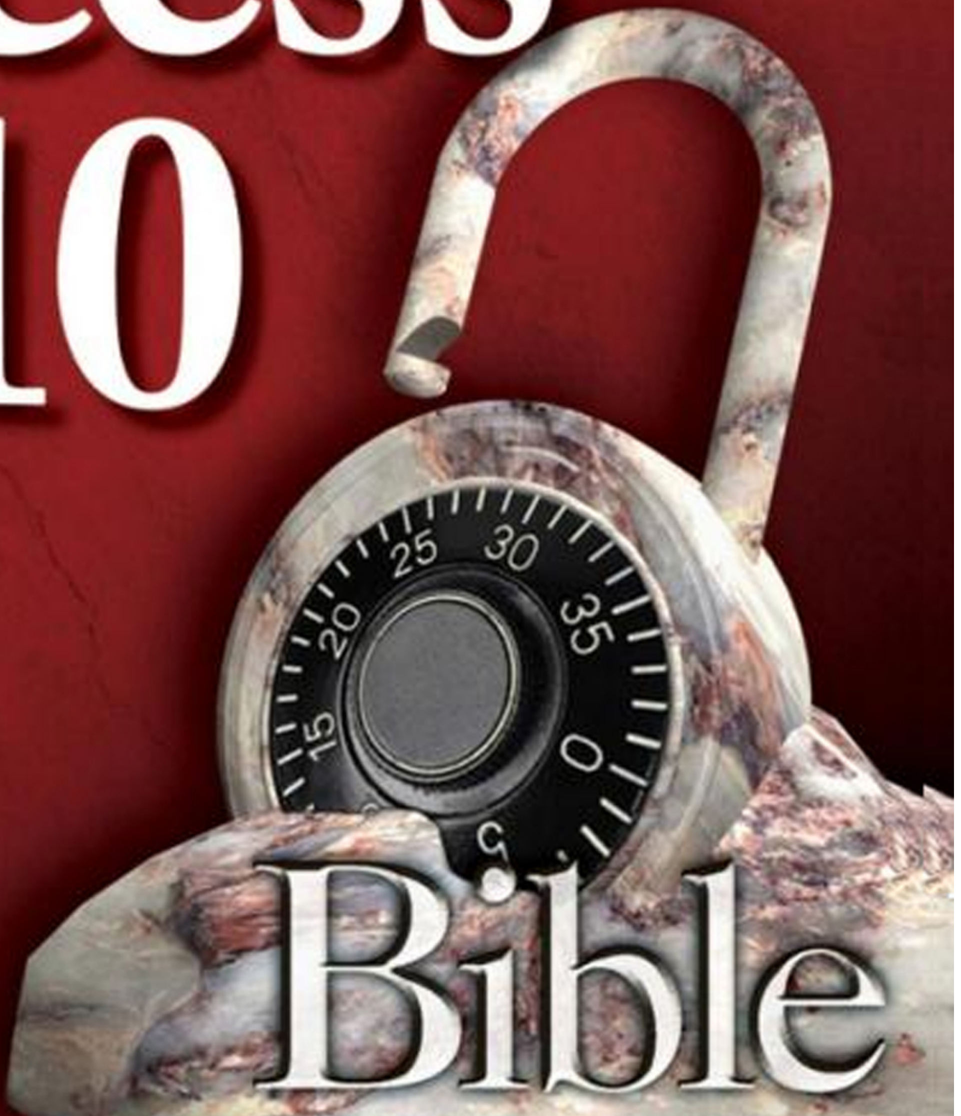
**CD-ROM Included!**
- Examples from the book
- Searchable PDF of the book

Michael R. Groh

# Microsoft® Access® 2010 Bible



**Create** tables and design bulletproof databases

**Integrate** Access with other applications

**Organize,** view, analyze, and share data

The book you need to succeed!

# Table of Contents

CHAPTER 10

# VBA Programming Fundamentals

**IN THIS CHAPTER**

Most Access developers use macros now and then. Although macros provide a quick and easy way to automate an application, writing Visual Basic for applications (VBA) modules is the best way to create applications. VBA provides data access, looping and branching, and other features that macros simply don't support — or at least don't support with the flexibility most developers want. In this chapter, you learn how to use VBA to extend the power and usefulness of your applications.

## On the CD-ROM

**Use the database file `Chapter10.accdb` in this chapter.**

## Note

**Although many readers of this book are experienced Access developers and are comfortable working with VBA, this chapter and the other chapters in this part of the book assume that you have no experience with VBA. I include these chapters to provide you with a firm foundation for many of the techniques I discuss in later chapters. As you'll see in Part III and beyond, many advanced Access techniques simply can't be implemented without the use of VBA code.**

### The limitations of macros

For a number of reasons, this book doesn't extensively cover Access macro creation. To begin with, there are enough important topics that I had to choose which topics to cover in detail. Plus, macros are pretty easy to learn on your own and they're well documented in the Access online help. There are, however, two areas where macros can't be beat: data macros in tables and embedded macros on forms and controls. The ability to embed macros in tables and forms make macros much more attractive than in versions prior to Access 2007.

But, by far, the biggest reason I don't document macros is that macros are guaranteed to be non-portable to other applications. You can't use an Access macro anywhere other than in Access. VBA code, on the other hand, is very portable to Word, Excel, Outlook, Visio, and even Visual Studio .NET (with changes).

It's impossible to tell where an Access application might end up. Very often, Access apps are upsized and upgraded to SQL Server and Visual Studio .NET. The VBA code in your Access applications is readily converted to Visual Basic .NET, and many Access procedures can be used (perhaps with a few changes) in Word or Excel. VBA is a very portable, useful language, and VBA skills are applicable in many situations other than building Access applications.

I don't mean to imply that macros have no place in Access applications, or that macros are necessarily inferior to VBA code. Microsoft has issues related to previous versions of Access macros. In particular, macros in Access 2007 and 2010 include variables and simple error handling (mostly jumping to a named location when an error occurs). These updates to the Access macro engine are significant, but, in the opinion of many Access developers, they aren't enough to justify using macros instead of VBA in professional applications.

## Introducing Visual Basic for Applications

Visual Basic for Applications (VBA) is the programming language built into Microsoft Access. VBA is shared among all the Microsoft Office applications, including Word, Excel, Outlook, PowerPoint, and even Visio. If you aren't already a VBA programmer, learning the VBA syntax and how to hook VBA into the Access event model is a definite career builder.

VBA is a key element in most professional Microsoft Access applications. Microsoft provides VBA in Access because VBA provides significant flexibility and power to Access database applications. Without a full-fledged programming language like VBA, Access applications would have to rely on the somewhat limited set of actions offered by Access macros. Although macro programming also adds flexibility to Access applications, VBA is much easier to work with when you're programming complex data-management features or sophisticated user-interface requirements.

### Cross-Reference

**If you want more information on macros, turn to Chapter 30.**

## What's in a name?

The name Visual Basic is a source of endless confusion for people working with the Microsoft products. Microsoft has applied the name Visual Basic to a number of different products and technologies. For more than a decade, Microsoft marketed a stand-alone product named Visual Basic that was, in many ways, comparable to and competitive with Microsoft Access. Visual Basic was folded into Visual Studio in its very first version. In 1995, Microsoft added the Visual Basic for Applications (VBA) programming language to Access, Word, and Excel in Microsoft Office. The name Visual Basic for Applications was chosen because the VBA syntax is identical in Access, Word, and Excel.

