argument in the argument list.

Use the Exit Property statement anywhere in your property procedure to clear any error conditions and exit your procedure normally, returning to the calling procedure. If Visual Basic runs your code until it encounters the End Property statement, control is passed to the calling procedure but any errors are not cleared. If this procedure causes an error and terminates with the End Property statement, Visual Basic passes the error to the calling procedure. See "Trapping Errors," on page 1551, for details.

Examples

To declare a Filename property as a string and return it from a variable defined in the Declarations section of your class module, enter the following:

```
Option Explicit
Dim strFileName As String
Property Get Filename() As String
    ' Return the saved file name as a property
    Filename = strFilename
End Property
```

You can see an example of Property Get in the ComDlg class where we added similar code.

To establish a new instance of the object defined by the ComDlg class module and then fetch its Filename property, enter the following in a function or sub:

```
Dim clsDialog As New ComDlg, strFile As String
  With clsDialog
    ' Set the title of the dialog box
    .DialogTitle = "Locate Conrad Systems Contacts Data File"
    ' Set the default file name
    .FileName = "ContactsData.accdb"
    ' .. and start directory
    .Directory = CurrentProject.Path
    ' .. and file extension
    .Extension = "accdb"
    ' .. but show all accdb files just in case
    .Filter = "Conrad Systems File (*.accdb)|*.accdb"
    ' Default directory is where this file is located
    .Directory = CurrentProject.Path
    ' Tell the common dialog that the file and path must exist
    .ExistFlags = FileMustExist + PathMustExist
    ' If the ShowOpen method returns True
    If .ShowOpen Then
      ' Then fetch the Filename property
      strFile = .FileName
    Else
      Err.Raise 3999
    End If
  End With
```

Property Let

Use a Property Let procedure to define code that executes when the calling code attempts to assign a value to a data property of the object defined by your class module. You cannot define both a Property Let and a Property Set procedure for the same property.

Syntax

```
[Public | Private | Friend] [Static] Property Let propertyname
  ([<arguments>,] propvalue [As datatype])
    [ <property statements> ]
    [Exit Property]
    [ <property statements> ]
End Property
```

where <*arguments*> is

```
{[Optional][ByVal | ByRef][ParamArray]
   argumentname[()] [As datatype][ = default]},...
```

Notes

Use the Public keyword to make this property available to all other procedures in all modules. Use the Private keyword to make this property available only to other procedures in the same module. When you declare a property as private in a class module, you cannot reference the property from another module. Use the Friend keyword to declare a property that is public to all other code in your application but is not visible to outside code that activates your project via automation.

Include the Static keyword to preserve the value of all variables declared within the property procedure, whether explicitly or implicitly, so long as the module containing the procedure is open. This is the same as using the Static statement (discussed earlier in this chapter) to explicitly declare all variables created in this property procedure.

You should declare the data type of all arguments in the property procedure's parameter list. Valid *datatype* entries are Byte, Boolean, Integer, Long, Currency, Single, Double, Date, String (for variable-length strings), String * *length* (for fixed-length strings), Object, Variant, or one of the object types described earlier in this chapter. Note that the names of the variables passed by the calling procedure can be different from the names of the variables as known by this procedure. Also, the names and data types of the arguments must exactly match the arguments declared for the companion Property Get procedure. If you use the ByVal keyword to declare an argument, Visual Basic passes a copy of the argument to your property procedure. Any change you make to a ByVal argument does not change the original variable in the calling procedure. If you use the ByRef keyword, Visual Basic passes the actual memory address of the variable, allowing the procedure to change the variable's value in the calling procedure. (If the argument passed by the calling procedure is an expression, Visual Basic treats it as if you had declared it by using ByVal.) Visual Basic always passes arrays by reference (using ByRef).

Use the Optional keyword to declare an argument that isn't required. All optional arguments must be the Variant data type. If you declare an optional argument, all arguments that follow in the argument list must also be declared as optional. You can specify a *default* value only for optional parameters. Use the IsMissing built-in function to test for the absence of optional parameters. You can also use the ParamArray argument to declare an array of optional elements of the Variant data type. When you attempt to assign a value to this property in an object set to the class, you can then pass it an arbitrary number of arguments. The ParamArray argument must be the last argument in the argument list.

You must always declare at least one parameter, *propvalue*, to be the variable that contains the value that the calling code wants to assign to your property. This is the value or expression that appears on the right side of the assignment statement executed in the calling code. If you declare a data type, it must match the data type declared by the companion Property Get procedure. Also, when you declare a data type, the caller receives a data type mismatch error if the assignment statement attempts to pass an incorrect data type. You cannot modify this value, but you can evaluate it and save it as a value to be returned later by your Property Get procedure.

Use the Exit Property statement anywhere in your property procedure to clear any error conditions and exit your procedure normally, returning to the calling procedure. If Visual Basic runs your code until it encounters the End Property statement, control is passed to the calling procedure, but errors are not cleared. If this procedure causes an error and terminates with the End Property statement, Visual Basic passes the error to the calling procedure. See "Trapping Errors," on page 1551, for details.

Examples

To declare a FileName property, accept a value from a caller, and save the value in a variable defined in the Declarations section of your class module, enter the following:

```
Option Explicit
Dim strFileName As String
Property Let FileName(strFile)
  If Len(strFile) <= 64 Then _
    strFileName = strFile
End Property
```

You can see an example of Property Let in the ComDlg class, where we added similar code.

To establish a new instance of the object defined by the ComDlg class module and then set its Filename property, enter the following:

```
Dim clsDialog As New ComDlg, strFile As String
  With clsDialog
    ' Set the title of the dialog
    .DialogTitle = "Locate Conrad Systems Contacts Data File"
    ' Set the default file name
    .FileName = "ContactsData.accdb"
  End With
```

Property Set

Use a Property Set procedure to define code that executes when the calling code attempts to assign an object to an object property of the object defined by your class module. You cannot define both a Property Let and a Property Set procedure for the same property.

Syntax

```
[Public | Private | Friend] [Static] Property Set propertyname
  ([<arguments>,] object [As objecttype])
    [ <property statements> ]
    [Exit Property]
    [ <property statements> ]
End Property
```

where <*arguments*> is

```
{[Optional][ByVal | ByRef][ParamArray]
   argumentname[()] [As datatype][ = default]},...
```

Notes

Use the Public keyword to make this property available to all other procedures in all modules. Use the Private keyword to make this property available only to other procedures in the same module. When you declare a property as private in a class module, you cannot reference the property from another module. Use the Friend keyword to declare a property that is public to all other code in your application but is not visible to outside code that activates your project via automation.

Include the Static keyword to preserve the value of all variables declared within the property procedure, whether explicitly or implicitly, so long as the module containing the procedure is open. This is the same as using the Static statement (discussed earlier in this chapter) to explicitly declare all variables created in this property procedure.

You should declare the data type of all arguments in the property procedure's parameter list. Valid *datatype* entries are Byte, Boolean, Integer, Long, Currency, Single, Double, Date, String (for variable-length strings), String * *length* (for fixed-length strings), Object, Variant, or one of the object types described earlier in this chapter. Note that the names of the variables passed by the calling procedure can be different from the names of the variables as known by this procedure. Also, the names and data types of the arguments must exactly match the arguments declared for the companion Property Get procedure. If you use the ByVal keyword to declare an argument, Visual Basic passes a copy of the argument to your property procedure. Any change you make to a ByVal argument does not change the original variable in the calling procedure. If you use the ByRef keyword, Visual Basic passes the actual memory address of the variable, allowing the procedure to change the variable's value in the calling procedure. (If the argument passed by the calling procedure is an expression, Visual Basic treats it as if you had declared it by using ByVal.) Visual Basic always passes arrays by reference (using ByRef).

Use the Optional keyword to declare an argument that isn't required. All optional arguments must be the Variant data type. If you declare an optional argument, all arguments that follow in the argument list must also be declared as optional. You can specify a *default* value only for optional parameters. Use the IsMissing built-in function to test for the absence of optional parameters. You can also use the ParamArray argument to declare an array of optional elements of the Variant data type. When you attempt to assign a value to this property in an object set to the class, you can then pass it an arbitrary number of arguments. The ParamArray argument must be the last argument in the argument list.

You must always declare at least one parameter, object, to be the variable that contains the object that the calling code wants to assign to your property. This is the object reference that appears on the right side of the assignment statement executed in the calling code. If you include an *objecttype* entry, it must match the object type declared by the companion Property Get procedure. Also, when you declare an object type, the caller receives a data type mismatch error if the assignment statement attempts to pass an incorrect object type. You can evaluate the properties of this object, set its properties, execute its methods, and save the object pointer in another variable that your Property Get procedure can later return.

Use the Exit Property statement anywhere in your property procedure to clear any error conditions and exit your procedure normally, returning to the calling procedure. If Visual Basic runs your code until it encounters the End Property statement, control is passed to the calling procedure, but errors are not cleared. If this procedure causes an error and terminates with the End Property statement, Visual Basic passes the error to the calling procedure. See "Trapping Errors," on page 1551, for details.

Examples

To declare a ControlToUpdate property, accept a value from a caller, and save the value in an object variable defined in the Declarations section of your class module, enter the following:

```
Option Explicit
Dim ctlToUpdate As Control
Property Set ControlToUpdate(ctl As Control)
  ' Verify we have the right type of control
  Select Case ctl.ControlType
    ' Text box, combo box, and list box are OK
    Case acTextBox, acListBox, acComboBox
      ' Save the control object
      Set ctlToUpdate = ctl
    Case Else
      Err.Raise 3999
  End Select
End Property
```

Here is an example of how to establish a new instance of an object defined by a class module and then set its FileName property:

```
Public Property Set ctlToUpdate(Optional intD As Integer = 0, ctl As control)
' This procedure is an example of Property Set
' GetDateOCX opens this form by creating a new instance of the class
'  and then sets the required properties via a SET statement.
    ' First, validate the kind of control passed
    Select Case ctl.ControlType
        ' Text box, combo box, and list box are OK
        Case acTextBox, acListBox, acComboBox
        Case Else
            MsgBox "Invalid control passed to the Calendar."
            DoCmd.Close acForm, Me.Name
    End Select
    ' Save the pointer to the control to update
    Set ctlThisControl = ctl
    ' Save the date only value
    intDateOnly = intD
    ' If "date only"
    If (intDateOnly = -1) Then
        ' Resize my window
        DoCmd.MoveSize , , , 3935
        ' Hide some stuff just to be sure
        Me.txtHour.Visible = False
        Me.txtMinute.Visible = False
        Me.lblColon.Visible = False
        Me.lblTimeInstruct.Visible = False
        Me.SetFocus
    End If
    ' Set the flag to indicate we got the pointer
```

```
    intSet = True
    ' Save the "current" value of the control
    varDate = ctlThisControl.Value
    ' Make sure we got a valid date value
    If Not IsDate(varDate) Then
        ' If not, set the default to today
        varDate = Now
        Me.Calendar1.Value = Date
        Me.txtHour = Format(Hour(varDate), "00")
        Me.txtMinute = Format(Minute(varDate), "00")
    Else
        ' Otherwise, set the date/time to the one in the control
        ' Make sure we have a Date data type, not just text
        varDate = CDate(varDate)
        Me.Calendar1.Value = varDate
        Me.txtHour = Format(Hour(varDate), "00")
        Me.txtMinute = Format(Minute(varDate), "00")
    End If
End Property
```

Controlling the Flow of Statements

Visual Basic provides many ways for you to control the flow of statements in procedures. You can call other procedures, loop through a set of statements either a calculated number of times or based on a condition, or test values and conditionally execute sets of statements based on the result of the condition test. You can also go directly to a set of statements or exit a procedure at any time. The following sections demonstrate some of (but not all) the ways that you can control flow in your procedures.