Although the VBA language used in Access is very similar to Visual Basic .NET, they aren't exactly the same. You can do some things with VB .NET that can't be done with Access VBA, and vice versa.

In this book, the expressions "VBA" and "Visual Basic" refer to the programming language built into Access and should not be confused with the Microsoft VB .NET product.

If you're new to programming, try not to become frustrated or overwhelmed by the seeming complexity of the VBA language. As with any new skill, you're much better off approaching VBA programming by taking it one step at a time. You need to learn exactly what VBA can do for you and your applications, along with the general syntax, statement structure, and how to compose procedures using the VBA language.

This book is chock-full of examples showing you how to use the VBA language to accomplish useful tasks. Each of the procedures you see in this book has been tested and verified to work correctly. If you find that a bit of code in this book doesn't work as expected, take the time to ensure that you've used the example code exactly as presented in this book. Very often, the most difficult problems implementing any programming technique stem from simple errors, such as misspelling or forgetting to include a comma or parentheses where required.

### Note

**A programming language is much like a human language. Just as humans use words, sentences, and paragraphs to communicate with one another, a computer language uses words, statements, and procedures to tell the computer what you expect it to do. The primary difference between human and computer languages is that a computer language follows a very strict format. Every word and sentence must be precisely composed because a computer doesn't understand context or nuance. Every task must be carefully defined for the computer, using the syntax supported by the programming language.**

## Understanding VBA Terminology

Before you plunge into my VBA coverage, here's a review of some basic VBA terminology:

- **Keyword:** A word that has special meaning in VBA. For example, in the English language, the word *now* simply indicates a point in time. In VBA, `Now` is the name of a built-in VBA function that returns the current date and time.

- **Statement:** A single VBA word or combination of words that constitutes an instruction to be performed by the VBA engine.
- **Procedure:** A collection of VBA statements that are grouped together to perform a certain task. You might, for example, write a complex procedure that extracts data from a table, combines the data in a particular way, and then displays the data on a form. Or, you might write three smaller procedures, each of which performs a single step of the overall process.

  There are two types of VBA procedures: *subs* (subroutines) and *functions:*

  - **Subroutines** perform a single task and then just go away.
  - **Functions** perform a task and then return a value, such as the result of a calculation.

  The example described earlier, where the procedure extracts data from a table, is actually a subroutine. It performs a specific task; then, when it ends, the procedure just goes away.

  The example where the operation is split into three smaller procedures includes a function. In this case, the first procedure that opens the database and extracts data most likely returns the data as a recordset, and the recordset is passed to the other procedures that perform the data combination and data display.
- **Module:** Procedures live in *modules.* If statements are like sentences and procedures are like paragraphs, modules are the chapters or documents of the VBA language. A module consists of one or more procedures and other elements combined as a single entity within the application.
- **Variable:** Variables are sometimes tricky to understand. Because Access is a database development tool, it makes sense that VBA code has to have some way of managing the data involved in the application. A variable is nothing more than a name applied to represent a data value. In virtually all VBA programs, you create and use variables to hold values such as customer names, dates, and numeric values manipulated by the VBA code.

VBA is appropriately defined as a *language.* And, just as with any human language, VBA consists of a number of words, sentences, and paragraphs, all arranged in a specific fashion. Each VBA sentence is a *statement*. Statements are aggregated as *procedures,* and procedures live within *modules.* A *function* is a specific type of procedure — one that returns a value when it's run. For example, `Now()` is a built-in VBA function that returns the current date and time, down to the second. You use the `Now()` function in your application whenever you need to capture the current date and time, such as when assigning a timestamp value to a record.

## Starting with VBA Code Basics

Each statement in a procedure is an instruction you want Access to perform.

There are, literally, an infinite number of different VBA programming statements that could appear in an Access application. Generally speaking, however, VBA statements are fairly easy to read and understand. Most often, you'll be able to understand the purpose of a VBA statement based on the keywords (such as `DoCmd.OpenForm`) and references to database objects in the statement.

Each VBA statement is an instruction that is processed and executed by the VBA language engine built into Microsoft Access. Here's an example of a typical VBA statement that opens a form:

```
DoCmd.OpenForm "frmMyForm", acNormal
```

Notice that this statement consists of an action (`OpenForm`) and a noun (`frmMyForm`). Most VBA statements follow a similar pattern of action and a reference either to the object performing the action or to the object that's the target of the action.

`DoCmd` is a built-in Access object that performs numerous tasks for you. Think of `DoCmd` as a little robot that can perform many different jobs. The `OpenForm` that follows `DoCmd` is the task you want `DoCmd` to run, and `frmMyForm` is the name of the form to open. Finally, `acNormal` is a modifier that tells `DoCmd` that you want the form opened in its "normal" view. The implication is that there are other view modes that may be applied to opening a form; these modes include Design (`acDesign`) or Datasheet (`acFormDS`) view, and Print Preview (`acPreview`, when applied to reports).

### Note

**Writing programming statements and code usually requires more work than the alternatives, such as using macros or the Access Command Button Wizard. Very often, the VBA code you write refuses to work as expected. And, sometimes the things you want to do with VBA are just too difficult or might be impossible to accomplish with VBA. At the same time, VBA programming skills are a great addition to a developer's résumé. The ability to efficiently program complex business rules or use code to clean up data before they're added to a database are valuable skills.**

**Although this and the following chapters provide only the fundamentals of VBA programming, you'll learn more than enough to be able to add advanced features to your Access applications. You'll also have a good basis for deciding whether you want to continue studying this important programming language.**

## Migrating from Macros to VBA

Should you now convert all the macros in your applications to VBA? The answer depends on what you're trying to accomplish. The fact that Access includes VBA doesn't mean that Access macros are no longer useful; it simply means that Access developers should learn VBA and add it to their arsenal of tools for creating Access applications.

VBA isn't always the answer. Some tasks, such as creating global key assignments, can be accomplished only via macros. You can perform some actions more easily and effectively by using a macro than by writing VBA code.

### Note

**An Access macro is a stepwise list of actions that you compose using the Access macro editor. Microsoft Word and Excel use the word macro to refer to procedures written in the VBA programming language, often through the use of the Word or Excel macro recorder. When working within the Access environment, a macro always refers to the stepwise set of instructions composed using the Access macro editor. Most Access developers refer to VBA code as either procedures or modules, and virtually never refer to these objects as macros.**

## Knowing when to use macros and when to use VBA

In Access, macros often offer a great way to take care of many details, such as opening reports and forms. Macros can usually be created very quickly because the arguments for each macro action are displayed in the macro editor. You don't have to remember complex or difficult syntax.