Call Statement

Use a Call statement to transfer control to a subroutine.

Syntax

```
Call subroutinename [(<arguments>)]
or
subroutinename [<arguments>]
```

where <*arguments*> is

```
{[ByVal | ByRef] <expression> },...
```

Notes

The Call keyword is optional, but if you omit it, you must also omit the parentheses surrounding the parameter list. If the subroutine accepts arguments, the names of the variables passed by the calling procedure can be different from the names of the variables as known by the subroutine. You can use the ByVal and ByRef keywords in a Call statement only when you're making a call to a dynamic link library (DLL) procedure. Use ByVal for string arguments to indicate that you need to pass a pointer to the string rather than pass the string directly. Use ByRef for nonstring arguments to pass the value directly. If you use the ByVal keyword to declare an argument, Visual Basic passes a copy of the argument to the subroutine. The subroutine cannot change the original variable in the calling procedure. If you use the ByRef keyword, Visual Basic passes the actual memory address of the variable, allowing the procedure to change the variable's value in the calling procedure. (If the argument passed by the calling procedure is an expression, Visual Basic treats it as if you had declared it by using ByVal.)

Examples

To call a subroutine named MySub and pass it an integer variable and an expression, enter the following:

```
Call MySub (intMyInteger, curPrice * intQty)
```

An alternative syntax is

```
MySub intMyInteger, curPrice * intQty
```

Do...Loop Statement

Use a Do...Loop statement to define a block of statements that you want executed multiple times. You can also define a condition that terminates the loop when the condition is false.

Syntax

```
Do [{While | Until} <condition>]
    [<procedure statements>]
    [Exit Do]
    [<procedure statements>]
Loop
```

or

```
Do
    [<procedure statements>]
    [Exit Do]
    [<procedure statements>]
Loop [{While | Until} <condition>]
```

Notes

The <*condition*> is a comparison predicate or expression that Visual Basic can evaluate to True (nonzero) or False (zero or Null). The While clause is the opposite of the Until clause. If you specify a While clause, execution continues so long as <*condition*> is true. If you specify an Until clause, execution of the loop stops when <*condition*> becomes true. If you place a While or an Until clause in the Do clause, the condition must be met for the statements in the loop to execute at all. If you place a While or an Until clause in the Loop clause, Visual Basic executes the statements within the loop before testing the condition.

You can place one or more Exit Do statements anywhere within the loop to exit the loop before reaching the Loop statement. Generally you'll use the Exit Do statement as part of some other evaluation statement structure, such as an If...Then...Else statement.

Example

To read all the rows in the tblCompanies table until you reach the end of the recordset (that is, the EOF property is true), enter the following:

```
Dim dbContacts As DAO.Database
Dim rcdCompanies As DAO.RecordSet
Set dbContacts = CurrentDb
Set rcdCompanies = dbContacts.OpenRecordSet("tblCompanies")
Do Until rcdCompanies.EOF
    <procedure statements>
    rcdCompanies.MoveNext
Loop
```

For...Next Statement

Use a For...Next statement to execute a series of statements a specific number of times.

Syntax

```
For counter = first To last [Step stepamount]
    [<procedure statements>]
    [Exit For]
    [<procedure statements>]
Next [counter]
```

Notes

The *counter* must be a numeric variable that is not an array or a record element. Visual Basic initially sets the value of *counter* to *first*. If you do not specify a stepamount, the default stepamount value is +1. If the *stepamount* value is positive or 0, Visual Basic executes the loop so long as *counter* is less than or equal to *last*. If the *stepamount* value is negative, Visual Basic executes the loop so long as counter is greater than or equal to *last*. Visual Basic adds *stepamount* to *counter* when it encounters the corresponding Next statement. You can change the value of *counter* within the For loop, but this might make your procedure more difficult to test and debug. Changing the value of *last* within the loop does not affect execution of the loop. You can place one or more Exit For statements anywhere within the loop to exit the loop before reaching the Next statement. Generally you'll use the Exit For statement as part of some other evaluation statement structure, such as an If...Then...Else statement.

You can nest one For loop inside another. When you do, you must choose a different *counter* name for each loop.

Example

To list in the Immediate window the names of the first five queries in the Conrad Systems Contacts database, enter the following in a function or sub:

```
Dim dbContacts As DAO.Database
Dim intI As Integer
Set dbContacts = CurrentDb
For intI = 0 To 4
    Debug.Print dbContacts.QueryDefs(intI).Name
Next intI
```

For Each...Next Statement

Use a For Each...Next statement to execute a series of statements for each item in a collection or an array.

Syntax

```
For Each item In group
    [<procedure statements>]
    [Exit For]
    [<procedure statements>]
Next [item]
```

Notes

The *item* must be a variable that represents an object in a collection or an element of an array. The *group* must be the name of a collection or an array. Visual Basic executes the loop so long as at least one item remains in the collection or the array. All the statements in the loop are executed for each item in the collection or the array. You can place one or more Exit For statements anywhere within the loop to exit the loop before reaching the Next statement. Generally you'll use the Exit For statement as part of some other evaluation statement structure, such as an If...Then...Else statement.

You can nest one For Each loop inside another. When you do, you must choose a different *item* name for each loop.

Example

To list in the Immediate window the names of all the queries in the Conrad Systems Contacts database, enter the following in a function or sub:

```
Dim dbContacts As DAO.Database
Dim qdf As DAO.QueryDef
Set dbContacts = CurrentDb
For Each qdf In dbContacts.QueryDefs
    Debug.Print qdf.Name
Next qdf
```

CAUTION!

If you execute code within the For Each loop that modifies the members of the group, then you might not process all the members. For example, if you attempt to close all open forms using the following code, you will skip some open forms because you are eliminating members from the group (the Forms collection) inside the loop:

```
Dim frm As Form
For Each frm In Forms
  DoCmd.Close acForm, frm.Name
Next frm
```

The correct way to close all open forms is as follows:

```
Dim intI As Integer
For intI = Forms.Count - 1 To 0 Step - 1
  DoCmd.Close acForm, Forms(intI).Name
Next intI
```

GoTo Statement

Use a GoTo statement to jump unconditionally to another statement in your procedure.

Syntax

```
GoTo {label | linenumber}
```

Notes

You can label a statement line by starting the line with a string of no more than 40 characters that starts with an alphabetic character and ends with a colon (:). A line label cannot be a Visual Basic or Access reserved word. If you want, you can also number the statement lines in your procedure. Each line number must contain only numbers, must be different from all other line numbers in the procedure, must be the first nonblank characters in a line, and must contain 40 characters or less. To jump to a line number or a labeled line, use the GoTo statement and the appropriate *label* or *linenumber*.

Example

To jump to the statement line labeled SkipOver, enter the following:

```
GoTo SkipOver
```

If...Then...Else Statement

Use an If...Then...Else statement to conditionally execute statements based on the evaluation of a condition.

Syntax

```
If <condition1> Then
    [<procedure statements 1>]
[ElseIf <condition2> Then
    [<procedure statements 2>]]...
[Else
    [<procedure statements n>]]
End If
```

or

```
If <condition> Then <thenstmt> [Else <elsestmt>]
```

Notes

Each condition is a numeric or string expression that Visual Basic can evaluate to True (non-zero) or False (0 or Null). A condition can also consist of multiple comparison expressions and Boolean operators. In addition, a condition can also be the special TypeOf...Is test to evaluate a control variable. The syntax for this test is

TypeOf *<Object>* **Is** *<ObjectType>*

where <*Object*> is the name of an object variable and <*ObjectType*> is the name of any valid object type recognized in Access. A common use of this syntax is to loop through all the controls in a form or report Controls collection and take some action if the control is of a specific type (for example, change the FontWeight property of all labels to bold). Valid control types are Attachment, BoundObjectFrame, CheckBox, ComboBox, CommandButton, CustomControl, Image, Label, Line, ListBox, ObjectFrame, OptionButton, OptionGroup, PageBreak, Rectangle, Subform, TabControl, TextBox, and ToggleButton.

If the condition is true, Visual Basic executes the statement or statements immediately following the Then keyword. If the condition is false, Visual Basic evaluates the next ElseIf condition or executes the statements following the Else keyword, whichever occurs next.

The alternative syntax does not need an End If statement, but you must enter the entire If...Then statement on a single line. Both <*thenstmt*> and <*elsestmt*> can be either a single Visual Basic statement or multiple statements separated by colons (:).

Example

To set an integer value depending on whether a string begins with a letter from *A* through *F*, from *G* through *N*, or from *O* through *Z*, enter the following:

```
Dim strMyString As String, strFirst As String, _
  intVal As Integer
' Grab the first letter and make it upper case
strFirst = UCase(Left(strMyString, 1))
If strFirst >= "A" And strFirst <= "F" Then
    intVal = 1
ElseIf strFirst >= "G" And strFirst <= "N" Then
    intVal = 2
ElseIf strFirst >= "O" And strFirst <= "Z" Then
    intVal = 3
Else
    intVal = 0
End If
```

RaiseEvent Statement

Use the RaiseEvent statement to signal a declared event in a class module.

Syntax

RaiseEvent *eventname* [(*<arguments>*)]

where *<arguments>* is

{ *<expression>* },...

Notes

You must always declare an event in the class module that raises the event. You cannot use RaiseEvent to signal a built-in event (such as Current) of a form or report class module. If an event passes no arguments, you must not include an empty pair of parentheses when you code the RaiseEvent statement. An event can be received only by another module that has declared an object variable using WithEvents that has been set to the class module or object containing this class.

See the WeddingList.accdb sample database—described in Chapter 25, "Automating Your Application with Visual Basic," on the companion CD—for an example using RaiseEvent to synchronize two forms.

Example

To define an event named Signal that returns a text string and then to signal that event in a class module, enter the following:

```
Option Explicit
Public Event Signal(ByVal strMsg As String)

Public Sub RaiseSignal(ByVal strText As String)
    RaiseEvent Signal(strText)
End Sub
```

Select Case Statement

Use a Select Case statement to execute statements conditionally based on the evaluation of an expression that is compared to a list or range of values.