You can accomplish many things with the VBA code and with macros:

- **Create and use your custom functions.** In addition to using built-in Access functions, VBA enables you to create and work with your own reusable functions.
- **Respond to errors.** Both macros and VBA handle errors quite well. However, macros are limited to jumping to a set of macro actions in response to the error, while VBA error handlers can examine what caused the error, take corrective action, and repeat the statement(s) that caused the error. Macro error handling is quite good, but not quite as strong as when working with VBA code.
- **Use automation to communicate with other Windows applications.** You can write VBA code to see whether a file or some data or value exists before you take some action, or you can communicate with another Windows application (such as a spreadsheet), passing data back and forth.
- **Use the Windows Application Programming Interface (API).** VBA enables you to hook into many resources provided by Windows, such as determining the user's Windows login name, or the name of the computer the user is working on. The Windows API provides a virtually unlimited number of ways to enhance your Access applications.

**Cross-Reference**

**Turn to Chapter 27 for more on the Windows API.**

- **Maintain the application.** Unlike macros, code can be built into a form or report, making maintaining the form or report more efficient. Plus, if you move a form or report from one database to another, the event procedures built into the form or report travel with it.
- **Create or manipulate objects.** In most cases, you'll find that you need to work with an object in Design view. In some situations, however, you might want to manipulate the definition of an object in code. Using VBA, you can manipulate all the objects in a database, including the database itself.

Access supports embedded macros in forms, reports, and controls. An embedded macro lives within its host object (form, report, or control), and travels with the object if it is copied to another Access database. This is a huge improvement over the old Access macro model where it was very difficult to know which macros were related to which forms and reports. Even then, however, embedded macros suffer from the performance issues associated with external macros, and they aren't portable to any other applications, like Word or Excel.

## Converting your existing macros to VBA

As you become comfortable with writing VBA code, you might want to rewrite existing macros as VBA procedures. As you begin this process, you quickly realize how challenging the effort can be as you review every macro in your macro libraries. You can't cut and paste a macro into a VBA code module. You have to analyze the task accomplished by each macro action, and then add the equivalent VBA statements to your code.

Fortunately, Access provides a feature to convert macros to VBA code automatically. One of the options in the Save As dialog box is Save As Module. You can use this option when a macro file is highlighted in the Macros object area of the Solution Explorer. This option enables you to convert an entire macro group to a VBA module in seconds.

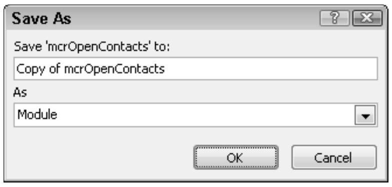
To try the conversion process, save the `mcrOpenContacts` macro in the `Chapter10.accdb` database as a module by following these steps:

1. Open the Macros section of the Navigation Pane, and s elect the `mcrOpenContacts` macro.
2. Click the File button in the upper-left corner of the main Access window to open the Backstage, and select Save As from the command list.

   The Save As dialog box appears, as shown in Figure 10.1.

**FIGURE 10.1**

Saving a macro as a module


Save As

Save 'mcrOpenContacts' to:

Copy of mcrOpenContacts

As

Module

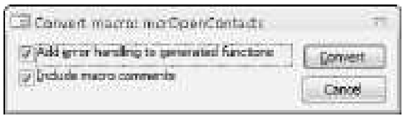
3. Select Module from the drop-down list on the Save As dialog box.

   Access assigns a default name for the new module as `Copy of` followed by the macro name. Oddly enough, when you save a macro as a VBA module, Access doesn't use the name displayed in the Save As dialog box, nor does it use any name that you enter in the Save As dialog box.

   The Convert Macro dialog box appears, as shown in Figure 10.2.

**FIGURE 10.2**

The Convert Macro dialog box


Convert macro: mcrOpenContacts

[✓] Add error handling to generated functions

[✓] Include macro comments

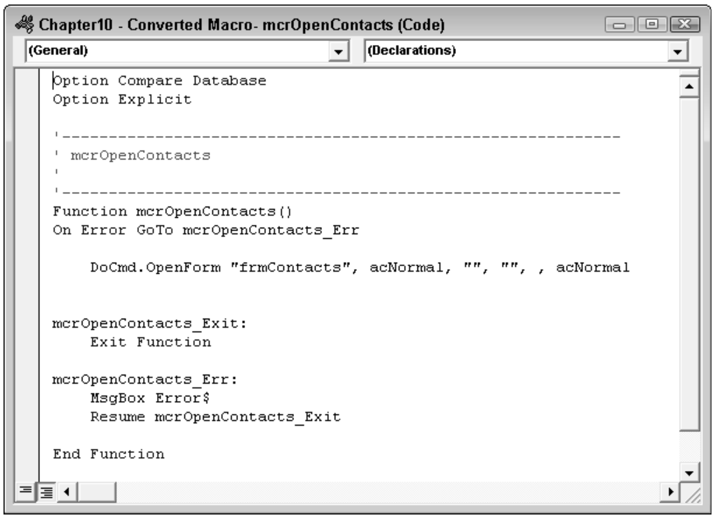
4. Select the options that include error handling and comments, and click Convert.

   Access briefly displays each new procedure in the VBA editor window as it's converted. When the conversion process completes, the `Conversion Finished!` message box appears.

Figure 10.3 shows the newly created VBA module, Converted Macro–mcrOpenContacts. Notice that the module and the function within the module are named after the converted macro, regardless of any other name you might have entered in the Save As dialog box.

**FIGURE 10.3**

The converted module



```
Option Compare Database
Option Explicit

'------------------------------------------------------------
' mcrOpenContacts
'
'------------------------------------------------------------
Function mcrOpenContacts()
On Error GoTo mcrOpenContacts_Err

    DoCmd.OpenForm "frmContacts", acNormal, "", "", , acNormal


mcrOpenContacts_Exit:
    Exit Function

mcrOpenContacts_Err:
    MsgBox Error$
    Resume mcrOpenContacts_Exit

End Function
```

When you specify that you want Access to include error processing for the conversion, Access automatically inserts an `On Error` statement as the first statement in the procedure, telling Access to branch to an error handler that displays an appropriate message and then exit the function.

## Cross-Reference

**Handling errors is covered in Chapter 23.**

The statement beginning with `DoCmd` is the code that Access created from the macro's actions. `DoCmd` methods mimic macro actions and perform important tasks such as opening forms and reports and setting the values of controls.

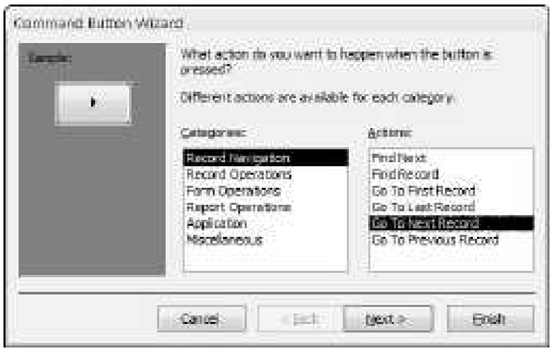
## Using the Command Button Wizard

One option when adding a button to an Access form is to use the Command Button Wizard. When Access creates a command button with a wizard, it adds an embedded macro attached to the button. The embedded macro performs whatever action (open form, open report, and so on) that you specified when you worked with the wizard. You can open the embedded macro in the macro editor (described in Chapters 15 and 30) and modify it to fit your needs.

Access supports more than 30 types of command buttons through the Command Button Wizard. These buttons include finding or printing records, as well as applying a filter to a form's data. Run this wizard by adding a command button to a form with the Use Control Wizards option selected in the Design tab of the Access ribbon. (You have to expand the controls palette to see the Use Control Wizards option.) Figure 10.4 shows a Go To Next Record command button being created. Notice that the Command Button Wizard even shows a preview of the image it selects for the button in the panel at the left side of the wizard dialog box.