Syntax

```
Select Case <test expression>
    [Case <comparison list 1>
        [<procedure statements 1>]]
    ...
    [Case Else
        [<procedure statements n>]]
End Select
```

where <*test expression*> is any numeric or string expression; where <*comparison list*> is

```
{<comparison element>,...}
```

where <*comparison element*> is

```
{expression | expression To expression |
  Is <comparison operator> expression}
```

and where <*comparison operator*> is

```
{= | <> | < | > | <= | >=}
```

Notes

If the <*test expression*> matches a <*comparison element*> in a Case clause, Visual Basic executes the statements that follow that clause. If the <*comparison element*> is a single expression, the <*test expression*> must equal the <*comparison element*> for the statements following that clause to execute. If the <*comparison element*> contains a To keyword, the first expression must be less than the second expression (either in numeric value if the expressions are numbers or in collating sequence if the expressions are strings) and the <*test expression*> must be between the first expression and the second expression. If the <*comparison element*> contains the Is keyword, the evaluation of <*comparison operator*> expression must be true.

If more than one Case clause matches the <*test expression*>, Visual Basic executes only the set of statements following the first Case clause that matches. You can include a block of statements following a Case Else clause that Visual Basic executes if none of the previous Case clauses matches the <*test expression*>. You can nest another Select Case statement within the statements following a Case clause.

Example

To assign an integer value to a variable, depending on whether a string begins with a letter from *A* through *F*, from *G* through *N*, or from *O* through *Z*, enter the following:

```
Dim strMyString As String, intVal As Integer
Select Case UCase$(Mid$(strMyString, 1, 1))
    Case "A" To "F"
        intVal = 1
    Case "G" To "N"
        intVal = 2
    Case "O" To "Z"
        intVal = 3
    Case Else
        intVal = 0
End Select
```

Stop Statement

Use a Stop statement to suspend execution of your procedure.

Syntax

```
Stop
```

Notes

A Stop statement has the same effect as setting a breakpoint on a statement. You can use the Visual Basic debugging tools, such as the Step Into and the Step Over buttons and the Debug window, to evaluate the status of your procedure after Visual Basic halts on a Stop statement. You should not use the Stop statement in a production application.

While...Wend Statement

Use a While...Wend statement to continuously execute a block of statements so long as a condition is true.

Syntax

```
While <condition>
    [<procedure statements>]
Wend
```

Notes

A While...Wend statement is similar to a Do...Loop statement with a While clause, except that you can use an Exit Do statement to exit from a Do loop. Visual Basic provides no similar Exit clause for a While loop. The <*condition*> is an expression that Visual Basic can evaluate to True (nonzero) or False (0 or Null). Execution continues so long as the <*condition*> is true.

Example

To read all the rows in the tblCompanies table until you reach the end of the recordset, enter the following in a function or sub:

```
Dim dbContacts As DAO.Database
Dim rcdCompanies As DAO.RecordSet
Set dbContacts = CurrentDb
Set rcdCompanies = dbContacts.OpenRecordSet("tblCompanies")
While Not rcdCompanies.EOF
    <procedure statements>
    rcdCompanies.MoveNext
Wend
```

With...End Statement

Use a With statement to simplify references to complex objects in code. You can establish a base object using a With statement and then use a shorthand notation to refer to objects, collections, properties, or methods on that object until you terminate the With statement. When you plan to reference an object many times within a block of code, using With also improves execution speed.

Syntax

```
With <object reference>
    [<procedure statements>]
End With
```

Example

To use shorthand notation on a recordset object to add a new row to a table, enter the following:

```
Dim rcd As DAO.Recordset, db As DAO.Database
Set db = CurrentDb
Set rcd = db.OpenRecordset("MyTable", _
  dbOpenDynaset, dbAppendOnly)
With rcd
```

```
    ' Start a new record
    .Addnew
    ' Set the field values
    ![FieldOne] = "1"
    ![FieldTwo] = "John"
    ![FieldThree] = "Viescas"
    .Update
    .Close
End With
```

To write the same code without the With, you would have to say:

```
Dim rcd As DAO.Recordset, db As DAO.Database
Set db = CurrentDb
Set rcd = db.OpenRecordset("MyTable", _
  dbOpenDynaset, dbAppendOnly)
    ' Start a new record
    rcd.Addnew
    ' Set the field values
    rcd![FieldOne] = "1"
    rcd![FieldTwo] = "John"
    rcd![FieldThree] = "Viescas"
    rcd.Update
    rcd.Close
```

Running Macro Actions and Menu Commands

From within Visual Basic, you can execute most of the macro actions that Access provides and any of the built-in menu commands. Only a few of the macro actions have direct Visual Basic equivalents. To execute a macro action or menu command, use the methods of the DoCmd object, described next.

DoCmd Object

Use the methods of the DoCmd object to execute a macro action or menu command from within a Visual Basic procedure.

Syntax

DoCmd.*actionmethod* [*actionargument*],...

Notes

Some of the macro actions you'll commonly execute from Visual Basic include ApplyFilter, Close, FindNext and FindRecord (for searching the recordset of the current form and immediately displaying the result), Hourglass, Maximize, Minimize, MoveSize, OpenForm,

OpenQuery (to run a query that you don't need to modify), OpenReport, and RunCommand. Although you can run the Echo, GoToControl, GoToPage, RepaintObject, and Requery actions from Visual Basic using a method of the DoCmd object, it's more efficient to use the Echo, SetFocus, GoToPage, Repaint, and Requery methods of the object to which the method applies.

Examples

To open a form named frmCcompanies in Form view for data entry, enter the following:

```
DoCmd.OpenForm "frmCcompanies", acNormal, , , acAdd
```

To close a form named frmContacts, enter the following:

```
DoCmd.Close acForm, "frmContacts"
```

Executing an Access Command

To execute an Access command (one of the commands you can find on the ribbon), use the RunCommand method of either the DoCmd or Application object and supply a single action argument that is the numeric code for the command.

Syntax

```
[DoCmd.]RunCommand [actionargument],...
```

Notes

You can also use one of many built-in constants for *actionargument* to reference the command you want. When you use RunCommand, you can leave out the DoCmd or Application object if you want.

Examples

To execute the Save command from the Records group on the Home tab, enter the following:

```
RunCommand acCmdSaveRecord
```

To switch an open form to PivotChart view (execute the PivotChart View command in the Views group on the Home tab), enter the following:

```
RunCommand acCmdPivotChartView
```

To open the Find window while the focus is on a form (execute the Find command in the Find group on the Home tab), enter the following:

```
RunCommand acCmdFind
```

> **Note**
>
> **Visual Basic provides built-in constants for many of the macro action and RunCommand parameters. For more information, search for "Microsoft Access Constants" and "RunCommand Method" in Help.**

Actions with Visual Basic Equivalents

A few macro actions cannot be executed from a Visual Basic procedure. All but one of these actions, however, have equivalent statements in Visual Basic, as shown in Table 24-4.

Table 24-4 Visual Basic Equivalents for Macro Actions

Macro Action	Visual Basic Equivalent
AddMenu	No equivalent
MessageBox	MsgBox statement or function
RunApplication*	Shell function
RunCode	Call subroutine
SendKeys	SendKeys statement
SetValue	Variable assignment (=)
StopAllMacros	Stop or End statement
StopMacro	Exit Sub or Exit Function statement

* Database must be Trusted to execute this action.

Trapping Errors

One of the most powerful features of Visual Basic is its ability to trap all errors, analyze them, and take corrective action. In a well-designed production application, the user should never see any of the default error messages or encounter a code halt when an error occurs. Also, setting an error trap is often the best way to test certain conditions. For example, to find out if a query exists, your code can set an error trap and then attempt to reference the query object. In an application with hundreds of queries, using an error trap can also be faster than looping through all QueryDef objects. To enable error trapping, you use an On Error statement.

On Error Statement

Use an On Error statement to enable error trapping, establish the procedure to handle error trapping (the error handler), skip past any errors, or turn off error trapping.

Syntax

On Error {**GoTo** *lineID* | **Resume Next** | **GoTo 0**}

Notes

Use a GoTo *lineID* clause to establish a code block in your procedure that handles any error. The *lineID* can be a line number or a label.

Use a Resume Next clause to trap errors but skip over any statement that causes an error. You can call the Err function in a statement immediately following the statement that you suspect might have caused an error to see whether an error occurred. Err returns 0 if no error has occurred.

Use a GoTo 0 statement to turn off error trapping for the current procedure. If an error occurs, Visual Basic passes the error to the error routine in the calling procedure or opens an error dialog box if there is no previous error routine.

In your error handling statements, you can examine the built-in Err variable (the error number associated with the error) to determine the exact nature of the error. You can use the Error function to examine the text of the message associated with the error. If you use line numbers with your statements, you can use the built-in Erl function to determine the line number of the statement that caused the error. After taking corrective action, use a Resume statement to retry execution of the statement that caused the error. Use a Resume Next statement to continue execution at the statement immediately following the statement that caused the error. Use a Resume statement with a statement label to restart execution at the indicated label name or number. You can also use an Exit Function or Exit Sub statement to reset the error condition and return to the calling procedure.

Examples

To trap errors but continue execution with the next statement, enter the following:

```
On Error Resume Next
```

To trap errors and execute the statements that follow the MyError: label when an error occurs, enter the following:

```
On Error GoTo MyError
```

To turn off error trapping in the current procedure, enter the following:

```
On Error GoTo 0
```

If you create and run the following function with zero as the second argument, such as MyErrExample(3,0), the function will trigger an error by attempting to divide by zero, trap the error, display the error in an error handling section, and then exit gracefully:

```
Public Function MyErrExample(intA As Integer, intB As Integer) As Integer
' Set an error trap
On Error GoTo Trap_Error
  ' The following causes an error if intB is zero
  MyErrExample = intA / intB
ExitNice:
  Exit Function
Trap_Error:
  MsgBox "Something bad happened: " & Err & ", " & Error
  Resume ExitNice
End Function
```

Some Complex Visual Basic Examples

A good way to learn Visual Basic techniques is to study complex code that has been developed and tested by someone else. In the Conrad Systems Contacts and Housing Reservations sample databases, you can find dozens of examples of complex Visual Basic code that perform various tasks. The following sections describe two of the more interesting ones in detail.

A Procedure to Randomly Load Data

You've probably noticed a lot of sample data in both the Conrad Systems Contacts and the Housing Reservations databases. No, we didn't sit at our keyboards for hours entering sample data! Instead, we built a Visual Basic procedure that accepts some parameters entered on a form. In both databases, the form to load sample data is saved as zfrmLoadData. If you open this form in Contacts.accdb from the Navigation pane, you'll see that you use it to enter a beginning date, a number of days (max 365), a number of companies to load (max 25), a maximum number of contacts per company (max 10), and a maximum number of events per contact (max 25). You can also select the check box to delete all existing data before randomly loading new data. (The zfrmLoadData form in the Housing Reservations database offers some slightly different options.) Figure 24-15 shows this form with the values we used to load the Conrad Systems Contacts database.

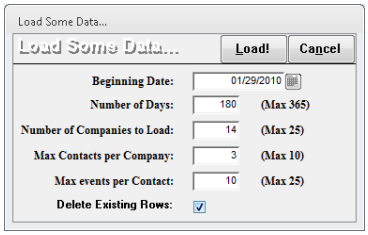

Figure 24-15 The zfrmLoadData form in the Conrad Systems Contacts sample database makes it easy to load sample data.