FIGURE 10.4

The Command Button Wizard



What action do you want to happen when the button is pressed?

Different actions are available for each category.

Categories:
- Record Navigation
- Record Operations
- Form Operations
- Report Operations
- Application
- Miscellaneous

Actions:
- Find Next
- Find Record
- Go To First Record
- Go To Last Record
- Go To Next Record
- Go To Previous Record

The `Chapter10.accdb` example database includes a form named `frmButtonWizardSamples_Macros`. This form, shown in Figure 10.5 in Design mode, contains a dozen command buttons created with the Command Button Wizard. Review the procedures for the buttons on this form to see how powerful Access macros can be. The buttons on this form don't actually do anything because there is no data on the form, and the macros have nothing to actually work with, but they show the variety of actions supported by the command button wizard.

Figure 10.6 shows the code for the Go To First Record command button.

The macro produced by the Command Button Wizard is very simple but effective.

FIGURE 10.5

Examples of Command Button Wizard buttons

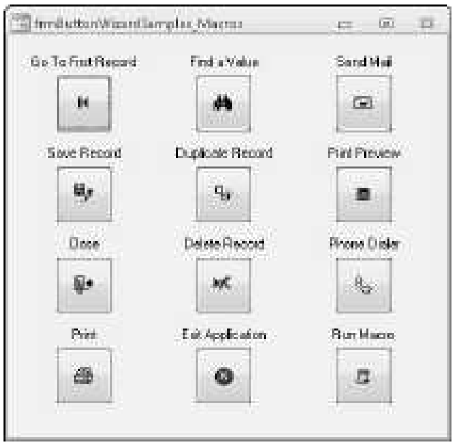
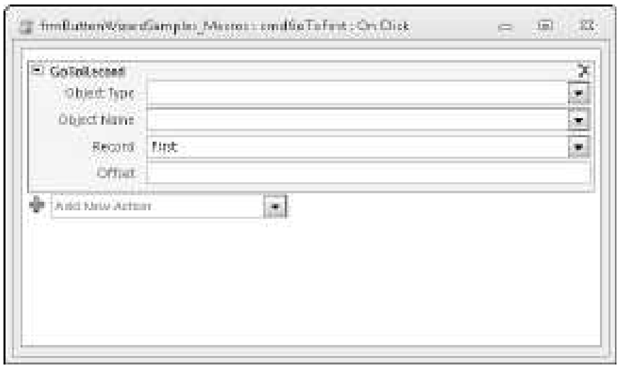


FIGURE 10.6

The Go To First Record button's `On Click` procedure


frmButtonWizardSamples_Macros : cmdGoToFirst : On Click

GoToRecord

| | |
|---|---|
| Object Type | |
| Object Name | |
| Record | First |
| Offset | |

## Creating VBA Programs

Access has a wide variety of tools that enable you to work with tables, queries, forms, and reports without ever having to write a single line of code. At some point, you might begin building more sophisticated applications. You might want to "bulletproof" your applications by providing more intensive data-entry validation or implementing better error handling.

Some operations can't be accomplished through the user interface, even with macros. You might find yourself saying, "I wish I had a way to . . ." or "There just has to be a function that will let me. . . ." At other times, you find that you're continually putting the same formula or expression in a query or filter. You might find yourself saying, "I'm tired of typing this formula into . . ." or "Doggone it, I typed the wrong formula in this. . . ."

For situations such as these, you need the horsepower of a high-level programming language such as VBA. VBA is a modern, structured programming language offering many of the programming structures available in most programming languages. VBA is *extensible* (capable of calling Windows API routines) and can interact through ActiveX Data Objects (ADO), through Data Access Objects (DAO), and with any Access or VBA data type.

Getting started with VBA programming in Access requires an understanding of its event-driven environment.

## Events and event procedures

In Access, unlike old-fashioned programming environments, the user controls the actions and flow of the application. The user determines what to do and when to do it, such as changing information in a field or clicking a command button. The user determines the flow of action and, through *events,* the application determines what action to take or ignore.

In contrast, procedural programming languages require that the programmer determine the flow of actions that the user must follow. In fact, the programmer must accommodate all possibilities of user intervention — for example, keystrokes a user might enter in error — and must determine what actions to take in response to the user.

Using macros and event procedures, you implement the responses to these actions. Access provides *event properties* for each control you place on a form. By attaching a VBA procedure to a control's event property, you don't have to worry about the order of actions a user might take on a particular form.

In an event-driven environment such as Access, the objects (forms, reports, and controls) respond to events. Basically, an event procedure is VBA code that executes when an event (such as a button click) occurs. The code is directly attached to the form or report containing the event being processed. An Exit command button, for example, closes the form when the user clicks the button. Clicking the command button triggers its `Click` event. The event procedure is the VBA code attached to the `Click` event. The event procedure automatically runs every time the user clicks the command button.

There are two types of procedures: subprocedures (often called subs) and functions.

Sub and function procedures are grouped and stored in *modules.* The Modules object button in the Navigation Pane contains the common global, or *standard,* modules that any form or report can access. You *could* store all your procedures in a single module, but that wouldn't be a good idea. You'll probably want to group related procedures into separate modules, categorizing them by the nature of the operations they perform. For example, an `Update` module might include procedures for adding and deleting records from a table.

### Subprocedures

A subprocedure (or *sub*) is the simplest type of procedure in a VBA project. A subprocedure is nothing more than a container for VBA statements that typically perform a task such as opening a form or report or running a query. The code in a subprocedure simply executes and then goes away without leaving a trace other than whatever work it performed.

All Access event procedures are subs. Clicking on a command button triggers the button's `Click` event, for example.

These VBA statements within a sub are the code you want to run every time the procedure is executed. The following example shows an Exit command button's subprocedure:

```
Sub cmdExit_Click()
  DoCmd.Close
End Sub
```

The first line of this procedure notifies the VBA engine that the procedure is a `sub` and that its name is `cmdExit_Click`. If *parameters* (data passed to the procedure) are associated with this sub, they appear within the parentheses.

There is only one VBA statement within this sub: `DoCmd.Close`. The `End Sub` statement at the bottom ends this procedure. The `cmdExit_Click ()` subprocedure is attached to the Exit button's `Click` event. The event procedure closes the form when the user clicks the Exit command button.

### Functions

A function is very similar to a subprocedure with one major exception: A function returns a value when it ends. A simple example is the built-in VBA `Now()` function, which returns the current date and time. `Now()` can be used virtually anywhere your application needs to use or display the current date and time. An example is including `Now()` in a report header or footer so that the user knows exactly when the report was printed.

`Now()` is just one of several hundred built-in VBA functions. As you'll see throughout this book, the built-in VBA functions provide useful and very powerful features to your Access applications.

In addition to built-in functions, you might add custom functions that perform tasks required by your applications. An example is a data transformation routine that performs a mathematical operation (such as currency conversion or calculating shipping costs) on an input value. It doesn't matter where the input value comes from (table, form , query, and so on). The function always returns exactly the correct calculated value, no matter where the function is used.