As you might expect, when you click Load!, our procedure examines the values entered and loads some sample data into tblCompanies, tblContacts, tblCompanyContacts, tblContactEvents, and tblContactProducts. The code picks random company names from ztblCompanies (a table containing a list of fictitious company names) and random person names from ztblPeople (a table containing names of Microsoft employees who have agreed to allow their names to be used in sample data). It also chooses random ZIP codes (and cities, counties, and states) from tlkpZips (a table containing U.S. ZIP codes, city names, state names, county names, and telephone area codes as of December 2002 that we licensed from CD Light, LLC—*http://www.zipinfo.com*). Figure 24-16 shows you the design of the query used in the code to pick random person names.

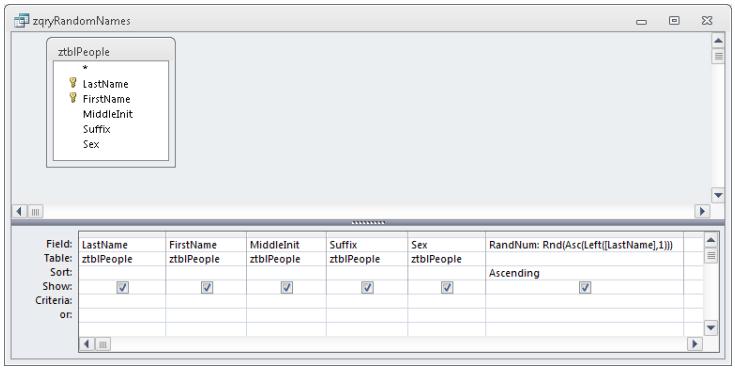

Figure 24-16 This query returns person names in a random sequence.

The query creates a numeric value to pass to the Rnd (random) function by grabbing the first character of the LastName field and then calculating the ASCII code value. The Rnd function returns some floating-point random value less than 1 but greater than or equal to zero. Asking the query to sort on this random number results in a random list of values each time you run the query.

> **Note**
>
> **If you open zqryRandomNames in Datasheet view, the RandNum column won't appear to be sorted correctly. In fact, the values change as you scroll through the data or resize the datasheet window. The database engine actually calls the Rnd function on a first pass through the data to perform the sort. Because the function depends on a value of one of the columns (LastName), Access assumes that other users might be changing this column—and therefore, the calculated result—as you view the data. Access calls the Rnd function again each time it refreshes the data display, so the actual values you see aren't the ones that the query originally used to sort the data.**

If you want to run this code, you should either pick a date starting after January 24, 2011, or select the option to delete all existing records first.

You can find the code in the cmdLoad_Click event procedure that runs when you click the Load button on the zfrmLoadData form. We've added line numbers to some of the lines in this code listing in the book so that you can follow along with the line-by-line explanations in Table 24-5, which follows the listing. Because the code loads data into both a multi-value field and an attachment field in the tblContacts table, it uses the DAO object model exclusively. (You cannot manipulate multi-value or attachment fields using the ADO object model.)

```
1 Private Sub cmdLoad_Click()
2 ' Code to load a random set of companies,
  '  contacts, events, and products
  ' Database variable
3 Dim db As DAO.Database
  ' Table delete list (if starting over)
  Dim rstDel As DAO.Recordset
  ' Company recordset; Contact recordset (insert only)
  Dim rstCo As DAO.Recordset, rstCn As DAO.Recordset
  ' Photo (attachment) and ContactType (multi-value) recordset
  Dim rstComplex As DAO.Recordset2
  ' CompanyContact recordset, ContactEvent recordset (insert only)
  Dim rstCoCn As DAO.Recordset, rstCnEv As DAO.Recordset
  ' A random selection of zips
  Dim rstZipRandom As DAO.Recordset
  ' ..and company names
  Dim rstCoRandom As DAO.Recordset
  ' .. and people names
  Dim rstPRandom As DAO.Recordset
  ' A recordset to pick "close" zip codes for contacts
  Dim rstZipClose As DAO.Recordset
  ' A recordset to pick contact events
  Dim rstEvents As DAO.Recordset
  ' Place to generate Picture Path
```

```
4 Dim strPicPath As String
  ' Places for path to backend database and folder
  Dim strBackEndPath As String, strBackEndFolder As String
  ' Place to generate a safe "compact to" name
  Dim strNewDb As String
  ' Places to save values from the form controls
  Dim datBeginDate As Date, intNumDays As Integer
  Dim intNumCompanies As Integer, intNumContacts As Integer
  Dim intNumEvents As Integer
  ' Lists of street names and types
5 Dim strStreetNames(1 To 9) As String, strStreetTypes(1 To 5) As String
  ' As string of digits for street addresses and area codes
  Const strDigits As String = "1234567890"
  ' List of Person Titles by gender
  Dim strMTitles(1 To 6) As String, strFTitles(1 To 7) As String
  ' Place to put male and female picture file names
  Dim strMPicture() As String, intMPicCount As Integer
  Dim strFPicture() As String, intFPicCount As Integer
  ' Some working variables
  Dim intI As Integer, intJ As Integer, intK As Integer
  Dim intL As Integer, intM As Integer, intR As Integer
  Dim varRtn As Variant, intDefault As Integer
  Dim datCurrentDate As Date, datCurrentTime As Date
  ' Variables to assemble Company and Contact records
  Dim strCompanyName As String, strCoAddress As String
  Dim strAreaCode As String, strPAddress As String
  Dim strThisPhone As String, strThisFax As String
  Dim strWebsite As String
  Dim lngThisCompany As Long
  Dim lngThisContact As Long, strProducts As String
  ' Set up to bail if something funny happens (it shouldn't)
6   On Error GoTo BailOut
    ' Initialize Streets
7   strStreetNames(1) = "Main"
    strStreetNames(2) = "Central"
    strStreetNames(3) = "Willow"
    strStreetNames(4) = "Church"
    strStreetNames(5) = "Lincoln"
    strStreetNames(6) = "1st"
    strStreetNames(7) = "2nd"
    strStreetNames(8) = "3rd"
    strStreetNames(9) = "4th"
    strStreetTypes(1) = "Street"
    strStreetTypes(2) = "Avenue"
    strStreetTypes(3) = "Drive"
    strStreetTypes(4) = "Parkway"
    strStreetTypes(5) = "Boulevard"
    ' Initialize person titles
    strMTitles(1) = "Mr."
    strMTitles(2) = "Dr."
    strMTitles(3) = "Mr."
    strMTitles(4) = "Mr."
```

```
    strMTitles(5) = "Mr."
    strMTitles(6) = "Mr."
    strFTitles(1) = "Mrs."
    strFTitles(2) = "Dr."
    strFTitles(3) = "Ms."
    strFTitles(4) = "Mrs."
    strFTitles(5) = "Ms."
    strFTitles(6) = "Mrs."
    strFTitles(7) = "Ms."
    ' Search for male picture names (should be in Current Path\Pictures)
 8  strPicPath = Dir(CurrentProject.Path & "\Pictures\PersonM*.bmp")
    ' Loop until Dir returns nothing (end of list or not found)
 9  Do Until (strPicPath = "")
      ' Add 1 to the count
      intMPicCount = intMPicCount + 1
      ' Extend the file name array
10    ReDim Preserve strMPicture(1 To intMPicCount)
      ' Add the file name to the array
      strMPicture(intMPicCount) = strPicPath
      ' Get next one
      strPicPath = Dir
11  Loop
    ' Search for female picture names (should be in Current Path\Pictures)
    strPicPath = Dir(CurrentProject.Path & "\Pictures\PersonF*.bmp")
    ' Loop until Dir returns nothing (end of list or not found)
12  Do Until (strPicPath = "")
      ' Add 1 to the count
      intFPicCount = intFPicCount + 1
      ' Extend the file name array
      ReDim Preserve strFPicture(1 To intFPicCount)
      ' Add the file name to the array
      strFPicture(intFPicCount) = strPicPath
      ' Get next one
      strPicPath = Dir
13  Loop
    ' Capture values from the form
14  datBeginDate = CDate(Me.BeginDate)
    intNumDays = Me.NumDays
    intNumCompanies = Me.NumCompanies
    intNumContacts = Me.NumContacts
    intNumEvents = Me.NumEvents
    ' Open the current database
15  Set db = CurrentDb
    ' Do they want to delete old rows?
16  If (Me.chkDelete = -1) Then
      ' Verify it
17    If vbYes = MsgBox("Are you SURE you want to delete " & _
        "all existing rows? " & vbCrLf & vbCrLf & _
        "(This will also compact the data file.)", _
        vbQuestion + vbYesNo + vbDefaultButton2, gstrAppTitle) Then
        ' Open the table that tells us the safe delete sequence
18      Set rstDel = db.OpenRecordset("SELECT * FROM " & _
```

```
            "ztblDeleteSeq ORDER BY Sequence", _
            dbOpenSnapshot, dbForwardOnly)
          ' Loop through them all
19        Do Until rstDel.EOF
           ' Check for tblContacts
           If rstDel!TableName = "tblContacts" Then
            ' Can't just delete all rows in the linked table - must do one at a time
            ' Open a recordset on tblContacts
            Set rstCn = db.OpenRecordset("tblContacts", dbOpenDynaset)
            ' Loop through them all
            Do Until rstCn.EOF
             ' Put it in edit mode
             rstCn.Edit
             ' Get the first complex field's recordset (ContactType)
             Set rstComplex = rstCn!ContactType.Value
             ' Loop and delete them all
             Do Until rstComplex.EOF
               ' Delete it
               rstComplex.Delete
               ' Get the next
               rstComplex.MoveNext
             Loop
             ' Get the second complex field's recordset (Photo)
             Set rstComplex = rstCn!Photo.Value
             ' Loop and delete them all
             Do Until rstComplex.EOF
               ' Delete it
               rstComplex.Delete
               ' Get the next
               rstComplex.MoveNext
               Loop
               ' Save the row with the deleted complex data
               rstCn.Update
               ' Now finally delete the contact
               rstCn.Delete
               ' Get the next one
               rstCn.MoveNext
             Loop
             ' Clear the objects
             Set rstComplex = Nothing
             rstCn.Close
             Set rstCn = Nothing
           Else
            ' Execute a delete
20          db.Execute "DELETE * FROM " & rstDel!TableName, _
              dbFailOnError
           End If
            ' Go to the next row
            rstDel.MoveNext
          Loop
          ' Figure out the path to the backend data
21        strBackEndPath = Mid(db.TableDefs("tblContacts").Connect, 11)
```

```
        ' Figure out the backend folder
22      strBackEndFolder = Left(strBackEndPath, _
          InStrRev(strBackEndPath, "\"))
        ' Calculate a "compact to" database name
        strNewDb = "TempContact" & Format(Now, "hhnnss") & ".accdb"
        ' Compact the database into a new name
23      DBEngine.CompactDatabase strBackEndPath, _
          strBackEndFolder & strNewDb
        ' Delete the old one
24      Kill strBackEndPath
        ' Rename the new
        Name strBackEndFolder & strNewDb As strBackEndPath
      Else
        ' Turn off the delete flag - changed mind
        Me.chkDelete = 0
25    End If
26  End If
    ' Initialize the randomizer on system clock
27  Randomize
    ' Open all output recordsets
28  Set rstCo = db.OpenRecordset("tblCompanies", dbOpenDynaset)
    Set rstCn = db.OpenRecordset("tblContacts", dbOpenDynaset)
    Set rstCoCn = db.OpenRecordset("tblCompanyContacts", dbOpenDynaset)
    Set rstCnEv = db.OpenRecordset("tblContactEvents", dbOpenDynaset)
    ' Open the random recordsets
    Set rstZipRandom = db.OpenRecordset("zqryRandomZips", dbOpenDynaset)
    Set rstCoRandom = db.OpenRecordset("zqryRandomCompanies", dbOpenDynaset)
    Set rstPRandom = db.OpenRecordset("zqryRandomNames", dbOpenDynaset)
    ' Open the Events/products list
    Set rstEvents = db.OpenRecordset("zqryEventsProducts", dbOpenDynaset)
    ' Move to the end to get full recordcount
    rstEvents.MoveLast
    ' Turn on the hourglass
29  DoCmd.Hourglass True
    ' Initialize the status bar
30  varRtn = SysCmd(acSysCmdInitMeter, "Creating Companies...", _
      intNumCompanies)
    ' Outer loop to add Companies
31  For intI = 1 To intNumCompanies
      ' Start a new company record
      rstCo.AddNew
      ' Clear the saved website
      strWebsite = ""
      ' Grab the next random "company" name
32    strCompanyName = rstCoRandom!CompanyName
      ' .. and the website
33    rstCo!Website = rstCoRandom!CompanyName & "#" & _
        rstCoRandom!Web & "##" & rstCoRandom!CompanyName & " Website"
      strWebsite = rstCo!Website
34    rstCo!CompanyName = strCompanyName
      ' Generate a random street number
35    intR = Int((7 * Rnd) + 1)
```

```
        strCoAddress = Mid(strDigits, intR, 4)
        ' Now pick a random street name
        intR = Int((9 * Rnd) + 1)
        strCoAddress = strCoAddress & " " & strStreetNames(intR)
        ' and street type
        intR = Int((5 * Rnd) + 1)
        strCoAddress = strCoAddress & " " & strStreetTypes(intR)
        rstCo!Address = strCoAddress
        ' Fill in random values from the zip code table
36      rstCo!City = rstZipRandom!City
        rstCo!County = rstZipRandom!County
        rstCo!StateOrProvince = rstZipRandom!State
        rstCo!PostalCode = rstZipRandom!ZipCode
        ' Generate a random Area Code
37      intR = Int((8 * Rnd) + 1)
        strAreaCode = Mid(strDigits, intR, 3)
        ' Generate a random phone number (0100 - 0148)
        intR = Int((48 * Rnd) + 1) + 100
        strThisPhone = strAreaCode & "555" & Format(intR, "0000")
        rstCo!PhoneNumber = strThisPhone
        ' Add 1 for the fax number
        strThisFax = strAreaCode & "555" & Format(intR + 1, "0000")
        rstCo!FaxNumber = strThisFax
        ' Save the new Company ID
38      lngThisCompany = rstCo!CompanyID
        ' .. and save the new Company
        rstCo.Update
        ' Now, do some contacts for this company
        '  - calc a random number of contacts
39      intJ = Int((intNumContacts * Rnd) + 1)
        ' Set up the recordset of Zips "close" to the Work Zip
40      Set rstZipClose = db.OpenRecordset("SELECT * FROM tlkpZips " & _
          "WHERE ZipCode BETWEEN '" & _
          Format(CLng(rstZipRandom!ZipCode) - 5, "00000") & _
          "' AND '" & Format(CLng(rstZipRandom!ZipCode) + 5, "00000") & _
          "'", dbOpenDyanaset)
        ' Move to last row to get accurate count
        rstZipClose.MoveLast
        ' Make the first contact the company default
        intDefault = True
        ' Loop to add contacts
41      For intK = 1 To intJ
          ' Start a new record
          rstCn.AddNew
          ' Put in the name info from the random people record
42        rstCn!LastName = rstPRandom!LastName
          rstCn!FirstName = rstPRandom!FirstName
          rstCn!MiddleInit = rstPRandom!MiddleInit
          rstCn!Suffix = rstPRandom!Suffix
          ' Select title  and picture based on gender of person
43        If rstPRandom!Sex = "f" Then
            ' Pick a random female title and picture
```

```
            intR = Int((7 * Rnd) + 1)
            rstCn!Title = strFTitles(intR)
            ' Make sure we have some picture file names
            If intFPicCount <> 0 Then
              ' Pick a random file name
              intR = Int((intFPicCount * Rnd) + 1)
              strPicPath = strFPicture(intR)
              ' Don't reuse it
              For intL = intR To intFPicCount - 1
                  strFPicture(intL) = strFPicture(intL + 1)
              Next intL
              intFPicCount = intFPicCount - 1
            Else
              ' Set empty picture name
              strPicPath = ""
            End If
44        Else
            ' Pick a random male title and picture
            intR = Int((6 * Rnd) + 1)
            rstCn!Title = strMTitles(intR)
            ' Make sure we have some picture file names
            If intMPicCount <> 0 Then
              ' Pick a random file name
              intR = Int((intMPicCount * Rnd) + 1)
              strPicPath = strMPicture(intR)
              ' Don't reuse it
              For intL = intR To intMPicCount - 1
                  strMPicture(intL) = strMPicture(intL + 1)
              Next intL
              intMPicCount = intMPicCount - 1
            Else
              ' Set empty picture name
              strPicPath = ""
            End If
45        End If
          ' Set contact type to "Customer" - complex data type
46        Set rstComplex = rstCn!ContactType.Value
          rstComplex.AddNew
          rstComplex!Value = "Customer"
          rstComplex.Update
47        ' Copy the company website
          rstCn!Website = strWebsite
          ' Set up a dummy email
          rstCn!EmailName = rstPRandom!FirstName & " " & _
            rstPRandom!LastName & "#mailto:" & Left(rstPRandom!FirstName, 1) & _
            rstPRandom!LastName & "@" _
            & Mid(rstCoRandom!Web, Instr(rstCoRandom!Web, "http://www.") + 11)
          ' Strip off the trailing "/"
          rstCn!EmailName = Left(rstCn!EmailName, Len(rstCn!EmailName) - 1)
          ' Pick a random birth date between Jan 1, 1940 and Dec 31, 1979
          ' There are 14,610 days between these dates
          intR = Int((14610 * Rnd) + 1)
```

```
            rstCn!BirthDate = #12/31/1939# + Int((14610 * Rnd) + 1)
            ' Set Default Address to 'work'
            rstCn!DefaultAddress = 1
            ' Copy work address from Company
            rstCn!WorkAddress = strCoAddress
            rstCn!WorkCity = rstZipRandom!City
            rstCn!WorkStateOrProvince = rstZipRandom!State
            rstCn!WorkPostalCode = rstZipRandom!ZipCode
            rstCn!WorkPhone = strThisPhone
            rstCn!WorkFaxNumber = strThisFax
            ' Generate a random street number for home address
            intR = Int((7 * Rnd) + 1)
            strPAddress = Mid(strDigits, intR, 4)
            ' Now pick a random street name
            intR = Int((9 * Rnd) + 1)
            strPAddress = strPAddress & " " & strStreetNames(intR)
            ' and street type
            intR = Int((5 * Rnd) + 1)
            strPAddress = strPAddress & " " & strStreetTypes(intR)
            rstCn!HomeAddress = strPAddress
            ' Position to a "close" random zip
48          intR = rstZipClose.RecordCount
            intR = Int(intR * Rnd)
            rstZipClose.MoveFirst
            If intR > 0 Then rstZipClose.Move intR
            rstCn!HomeCity = rstZipClose!City
            rstCn!HomeStateOrProvince = rstZipClose!State
            rstCn!HomePostalCode = rstZipClose!ZipCode
            ' Generate a random phone number (0150 - 0198)
            intR = Int((48 * Rnd) + 1) + 149
            rstCn!HomePhone = strAreaCode & "555" & Format(intR, "0000")
            ' Add 1 for the fax number
            rstCn!MobilePhone = strAreaCode & "555" & Format(intR + 1, "0000")
            ' Save the new contact ID
49          lngThisContact = rstCn!ContactID
            ' If got a random photo name, load it
50          If strPicPath <> "" Then
              ' Open the special photo editing recordset
51            Set rstComplex = rstCn!Photo.Value
              rstComplex.Addnew
              rstComplex!FileData.LoadFromFile _
                (CurrentProject.Path & "\Pictures\" & strPicPath)
              rstComplex.Update
            End If
            ' Finally, save the row
            rstCn.Update
            ' Insert linking CompanyContact record
52          rstCoCn.AddNew
            ' Set the Company ID
            rstCoCn!CompanyID = lngThisCompany
            ' Set the Contact ID
            rstCoCn!ContactID = lngThisContact
```

```
        ' Make this the default company for the contact
        rstCoCn!DefaultForContact = True
        ' Set default for company - 1st contact will be the default
        rstCoCn!DefaultForCompany = intDefault
        ' Reset intDefault after first time through
        intDefault = False
        ' Save the linking row
        rstCoCn.Update
        ' Now, do some contacts events for this contact
        '  - calc a random number of events
53      intM = Int((intNumEvents * Rnd) + 1)
        ' Clear the Products sold string
        strProducts = ""
        ' Loop to add some events
54      For intL = 1 To intM
          ' Start a new row
          rstCnEv.AddNew
          ' Set the Contact ID
          rstCnEv!ContactID = lngThisContact
          ' Calculate a random number of days
          intR = Int(intNumDays * Rnd)
          datCurrentDate = datBeginDate + intR
          ' Calculate a random time between 8am and 8pm (no seconds)
          datCurrentTime = CDate(Format(((0.5 * Rnd) + 0.3333), "hh:nn"))
          ' Set the contact date/time
          rstCnEv!ContactDateTime = datCurrentDate + datCurrentTime
55 TryAgain:
          ' Position to a random event
56        intR = rstEvents.RecordCount
          intR = Int(intR * Rnd)
          rstEvents.MoveFirst
          If intR > 0 Then rstEvents.Move intR
          ' If a product sale event,
57        If (rstEvents!ContactEventProductSold = True) Then
            ' Can't sell the same product twice to the same contact
            If InStr(strProducts, _
              Format(rstEvents!ContactEventProductID, "00")) <> 0 Then
              ' ooops.  Loop back to pick a different event
58            GoTo TryAgain
            End If
          End If
          ' Set the Event Type
59        rstCnEv!ContactEventTypeID = rstEvents!ContactEventTypeID
          ' Set the follow-up
          rstCnEv!ContactFollowUp = rstEvents!ContactEventRequiresFollowUp
          ' Set the follow-up date
          If (rstEvents!ContactEventRequiresFollowUp = True) Then
            rstCnEv!ContactFollowUpDate = datCurrentDate + _
              rstEvents!ContactEventFollowUpDays
          End If
          ' Save the record
60        rstCnEv.Update
```

```
            ' If this event is a product sale,
61          If (rstEvents!ContactEventProductSold = True) Then
              ' Call the routine to also add a product record!
              varRtn = Add_Product(lngThisCompany, lngThisContact, _
                rstEvents!ContactEventProductID, datCurrentDate)
              ' Add the product to the products sold string
              strProducts = strProducts & " " & _
                Format(rstEvents!ContactEventProductID, "00")
            End If
          ' Loop to do more events
62        Next intL
        ' Move to the next random person record
63      rstPRandom.MoveNext
        ' and loop to do more contacts
64      Next intK
65      rstZipClose.Close
        Set rstZipClose = Nothing
        ' Move to the next random zip record
66      rstZipRandom.MoveNext
        ' Update the status bar
67      varRtn = SysCmd(acSysCmdUpdateMeter, intI)
        ' Move to the next Company row
        rstCoRandom.MoveNext
      ' Loop until done
68    Next intI
      ' Clear the status bar
69    varRtn = SysCmd(acSysCmdClearStatus)
      ' Done with error trapping, too
      On Error GoTo 0
      ' Be nice and close everything up
      rstCo.Close
      rstCn.Close
      rstCoCn.Close
      rstCnEv.Close
      rstZipRandom.Close
      rstCoRandom.Close
      ' Finally, generate invoices for most records if all deleted
      If (Me.chkDelete = -1) Then
          intI = Do_Invoices()
      End If
      ' Turn off the hourglass
70    DoCmd.Hourglass False
      MsgBox "Done!", vbExclamation, gstrAppTitle
      DoCmd.Close acForm, Me.Name
71  Done:
      Set rstCo = Nothing
      Set rstCn = Nothing
      Set rstCoCn = Nothing
      Set rstCnEv = Nothing
      Set rstZipRandom = Nothing
      Set rstCoRandom = Nothing
      Set rstComplex = Nothing
```

```
     Set db = Nothing
     Exit Sub
72 BailOut:
     MsgBox "Unexpected error: " & Err & ", " & Error
     ' Turn off the hourglass
     DoCmd.Hourglass False
     varRtn = SysCmd(acSysCmdClearStatus)
     Resume Done
73 End Sub
```