Within the body of a function, you specify the function's return value by assigning a value to the function's name (and, yes, it does look pretty strange to include the function's name within the function's body). You then can use the returned value as part of a larger expression. The following function calculates the square footage of a room:

```
Function nSquareFeet(Height As Double, _
    Width As Double) As Double
  'Assign this function's value:
  nSquareFeet = Height * Width
End Function
```

This function receives two parameters: `Height` and `Width`. Notice that the function's name, `nSquareFeet`, is assigned a value within the body of the function. The function is declared as a Double data type, so the return value is recognized by the VBA interpreter as a Double.

The main thing to keep in mind about functions is that they return values. The returned value is often assigned to a variable or control on a form or report:

```
dblAnswer = nSquareFeet(Height, Width)
txtAnswer = nSquareFeet(Height, Width)
```

If the function (or subroutine, for that matter) requires information (such as the `Height` and `Width` in the case of the `nSquareFeet function`) the information is passed as arguments within the parentheses in the function's declaration.

You'll read much more about subroutines and functions in the remaining chapters of this book. You'll also see many more examples of passing arguments to procedures, returning values from functions, and using subs and functions to perform complicated work in Access applications.

One last point: Notice the underscore character at the end of the `nSquareFeet` function's declaration. A space followed by an underscore at the end of a VBA statement (called a *continuation character*) instructs the VBA engine to include the next line as part of the same statement.

## Modules

Modules and their procedures are the principal objects of the VBA programming language. The code that you write is added to procedures that are contained in *modules.*

### Looking at the types of modules

There are two types of VBA code modules in most Access applications. The first type is a free-standing, independent module that might be accessed by any other object (such as forms and reports) within the application. These modules are often called *standard* or *global* modules. The second type of module exists with (or behind, if you prefer) the forms and reports in an application and is normally accessible only to the form or report and its controls. This second type of module is usually referred to as *form* and *report modules*.

As you create VBA procedures for your Access applications, you use both types of modules.

**Cross-Reference**

**In addition to standard and the modules behind form and reports, Access supports a third type of VBA code module. Chapter 28 discusses class modules, and the object-oriented programming techniques supported by all versions of Access since Access 97. Object-oriented programming is an important concept and can lead to simplified programming and efficient code-reuse.**

### *Standard modules*

Standard modules are independent from forms and reports. Standard modules store code that is used from anywhere within your application. By default, these procedures are often called *global* or *public* because they're accessible to all elements of your Access application.

Use public procedures throughout your application in expressions, macros, event procedures, and other VBA code. To use a public procedure, you simply reference it from VBA code in event procedures or any other procedure in your application.

## Tip

**Procedures run; modules contain. Procedures are executed and perform actions. Modules, on the other hand, are simple containers, grouping procedures and declarations together. A module can't be run; instead, you run the procedures contained within the module.**

Standard modules are stored in the Module section of the Navigation Pane. Form and report modules (see the next section) are attached to their hosts and are accessed through the Form Property Sheet or Report Property Sheet.

## Tip

**Generally speaking, you should group related procedures into modules, such as putting all of an application's data conversion routines into a single module. Logically grouping procedures make maintenance much easier because there is a single place in the application for all the procedures supporting a particular activity. Plus, most modules contain procedures that are related in some way.**

### *Form and report modules*

All forms and reports support events. The procedures associated with form and report events can be macros or VBA code. Every form or report you add to your database contains a VBA code module (unless its `Has Module` property is set to `No`). This form or report module is an integral part of the form or report; it's used as a container for the event procedures you create for the form or report. The module attached to each form or report is a convenient way to place all the object's event procedures in a single location.

## Cross-Reference

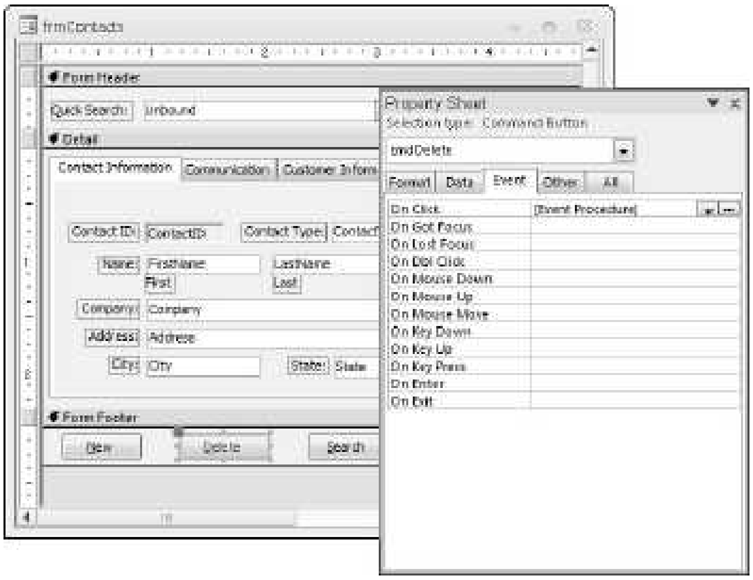
**For more on events, turn to Chapter 12.**

Adding VBA event procedures to a form module is very powerful and efficient. Because the module is an integral and permanent part of the form or report, all the event procedures travel with the object when it's exported to another Access database.

Modifying a control's event procedure is easy: Simply click the builder button (with the ellipsis: . . .) in the Property Sheet next to the event property name, to open the form's code module. Figure 10.7 illustrates accessing the `Click` event procedure of the Delete button on the Contacts form.

**FIGURE 10.7**

Accessing a control's event procedure from the Property Sheet



Selection type: Command Button

cmdDelete

| | |
|---|---|
| On Click | [Event Procedure] |
| On Got Focus | |
| On Lost Focus | |
| On Dbl Click | |
| On Mouse Down | |
| On Mouse Up | |
| On Mouse Move | |
| On Key Down | |
| On Key Up | |
| On Key Press | |
| On Enter | |
| On Exit | |

Notice the `[Event Procedure]` in the control's `On Click` property. It tells you that there is code attached to the control's `Click` event. Clicking on the builder button (with the ellipsis, or . . .) in the `On Click` property opens a dialog box where you can choose to build a macro or VBA code, or create an expression for the event. Choosing the Code Builder option opens the VBA code editor, displaying the event procedure.

## Tip

**Many Access developers prefer to routinely use VBA code for their procedures. You can instruct Access to always use VBA code by selecting the Always Use Event Procedures check box in the Form/Report Design view section under Object Designers in the Access Options screen (found in the Backstage).**

## Caution

**Event procedures that directly work with a form or report belong in the module of the form or report. A form's module should contain only the declarations and event procedures needed for that form and its controls (buttons, check boxes, labels, text boxes, combo boxes, and so on). Placing procedures shared with other forms in a form's module really doesn't make sense and should be avoided.**

### Creating a new module

Using the Modules section of the Navigation Pane, you create and edit VBA code contained in standard modules. You could, for example, create a `Beep` procedure that makes the computer beep as a warning or notification that something has happened in your program.

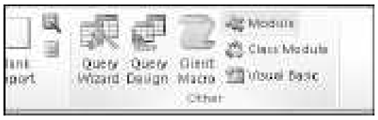
Think of a module as a collection of procedures. Your Access databases can contain thousands of modules, although most Access applications include only a few dozen standard modules.