Table 24-5 lists the statement line numbers and explains the code on key lines in the preceding Visual Basic code example.

Table 24-5 Explanation of Example Code to Load Random Data

Line	Explanation
1	Declare the beginning of the subroutine. The subroutine has no arguments.
2	You can begin a comment anywhere on a statement line by preceding the comment with a single quotation mark. You can also create a comment statement using the Rem statement.
3	Declare local variables for a DAO Database object and all the DAO Recordset objects used in this code.
4	Beginning of the declarations of all local variables. You should always explicitly define variables in your code.
5	This procedure uses several arrays in which it stores street names, street types, male person titles, female person titles, and the paths to male and female pictures. Code later in the procedure randomly chooses values from these arrays.
6	Set an error trap; the BailOut label is at line 71.
7	Code to initialize the arrays begins here. Note that separate arrays handle male and female titles.
8	Use the Dir function to find available male picture names in the Pictures subfolder under the location of the current database. Note that if you move the sample database, this code won't find any pictures to load. When Dir finds a matching file, it returns the file name as a string. The code subsequently calls Dir with no arguments inside the following loop to ask for the next picture.
9	Begin a loop to load male pictures, and keep looping until the picture file name is an empty string (Dir found no more files).
10	Note the use of ReDim Preserve to dynamically expand the existing file name array for male pictures without losing any entries already stored.
11	End of the loop started at statement number 9.
12	This loop finds all the female pictures available and loads them into the array that holds picture file names for females.
13	End of the loop started at statement number 12.

Line	Explanation
14	The next several lines of code capture the values from the form. Validation rules in the form controls make sure that the data is valid.
15	Initialize the Database object.
16	Check to see if you selected the option to delete all existing rows.
17	Use the MsgBox function to verify that you really want to delete existing data.
18	The ztblDeleteSeq table contains the table names in a correct sequence for deletes from the bottom up so that this code doesn't violate any referential integrity rules. Note that the recordset is opened as a forward-only snapshot for efficiency.
19	Start a loop to process all the table names in ztblDeleteSeq. If tblContacts needs to be deleted, loop through each contact record individually and delete all complex data from the ContactType and Photo fields. Delete each contact record after complex data finishes the deletions.
20	Use the Execute method of the Database object to run the DELETE SQL commands on remaining tables.
21	Figure out the path to the linked data file by examining the Connect property of one of the linked tables.
22	Extract the folder name of the data file using the Left and InStrRev functions.
23	Use the CompactDatabase method of the DBEngine object to compact the data file into a new one—TempContact*hhmmss*.accdb—where *hhmmss* is the current time to avoid conflicts.
24	Use the Kill command to delete the old file and the Name command to rename the compacted temp copy.
25	Terminate the If statement on line 17.
26	Terminate the If statement on line 16.
27	Initialize the randomizer so that all random recordsets are always different.
28	Open all the recordsets needed in this code.
29	Turn the mouse pointer into an hourglass to let you know the transaction is under way and might take a while. You could also set the Screen.MousePointer property to 11 (busy).
30	The SysCmd utility function provides various useful options such as finding out the current directory for msaccess.exe (the Access main program), and the current version of Access. It also has options to display messages and a progress meter on the status bar. This code calls SysCmd to initialize the progress meter you see as the code loads the data.
31	Start the main loop to load company data.
32	Save the company name from the random recordset in a local variable.
33	Generate the website hyperlink from the company name and the Web field.
34	Set the company name in the new company record.

Line	Explanation
35	The next several lines of code use the Rnd function to randomly generate a four-digit street address and randomly choose a street name and street type from the arrays loaded earlier.
36	Grab the city, county, state, and ZIP code from the current row in the random ZIP Code query.
37	Use Rnd again to generate a fake phone area code and phone and fax numbers.
38	The primary key of tblCompanies is an AutoNumber field. Access automatically generates the next number as soon as you update any field in a new record. This code saves the new company ID to use in related records and writes the company record with the Update method.
39	Calculate a random number of contacts to load for the new company based on the maximum you specified in the form.
40	Open a recordset that chooses the ZIP codes that are five higher or lower than the random ZIP code for the company. (It makes sense that the employees of the company live nearby.)
41	Start the loop to add contacts for this company.
42	Update the new contacts record with a random name plucked from the random person names query.
43	The records in the ztblPeople table have a gender field to help choose an appropriate title and picture for the contact. The statements following this If statement load female data, and the statements following the Else statement on line 44 load male data.
44	This Else statement matches the If on line 43. Statements following this choose male data.
45	This End If closes the If on line 43.
46	The ContactType field is a multi-value field, so it must open a recordset on the field's Value property even though we're specifying only one value.
47	Finish generating fields for the contacts record, including the website copied from the company, a fake e-mail name, and a random birth date and addresses.
48	Choose a random ZIP code for the contact near the company ZIP code from the recordset opened on line 40. Also generate phone and fax numbers.
49	The primary key for tblContacts is also an AutoNumber field, so save the new value to use to generate related records and save the new contact.
50	If the code found a good picture file name earlier (male or female), then the following code adds that picture to the record.
51	Photo is an attachment field that works similarly to multi-value fields in code. The code opens a recordset and uses the LoadFromFile method to insert the picture using its file path.
52	Create the linking record in tblCompanyContacts from the saved CompanyID and ContactID. The first contact created is always the default contact for the company.
53	Calculate a random number of events to load for this contact.

Line	Explanation
54	Start the loop to add contact events. The following several lines calculate a random contact date and time within the range you specified on the form.
55	Code at line 58 goes here if the random product picked was already sold to this contact.
56	Choose a random event.
57	If the random event is a product sale, verify that this product isn't already sold to this contact. A product can be sold to a contact only once.
58	The code loops back up to line 55 to choose another event if this is a duplicate product.
59	Finish updating the fields in the new contact event record.
60	Save the new contact event.
61	If the event was a product sale, call the Add_Product function that's also in this form module to add a row to tblContactProducts. This code passes the company ID, contact ID, product ID, and the date of the event to the function. It also saves the product ID to be sure it isn't sold again to this contact.
62	This Next statement closes the loop started on line 54.
63	Move to the next random person record.
64	Loop back up to line 41.
65	Close the recordset of ZIP codes close to the company ZIP code.
66	Get the next random ZIP code for the next company.
67	Update the status bar to indicate you're done with another company.
68	Loop back up to line 31.
69	Clear the status bar and close all recordsets.
70	Clear the hourglass set on line 29. Also issue the final MsgBox confirming that all data is now loaded. Finally, close this form and exit.
71	Set all recordsets to nothing and exit the subroutine.
72	Any trapped error comes here. This code simply displays the error, clears the mouse pointer and the status bar, and exits. (If you don't reset the mouse pointer and clear the status bar, Access won't do it for you.)
73	End of the subroutine.

A Procedure to Examine All Error Codes

In the Housing Reservations database (Housing.accdb), we created a function that dynamically creates a new table and then inserts into the table (using DAO) a complete list of all the error codes used by Access and the text of the error message associated with each error code. You can find a partial list of the error codes in Help, but the table in the Housing Reservations sample database provides the best way to see a list of all the error codes. You

might find this table useful as you begin to create your own Visual Basic procedures and set error trapping in them.

> **Note**
>
> **You can find the ADO equivalent of this example in the modExamples module in the Conrad Systems Contacts sample database.**

The name of the function is CreateErrTable, and you can find it in the modExamples module. The function statements are listed next. You can execute this function by entering the following in the Immediate window:

```
?CreateErrTable
```

The sample database contains the ErrTable table, so the code will ask you if you want to delete and rebuild the table. You should click Yes to run the code. Again, we've added line numbers to some of the lines in this code listing so that you can follow along with the line-by-line explanations in Table 24-6, which follows the listing.

```
 1 Function CreateErrTable()
' This function creates a table containing a list of
'  all the valid Access application error codes
'  You can find the ADO version of this procedure in Contacts.accdb
 2   ' Declare variables used in this function
 3   Dim dbMyDatabase As DAO.Database, tblErrTable As DAO.TableDef, _
       fldMyField As DAO.Field, idxPKey As DAO.Index
 4   Dim rcdErrRecSet As DAO.Recordset, lngErrCode As Long, _
       intMsgRtn As Integer
 5   Dim varReturnVal As Variant, varErrString As Variant, _
       ws As DAO.Workspace
     ' Create Errors table with Error Code and Error String fields
     ' Initialize the MyDatabase database variable
     ' to the current database
 6   Set dbMyDatabase = CurrentDb
 7   Set ws = DBEngine.Workspaces(0)
     ' Trap error if table doesn't exist
     ' Skip to next statement if an error occurs
 8   On Error Resume Next
 9   Set rcdErrRecSet = dbMyDatabase.OpenRecordset("ErrTable")
10   Select Case Err  ' See whether error was raised
11     Case 0  ' No error - table must already exist
12       On Error GoTo 0  ' Turn off error trapping
13       intMsgRtn = MsgBox("ErrTable already " & _
           "exists. Do you want to delete and " & _
           "rebuild all rows?", vbQuestion + vbYesNo, _
           "Access 2010 Inside Out")
14       If intMsgRtn = vbYes Then
```

```
          ' Reply was YES—delete rows and rebuild
          ' Run quick SQL to delete rows
15        dbMyDatabase.Execute_
            "DELETE * FROM ErrTable;", dbFailOnError
16      Else                    ' Reply was NO—done
17        rcdErrRecSet.Close    ' Close the table
18        Exit Function         ' And exit
19      End If
20    Case 3011, 3078           ' Couldn't find table,
                                ' so build it
21      On Error GoTo 0   ' Turn off error trapping
        ' Create a new table to contain error rows
22      Set tblErrTable = _
          dbMyDatabase.CreateTableDef("ErrTable")
        ' Create a field in ErrTable to contain the
        ' error code
23      Set fldMyField = tblErrTable.CreateField( _
          "ErrorCode", DB_LONG)
        ' Append "ErrorCode" field to the fields
        ' collection in the new table definition
24      tblErrTable.Fields.Append fldMyField
        ' Create a field in ErrTable for the error
        ' description
25      Set fldMyField = _
          tblErrTable.CreateField("ErrorString", _
          DB_MEMO)
        ' Append the "ErrorString" field to the fields
        ' collection in the new table definition
26      tblErrTable.Fields.Append fldMyField
        ' Append the new table to the TableDefs
        ' collection in the current database
27      dbMyDatabase.TableDefs.Append tblErrTable
        ' Set text field width to 5" (7200 twips)
        ' (calls sub procedure)
28      SetFieldProperty _
          tblErrTable![ErrorString], _
          "ColumnWidth", DB_INTEGER, 7200
        ' Create a Primary Key
29      Set idxPKey = tblErrTable.CreateIndex("PrimaryKey")
        ' Create and append the field to the index fields collection
30      idxPKey.Fields.Append idxPKey.CreateField("ErrorCode")
        ' Make it the Primary Key
        idxPKey.Primary = True
        ' Create the index
31      tblErrTable.Indexes.Append idxPKey
        ' Set recordset to Errors Table recordset
32      Set rcdErrRecSet = _
          dbMyDatabase.OpenRecordset("ErrTable")
33    Case Else
        ' Can't identify the error—write message
        ' and bail
34      MsgBox "Unknown error in CreateErrTable " & _
```

```
          Err & ", " & Error$(Err), 16
35        Exit Function
36    End Select
      ' Initialize progress meter on the status bar
37    varReturnVal = SysCmd(acSysCmdInitMeter, _
        "Building Error Table", 32767)
      ' Turn on hourglass to show this might take
      ' a while
38    DoCmd.Hourglass True
      ' Start a transaction to make it go fast
39    ws.BeginTrans
      ' Loop through Microsoft Access error codes,
      ' skipping codes that generate
      ' "Application-defined or object-define error"
      ' message.
40    For lngErrCode = 1 To 32767
41      varErrString = AccessError(lngErrCode)
        If IsNothing(varErrString) Or _
           varErrString = "Application-defined or object-defined error" Then
          ' If AccessError returned nothing, then try Error
          varErrString = Error(lngErrCode)
        End If
42      If Not IsNothing(varErrString) Then
43        If varErrString <> "Application-" & _
            "defined or object-defined error" Then
            ' Add each error code and string to
            ' Errors table
44          rcdErrRecSet.AddNew
45          rcdErrRecSet("ErrorCode") = lngErrCode
            ' Put the message text in the record
46          rcdErrRecSet("ErrorString") = varErrString
47          rcdErrRecSet.Update
48        End If
49      End If
        ' Update the status meter
50      varReturnVal = SysCmd(acSysCmdUpdateMeter, _
          lngErrCode)
        ' Process next error code
51    Next lngErrCode
      ' Commit all added rows
52    ws.CommitTrans
      ' Close recordset.
53    rcdErrRecSet.Close
      ' Turn off the hourglass — we're done
54    DoCmd.Hourglass False
      ' And reset the status bar
55    varReturnVal = SysCmd(acSysCmdClearStatus)
      ' Select new table in the Navigation pane
      ' to refresh the list
56    DoCmd.SelectObject acTable, "ErrTable", True
      ' Open a confirmation dialog box
```

```
57   MsgBox "Errors table created."
58 End Function
```

Table 24-6 lists the statement line numbers and explains the code on each line in the preceding Visual Basic code example.

Table 24-6 Explanation of Example Code to Examine Error Codes

Line	Explanation
1	Declare the beginning of the function. The function has no arguments.
2	You can begin a comment anywhere on a statement line by preceding the comment with a single quotation mark. You can also create a comment statement using the Rem statement.
3	Declare local variables for a Database object, a TableDef object, a Field object, and an Index object.
4	Declare local variables for a Recordset object, a Long Integer, and an Integer.
5	Declare local variables for a Variant that is used to accept the return value from the SysCmd function, a Variant that is used to accept the error string returned by the AccessError function, and a Workspace object.
6	Initialize the Database object variable by setting it to the current database.
7	Initialize the Workspace object by setting it to the current workspace.
8	Enable error trapping but execute the next statement if an error occurs.
9	Initialize the Recordset object variable by attempting to open the ErrTable table. If the table does not exist, this generates an error.
10	Call the Err function to see whether an error occurred. The following Case statements check the particular error values that interest you.
11	The first Case statement tests for an Err value of 0, indicating that no error occurred. If no error occurred, the table already existed and opened successfully.
12	Turn off error trapping because you don't expect any more errors.
13	Use the MsgBox function to ask whether you want to clear and rebuild all rows in the existing table. The vbQuestion intrinsic constant asks MsgBox to display the question icon, and the vbYesNo intrinsic constant requests Yes and No buttons (instead of the default OK button). The statement assigns the value returned by MsgBox so that you can test it on the next line.
14	If you click Yes, MsgBox returns the value of the intrinsic constant vbYes. (vbYes happens to be the integer value 6, but the constant name is easier to remember than the number.)
15	Run a simple SQL statement to delete all the rows in the error table.
16	Else clause that goes with the If statement on line 14.
17	Close the table if the table exists and you clicked the No button on line 13.
18	Exit the function.
19	End If statement that goes with the If statement on line 14.

Line	Explanation
20	Second Case statement. Error codes 3011 and 3078 are both "object not found."
21	Turn off error trapping because you don't expect any more errors.
22	Use the CreateTableDef method on the database to start a new table definition. This is the same as clicking the Table Design button in the Tables group on the Create tab on the ribbon.
23	Use the CreateField method on the new table to create the first field object—a long integer (the intrinsic constant DB_LONG) named ErrorCode.
24	Append the first new field to the Fields collection of the new Table object.
25	Use the CreateField method to create the second field—a memo field named ErrorString.
26	Append the second new field to the Fields collection of the new Table object.
27	Save the new table definition by appending it to the TableDefs collection of the Database object. If you were to halt the code at this point and repaint the Navigation pane, you would find the new ErrTable listed.
28	Call the SetFieldProperty subroutine in this module to set the column width of the ErrorString field to 7200 twips (5 inches). This ensures that you can see more of the error text when you open the table in Datasheet view.
29	Use the CreateIndex method of the TableDef to begin building an index.
30	Create a single field and append it to the Fields collection of the index. The following statement sets the Primary property of the index to True to indicate that this will be the primary key.
31	Save the new primary key index by appending it to the Indexes collection of the TableDef.
32	Open a recordset by using the OpenRecordset method on the table.
33	This Case statement traps all other errors.
34	Show a message box with the error number and the error message.
35	Exit the function after an unknown error.
36	End Select statement that completes the Select Case statement on line 10.
37	Call the SysCmd function to place a "building table" message on the status bar and initialize a progress meter. The CreateErrTable function will look at 32,767 different error codes.
38	Turn the mouse pointer into an hourglass to indicate that this procedure will take a few seconds.
39	Use the BeginTrans method of the Workspace object to start a transaction. Statements within a transaction are treated as a single unit. Changes to data are saved only if the transaction completes success fully with a CommitTrans method. Using transactions when you're updating records can speed performance by reducing disk access.
40	Start a For loop to check each error code from 1 through 32,767.

Line	Explanation
41	Assign the error text returned by the AccessError function to the variable varErrString. If the string is empty or returned "Application-defined or object-defined error," try calling the Error function to get the text of the message.
42	Call the IsNothing function in the modUtility module of the sample database to test whether the text returned is blank. You don't want blank rows, so don't add a row if the AccessError function for the current error code returns a blank string.
43	Lots of error codes are defined as "Application-defined or object-defined error." You don't want any of these, so this statement adds a row only if the AccessError function for the current error code doesn't return this string.
44	Use the AddNew method to start a new row in the table.
45	Set the ErrorCode field equal to the current error code.
46	Save the text of the message in the ErrorString field. Because we defined the field as a memo, we don't need to worry about the length of the text.
47	Use the Update method to save the new row.
48	End If statement that completes the If statement on line 43.
49	End If statement that completes the If statement on line 42.
50	After handling each error code, update the progress meter on the status bar to show how far you've gotten.
51	Next statement that completes the For loop begun on line 40. Visual Basic increments lngErrCode by 1 and executes the For loop again until lngErrCode is greater than 32,767.
52	CommitTrans method that completes the transaction begun on line 39.
53	After looping through all possible error codes, close the recordset.
54	Change the mouse pointer back to normal.
55	Clear the status bar.
56	Put the focus on the ErrTable table in the Navigation pane.
57	Display a message box confirming that the function has completed.
58	End of the function.

Working with 64-Bit Access Visual Basic for Applications

With the creation of 64-bit versions of the Office 2010 applications, Microsoft has introduced a new 64-bit version of Visual Basic. In general, your Visual Basic code runs without modification with 64-bit Visual Basic. However, there are a few issues and opportunities when using 64-bit Visual Basic. For any programming language, the biggest source of issues when moving between different-sized architectures is the size of pointers. As you learned earlier in this chapter, pointers are variables that hold memory addresses. When you are working on 32-bit systems, these pointers are 32-bit variables; and on 64-bit systems, they are 64-bit variables.

One of the great things about Visual Basic is that pointers are managed on behalf of programmers, relieving them of the tedium of managing pointers themselves. Still, there are situations in which a programmer needs to manage a pointer manually, in particular when interacting with the Windows API. Prior to Office 2010, Visual Basic had no official pointer data type. Moving to 64-bit with Access 2010 has both advantages and disadvantages. The advantage is that since there are no pointers, most existing Visual Basic code in your applications works just fine with 64-bit Access without any modifications. None of the data types in Visual Basic change their size when moving to 64-bit; in particular, a Long is still 32 bits.

The disadvantage is that although "officially" there was no pointer data type, some Visual Basic code used Long variables to hold memory addresses as an "unofficial" pointer. Microsoft, in fact, promoted this behavior for making calls to the Windows operating system. Since Long variables did not increase in size when moving to 64-bit, executing code that stored pointers in Long variables will result in unexpected behavior for your applications—even possibly crashing.

So what does this mean to you as an Access developer? This means that you need to identify all the places where a pointer could enter or exit Visual Basic and modify them. These include:

- Declare statements
- VarPtr functions
- StrPtr functions
- ObjPtr functions

Thankfully, the 64-bit Visual Basic compiler helps identify these situations.