## Cross-Reference

**You'll see many examples of creating functions and procedures in Chapters 11 through 14.**

For this example, you can use the `Chapter10.accdb` database or open a new blank database. Add a new module by selecting the Create tab of the Access ribbon, and then clicking on the Module button in the Other ribbon group (see Figure 10.8).

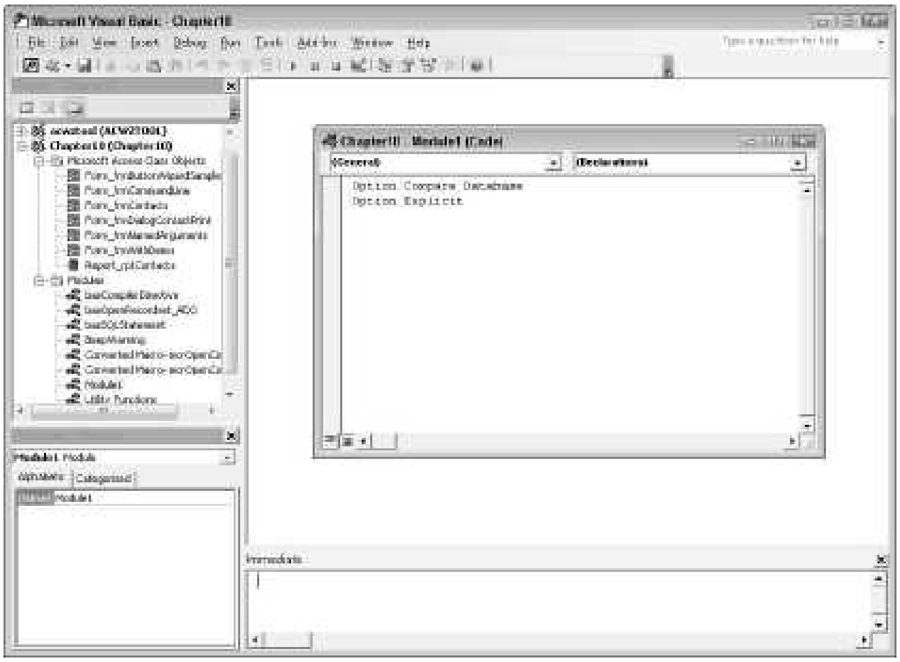
**FIGURE 10.8**



Adding a new module to an Access database

Access opens the VBA editor and adds a new module with a default name (see Figure 10.9).

**FIGURE 10.9**


Microsoft Visual Basic - Chapter10
File Edit View Insert Debug Run Tools Add-Ins Window Help
Type a question for help
Chapter10 - Module1 (Code)
(General)
(Declarations)
Microsoft Access Class Objects
Modules
Module1
Utility Functions
Module1 Module
Alphabetic
Categorized
Immediate

```
Option Compare Database
Option Explicit
```

The newly opened module in the VBA editor

### *Working in the code window*

Whenever you work on VBA procedures in your Access applications, you edit each module in a separate editor window. Although the code window may be confusing at first, it's easy to understand and use after you learn how each part works.

## Note

**Notice that the Access code window doesn't use the ribbon. Instead, the code window appears much as it has in every version of Access since Access 2000. Therefore, in this book you'll see references to the code window's toolbar and menu whenever I describe working with Access VBA modules. Don't confuse references to the code editor's toolbar with the main Access window's ribbon.**

When you enter Design mode of a module — whether it's via a form or report module or the Modules group on the Navigation Pane — the VBA editor and its menu and toolbar open to enable you to create or edit your procedures.

When you display VBA code within a form (or report) module, the Object and Procedure drop-down lists at the top of the code window contain the form's controls and events. You select these objects and events to create or edit event procedures for the form. Form and report modules can also include procedures that are not related to a control's events.

The Object drop-down list for a standard module offers only one choice: General. The Procedure drop-down list contains only the names of existing procedures within the standard module.

The code window's toolbar (shown in Figure 10.9) helps you create new modules and their procedures quickly. The toolbar contains buttons for the most common actions you use to create, modify, and debug modules.

The code window — the most important area of the VBA editor — is where you create and modify the VBA code for your procedures. The code window has the standard Windows features to resize, minimize, maximize, and move the window.

## Tip

**You can split the code window into two independent edit panes by dragging down the splitter bar (the little horizontal bar at the very top of the vertical scroll bar at the right edge of the code window). Splitting the window enables simultaneous editing of two sections of code. Each section of a split VBA code window scrolls independently, and changes you make in one pane of a split window show up in the other pane. Double-click the splitter bar to return the window to its former state, or grab the splitter bar with the mouse and drag it to the top of the code editor window to close the second edit pane. (Microsoft Word and Excel feature a similar splitter button, making it very easy to edit different parts of the same Word document or Excel worksheet.)**

The Immediate window (shown at the bottom of Figure 10.9) enables you to try a procedure while you're still in the module. See the "Checking your results in the Immediate window" section later in this chapter for an example.

## Cross-Reference

**You'll read much more about the Immediate window and the other debugging tools in Chapter 14.**

Each VBA code module includes two or more sections:

- A declarations section at the top of the module
- A section for each procedure

### *The declarations section*

You can use the declarations section at the top of a VBA module to declare (define) variables used in the module's procedures. A variable is a way to temporarily store values as your code runs. Examples of variables include

- `intCounter` (an integer)
- `curMySalary` (a currency)
- `dtmDate` (a date/time)

The three-character prefixes (`int`, `cur`, `dtm`) in these variable names constitute a simple naming convention and are entirely optional. Naming conventions are routinely used by most Access developers to help document a variable's data type (integer, currency, and datetime, respectively, in this case). Anything you can do to document your application helps reduce maintenance costs and make your code easier to modify later on.

The name you give a variable doesn't determine the type of data contained within the variable.

## Caution

**You might have noticed the `Option Explicit` line at the top of the VBA modules in this chapter's figures and example database. `Option Explicit` is a directive that instructs the VBA compiler to require every variable to be "explicitly" declared within the module. This means that every variable must be declared as a specific data type (integer, string, and so on).**

**Explicit variable declaration is always a good idea because it prevents stray variables from creeping into an application and causing bugs. Without the `Option Explicit` directive at the top of the module, every time you type in an identifier that the VBA compiler doesn't recognize, it automatically creates a new variable by that name. This means that, when using "implicit" variable declarations, if you've been using a variable named `strLastName` and type it in incorrectly (for example, `strLstName`), the VBA compiler creates a new variable named `strLstName` and begins using it. Bugs caused by simple misspellings can be very difficult to detect, because the application doesn't raise any errors, and the only way to detect the cause of the bug is to go through the code one line at a time until you find the misspelling.**

You aren't required to declare every variable used in a module within the declarations section, because variables are also declared within procedures. But be aware that all the variables defined in the declarations section are shared by all the procedures within the module.

## Cross Reference

**Variable scope (the term that describes where a variable can be used in an application) is described in Chapter 11.**

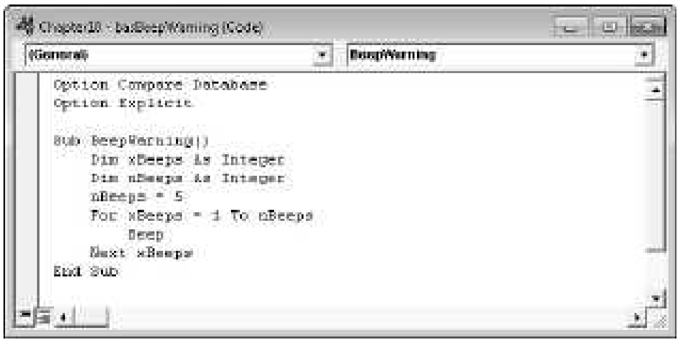
### Creating a new procedure

After you complete any declarations for the module, you're ready to create a procedure. Follow these steps to create a procedure called `BeepWarning`:

1. Open the `Module1` module you previously created, as shown in Figure 10.9.
2. Go to any empty line in the code window.
3. Type the code in *exactly* as shown in Figure 10.10.

**FIGURE 10.10**

Entering a new procedure in the code window



```
Option Compare Database
Option Explicit

Sub BeepWarning()
    Dim xBeeps As Integer
    Dim nBeeps As Integer
    nBeeps = 5
    For xBeeps = 1 To nBeeps
        Beep
    Next xBeeps
End Sub
```

4. Save the module (the save button is in the VBA editor's toolbar), naming it `basBeepWarning`.

Notice that, as you entered the first line of the procedure, Access automatically added the `End Sub` statement to the procedure.

In this example, you're running the program five times. The completed function should look like the one shown in Figure 10.10.

When `BeepWarning` actually runs, the five beeps will almost certainly blend together as a single beep from your computer's speaker.

If you enter the name of a function you previously created in this module (or in another module within the database), Access informs you that the function already exists. Access doesn't enable you to create another procedure with the same name.

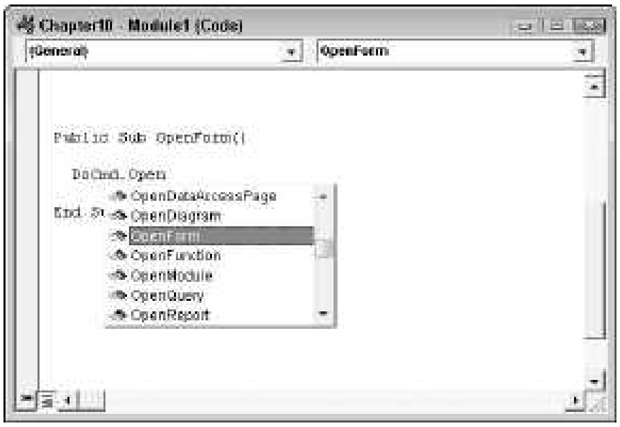
### Using IntelliSense

Suppose that you know you want to use a specific command, but you can't remember the exact syntax. Access features two types of IntelliSense to help you create each line of code:

- **Auto List Members:** Auto List Members is a drop-down list that is automatically displayed when you type the beginning of a keyword that has associated objects, properties, or methods. For example, if you enter `DoCmd.Open`, a list of the possible options displays, as shown in Figure 10.11. Scroll through the list box and press Enter to select the option you want.

FIGURE 10.11

Access Auto List Members help in a module

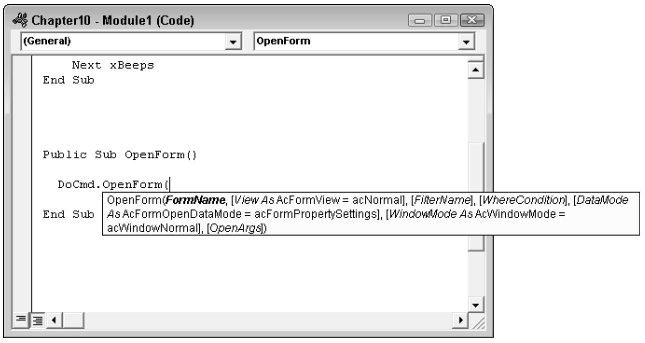


```
Public Sub OpenForm()

  DoCmd.Open

End Su
```

In this example, the `OpenForm` method is selected (actions associated with an object are called *methods*). After choosing an item in the list, more Auto List Members help is displayed. Or, if parameters are associated with the keyword, the other type of module help, Auto Quick Info (see the next bullet), is displayed, as shown in Figure 10.12.

FIGURE 10.12

Access Auto Quick Info help in a module



```
    Next xBeeps
End Sub



Public Sub OpenForm()

  DoCmd.OpenForm(

End Sub
```

OpenForm(***FormName***, [*View As* AcFormView = acNormal], [*FilterName*], [*WhereCondition*], [*DataMode As* AcFormOpenDataMode = acFormPropertySettings], [*WindowMode As* AcWindowMode = acWindowNormal], [*OpenArgs*])

- **Auto Quick Info:** Auto Quick Info guides you through all the options (called *parameters*) for the specific item. The bold word (`FormName`) is the next parameter available for the `DoCmd` object. Figure 10.12 shows that there are many parameters available for the `OpenForm` command. The parameters are separated by commas. As each parameter is entered the next parameter is highlighted in bold. The position of parameters is significant; they can't be rearranged without causing problems. Press the Esc key to hide Auto List Members help.

  Not every parameter is required for every VBA command. Parameters surrounded by square brackets (such as `View` in Figure 10.12) are optional. Access provides reasonable defaults for all optional arguments that are omitted from the statement using the command.

### *Compiling procedures*

After code has been written, you should compile it to complete the development process.

The compilation step converts the English-like VBA syntax to a binary format that is easily executed at runtime. Also, during compilation, all your code is checked for incorrect syntax and other errors that will cause problems when the user works with the application.

If you don't compile your Access applications during the development cycle, Access compiles the code whenever a user opens the application and begins using it. In this case, errors in your code might prevent the user from using the application, causing a great deal of inconvenience to everyone involved.

Compile your applications by choosing Debug➪Compile from the code window. An error window appears if the compilation is not successful.

## Note

**Access compiles all procedures in the module, and all modules in the Access database, not just the current procedure and module.**

### *Saving a module*

When you finish creating a procedure, you save it by saving the module. Save the module by choosing File➪Save, or simply close the code editor to save the module automatically. Access prompts you for a name to apply to the module if no name has yet been assigned. If the code you're working on is part of a form or report's module, the form or report is saved along with the module.

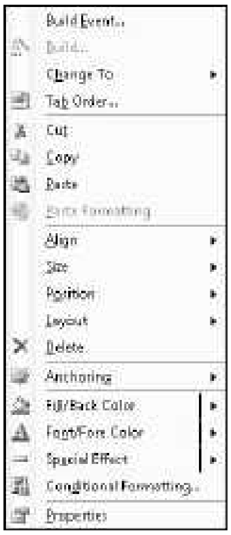
### *Creating procedures in the form or report design window*

All forms, reports, and their controls may have *event procedures* associated with their events. While you're in a form or report's Design view, you can add an event procedure in any of three ways:

- Choose `Build Event` from the shortcut menu (see Figure 10.13).
- Choose `Code Builder` in the Choose Builder dialog box when you click the builder button to the right of an event in the Property dialog box.