Using Declare Statements

The Visual Basic Declare statement is commonly used to access Windows APIs, although it can also be used to call any DLL entry point. For example, the following Declare statement sets up a call to the Windows RegOpenKeyA API for opening a Windows registry key:

```
Declare Function RegOpenKeyA Lib "advapi32.dll" _
    (ByVal Key As Long, ByVal SubKey As String, _
     NewKey As Long) As Long
```

The Key and NewKey parameters above are Windows *handles*—a handle is a pointer. When you run the statement on 32-bit computers running Windows, Key and NewKey are 32-bit values and fit nicely into a Long variable. This method of coding is precisely what was done for years.

When you are using a 64-bit version of Office 2010, however, pointers and handles are 64-bit values. The big problem here is that the Long variable in Visual Basic is still 32 bits. Using our example here, consider what happens with NewKey, which is filled in by the API call. If Visual Basic allocates only 32 bits to hold NewKey and Windows thinks this is a 64-bit quantity, Windows overwrites the adjacent memory to NewKey, resulting in undefined behavior, including possibly crashing Visual Basic. For this reason, Visual Basic blocks this Declare statement from running on 64-bit Visual Basic until you properly update the statement.

INSIDE OUT

Setting a Registry Key to Test Upper Memory

You can set a registry key on your computer to force Windows to use memory allocations into the upper 32 bits of memory to help find instances in your application code where a Long was not properly upgraded to a LongPtr. You can find more information about this registry setting at the following page on Microsoft's website: *http://www.microsoft.com/whdc/system/platform/server/PAE/PAEdrv.mspx*.

Using LongPtr Data Types

Before we update the Declare statement, let's recall what got us into this situation: lack of an official pointer data type and the subsequent overloading of Long to be the unofficial pointer data type. If Visual Basic had a pointer data type from the beginning, we could have written that Declare statement properly so that it runs correctly on both 32-bit and 64-bit versions of Visual Basic.

In Office 2010, Microsoft introduces a new data type for Visual Basic to fill this role as the official pointer data type—LongPtr. With 32-bit Visual Basic, LongPtr is a 32-bit quantity; and with 64-bit Microsoft Visual Basic for Applications (VBA), LongPtr is a 64-bit quantity.

Now, we can rewrite the previous Declare statement, which will work with both 32-bit and 64-bit Visual Basic, as follows:

```
Declare Function RegOpenKeyA Lib "advapi32.dll" _
    (ByVal Key As LongPtr, ByVal SubKey As String, _
     NewKey As LongPtr) As Long
```

You'll also need to update any Long sized pointers in User Defined Types (UDTs) that are passed to and from Declare procedures. Here's an example with the Windows process information structure:

```
Type PROCESS_INFORMATION
    hProcess As LongPtr
    hThread As LongPtr
    dwProcessId As Long
    dwThreadId As Long
End Type
```

You need to remember that all Windows handlers are pointers and therefore need to be accordingly increased in size in your code. You can use LongPtr to hold and pass handles.

Using PtrSafe Attributes

As you're following along, you might be noticing that we have another problem: How can Visual Basic tell if a Declare statement's parameters and return value have been properly updated to handle pointers and are safe to use with both 32-bit and 64-bit Visual Basic? The answer, of course, is that it cannot. There are many Declare statements that quite legitimately pass or return Long values that are not pointers.

To address this issue, Microsoft introduced a new attribute for Declare statements called PtrSafe. This attribute tells Visual Basic that the Declare statement you're writing is safe to run with both 32-bit and 64-bit Visual Basic and that all pointer-sized values are properly handled. Without the PtrSafe attribute, 64-bit Visual Basic does not compile Declare statements. To maintain backward compatibility, 32-bit Visual Basic continues to compile Declare statements without the PtrSafe attribute. So, using the example we've been working through previously, the Declare statement is as follows:

```
Declare PtrSafe Function RegOpenKeyA Lib "advapi32.dll" _
    (ByVal Key As LongPtr, ByVal SubKey As String, _
     NewKey As LongPtr) As Long
```

Microsoft recommends that all new Declare statements use LongPtr and PtrSafe.

Supporting Older Versions of Access

If you develop Access applications in Access 2010 for users with previous versions of Access, you still need to address one more issue. Versions of Visual Basic before Access 2010 do not understand the new data types and attributes and therefore generate errors. If the Visual Basic code you write for Access 2010 needs to run in previous versions of Access, then you must wrap the code in the new #If VBA7 construct and include the older format as well.

This new conditional compilation variable VBA7 is defined only for Visual Basic included with Access 2010. Using the example Declare statement we've been working on previously, you can write your code like the following:

```
#If VBA7 Then
    Declare PtrSafe Function RegOpenKeyA Lib "advapi32.dll" _
      (ByVal Key As LongPtr, ByVal SubKey As String, _
      NewKey As LongPtr) As Long
#Else
    Declare Function RegOpenKeyA Lib "advapi32.dll" _
      (ByVal Key As Long, ByVal SubKey As String, _
      NewKey As Long) As Long
#End If
```

Note that there is also a *Win64* conditional compilation variable, which can be useful if the Declare statement has other differences besides pointer size between 32-bit and 64-bit platforms. You write your code, in this case, like the following:

```
#If WIN64 Then
    Declare PtrSafe Function WindowFromPoint Lib "user32" Alias _
    "WindowFromPoint"(ByVal Point As LongLong) As LongPtr
#Else
    Declare PtrSafe Function WindowFromPoint Lib "user32" Alias _
    "WindowFromPoint"(ByVal xPoint As Long, ByVal yPoint As Long) As LongPtr
#End If
```

Many Access developers create Visual Basic code in their applications using Declare statements based on a text file called Windows API Declarations and Constants for Visual Basic—Win32API.txt—supplied by Microsoft. The Win32API.txt file includes around 1,500 examples of the most common Windows API calls. For Office 2010, Microsoft updated this reference file to include information about the new LongPtr data type and PtrSafe attribute, as outlined earlier. You can download this new reference file, called Win32API_PtrSafe.txt, from Microsoft's website at the following location:

http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=035b72a5-eef9-4baf-8dbc-63fbd2dd982b

Understanding Pointer Valued Functions and LongPtr Type Coercion

VarPtr, StrPtr, and ObjPtr are functions that return pointers to Visual Basic variables. You can use these functions to pass pointers to APIs, for example. In 32-bit Visual Basic, these functions return Long values as they always have, but when you use 64-bit VBA, these functions return LongPtr values. Although this seems correct on the surface, this can pose a problem with an unexpected result. Consider, for example, the following code:

```
Dim L as Long
Dim X as Long
L = VarPtr(X)
```

When you use 32-bit Visual Basic with the code sample here, your code should execute properly because L is a Long (32-bit) and VarPtr returns a Long (32-bit). However, if you use this code sample with 64-bit Visual Basic, L is a Long (32-bit), but VarPtr is a LongPtr (64-bit). In this case, we are assigning a possibly large value into a smaller variable. In most cases, Visual Basic handles this kind of conversion at run time. If the return value from VarPtr fits in 32 bits, then Visual Basic silently does the downsizing and you won't see a run-time error. Because pointer values are unpredictable, however, you might never encounter a run-time error while developing the application, and end users might only sporadically see a run-time error. These types of errors are hard to reproduce and very difficult to debug. To help alleviate this potential issue, 64-bit Visual Basic does not allow a LongPtr to be implicitly converted into a Long, even if the value being converted fits in the smaller variable. A LongPtr value can be explicitly converted by using the CLong() function.

Using LongLong Data Types

Besides LongPtr, Access 2010 Visual Basic includes another new data type called LongLong. The LongLong data type can hold an 8-byte signed integer value. The LongLong data type is useful when you are interacting with APIs on a computer running 64-bit Windows that consume or return 64-bit values. The LongLong data type is available only with the 64-bit version of Visual Basic. If you to use the LongLong data type in a 32-bit version of Access 2010, you'll receive a compile error. Table 24-7 shows a summary of the VBA7 language updates.

Table 24-7 Summary of VBA7 Language Updates

Name	Type	Description
PtrSafe	Keyword	Asserts that a Declare statement is targeted for 64-bit systems. This is required on 64-bit systems.
LongPtr	Data Type	Type alias that maps to Long on 32-bit systems or LongLong on 64-bit systems.
LongLong	Data Type	8 byte data type that is available only on 64-bit systems. Supports integer numbers in the range of –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. LongLong is a valid declared type only on 64-bit systems. In addition, you cannot implicitly convert a LongLong to a smaller type. For example, you cannot assign a LongLong data type to a Long. Explicit coercions are allowed, so in the previous example, you could apply CLng to a LongLong and assign the result to a Long on 64-bit platforms.

Name	Type	Description
^	LongLong type-declaration character	Explicitly declares a literal value as a LongLong. This is required to declare a LongLong literal that is larger than the maximum Long value. If you don't explicitly declare the value, Access converts it to a Double.
CLngPtr	Type conversion function	Converts a simple expression to a LongPtr. This is valid on 64-bit platforms only.
CLngLng	Type conversion function	Converts a simple expression to a LongLong data type. This is valid on 64-bit platforms only.
vbLongLong	VarType constant	Constant used with the VarType function. Note that there is no vbLongPtr, since LongPtr is a mapping to Long and LongLong and therefore is not really a separate type.
DefLngPtr	DefType statement	Sets the default data type for a range of variables as LongPtr.
DefLngLng	DefType statement	Sets the default data type for a range of variables as LongLong.

Working with .MDE and .ACCDE files in 64-Bit Environments

As you'll learn in Chapter 27, "Distributing Your Application," on the companion CD, .mde and .accde files are execute-only Access databases in which the Visual Basic source code (the text that the developer edits) is removed. The binary executable form of the Visual Basic project remains, however, which allows Visual Basic to continue executing in these databases. Without the source code, the Visual Basic project cannot be read or modified by another developer. Many Access developers use this feature included in Access to help protect their intellectual property.

Unfortunately, the binary executable form of Visual Basic is not compatible between 32-bit and 64-bit versions of Visual Basic. Normally, this is not a problem when you are using an .mdb or .accdb database; Visual Basic is designed to recompile from source code if it finds the binary executable form stored within the database is the wrong type. For .mde and .accde databases, however, this presents a problem because .mde and .accde files have no source code. As a result, .mde and .accde databases created with 32-bit Access 2010 can only be used with 32-bit versions of Access, and .mde and .accde databases created with 64-bit Access 2010 can only be used with 64-bit versions of Access. If you are distributing an application that needs to be run with both 32-bit and 64-bit versions of Access, you must create and distribute separate 32-bit and 64-bit .mde and .accde databases.

Note

The controls in the MSComCtl and MSComCtl2 libraries were not ported to 64-bit in Office 2010. You cannot use any of the following controls in a 64-bit environment of Office 2010:

- **MSComCtl Control Library: TabStrip, Toolbar, StatusBar, ProgressBar, TreeView, ListView, ImageList, Slider, and ImageComboBox**
- **MSComCt2 Control Library: Animation, UpDown, MonthView, DateTimePicker, and FlatScrollBar**

You should now have a basic understanding of how to create functions and subroutines using Visual Basic. In Chapter 25, you'll enhance what you've learned as you study major parts of the Conrad Systems Contacts, Housing Reservations, and Wedding List applications.