**FIGURE 10.13**

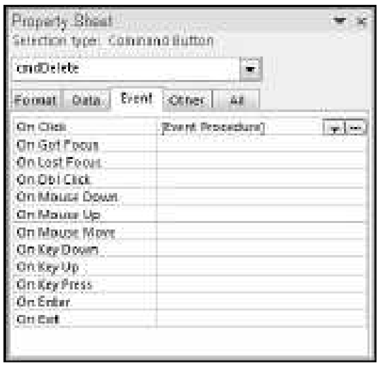
The shortcut menu for a control in the form design window



- Type **[Event Procedure]** into the event property, or select it from the top of the event drop-down list (see Figure 10.14).

**FIGURE 10.14**

The Property Sheet in the form design window


Property Sheet

Selection type: Command Button

cmdDelete

Format | Data | Event | Other | All

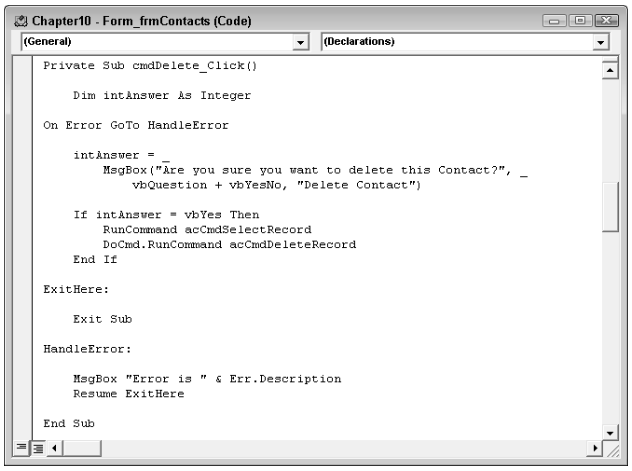
| | |
|---|---|
| On Click | [Event Procedure] |
| On Got Focus | |
| On Lost Focus | |
| On Dbl Click | |
| On Mouse Down | |
| On Mouse Up | |
| On Mouse Move | |
| On Key Down | |
| On Key Up | |
| On Key Press | |
| On Enter | |
| On Exit | |

Whether you choose `Build Event` from the shortcut menu or click the builder button in the Property dialog box (or press F7, for that matter), the Choose Builder dialog box appears.

Choosing the `Code Builder` item opens the VBA code editor, as shown in Figure 10.15. Clicking the View Microsoft Access button in the code window's toolbar toggles between the form designer and the VBA code window.

**FIGURE 10.15**

A form module open in Design view



```
Private Sub cmdDelete_Click()

    Dim intAnswer As Integer

On Error GoTo HandleError

    intAnswer = _
        MsgBox("Are you sure you want to delete this Contact?", _
            vbQuestion + vbYesNo, "Delete Contact")

    If intAnswer = vbYes Then
        RunCommand acCmdSelectRecord
        DoCmd.RunCommand acCmdDeleteRecord
    End If

ExitHere:

    Exit Sub

HandleError:

    MsgBox "Error is " & Err.Description
    Resume ExitHere

End Sub
```

### *Editing an existing procedure*

There a number of ways to access existing code behind a form or report. These are by far the most common:

- Click the `View Code` button in the Tools group in the Access ribbon (with the form or report open in Design view, of course).
- Select an event procedure from a control's event property (refer to Figure 10.7).

Opening a standard module is even easier. Simply select `Modules` in the Navigation Pane; then right-click on a module and select Design View from the shortcut menu (see Figure 10.16).

### *Checking your results in the Immediate window*

When you write code for a procedure, you might want to try the procedure while you're in the module, or you might need to check the results of an expression. The Immediate window (shown in Figure 10.17) enables you to try your procedures without leaving the module. You can run the module and check variables. You could, for example, type **?** and the name of the variable.

Press Ctrl+G to view the Immediate window, or choose View➪Immediate Window in the VBA code editor.

**FIGURE 10.16**

Selecting a module to edit

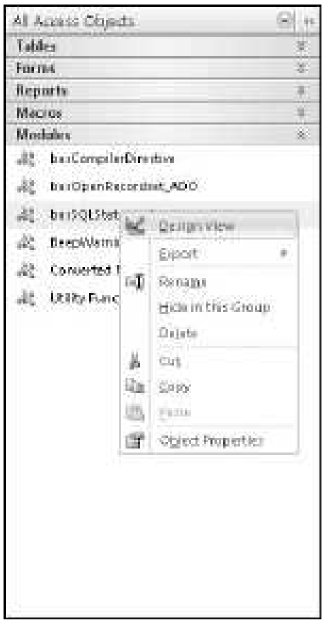
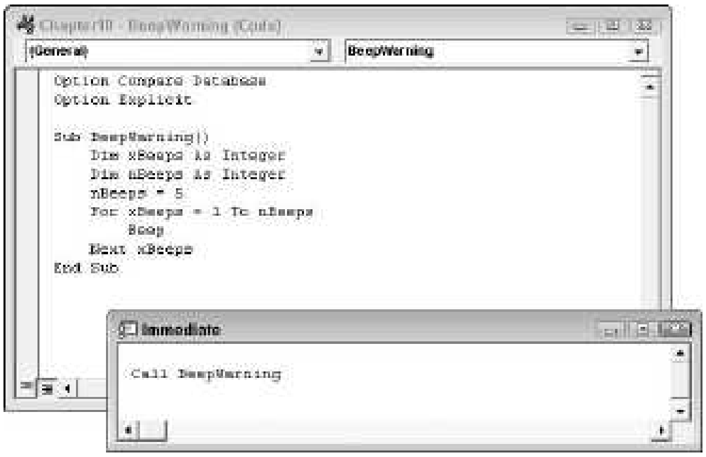


**FIGURE 10.17**

The Immediate window



```
Option Compare Database
Option Explicit

Sub BeepWarning()
    Dim xBeeps As Integer
    Dim nBeeps As Integer
    nBeeps = 5
    For xBeeps = 1 To nBeeps
        Beep
    Next xBeeps
End Sub
```

```
Call BeepWarning
```

Running the `BeepWarning` procedure is easy. Simply type **BeepWarning** into the Immediate window and press Enter. You might hear five beeps or only a continuous beep because the interval between beeps is short.

Figure 10.10, earlier in this chapter, also shows the VBA code for this subprocedure.

**Cross-Reference**

**You'll see many more examples of using the Immediate window in Chapter 14.**

## Understanding VBA Branching Constructs

The real power of any programming language is its capability to make a decision based on a condition that might be different each time the user works with the application. VBA provides two ways for a procedure to execute code conditionally: branching and looping.

### Branching

Often, a program performs different tasks based on some value. If the condition is `True`, the code performs one action. If the condition is `False`, the code performs a different action. An application's capability to look at a value and, based on that value, decide which code to run is known as *branching* (or conditional processing).

The procedure is similar to walking down a road and coming to a fork in the road; you can go to the left or to the right. If a sign at the fork points left for home and right for work, you can decide which way to go. If you need to go to work, you go to the right; if you need to go home, you go to the left. In the same way, a program looks at the value of some variable and decides which set of code should be processed.

VBA offers two sets of conditional processing statements:

- `If...Then...Else...End If`
- `Select Case...End Select`

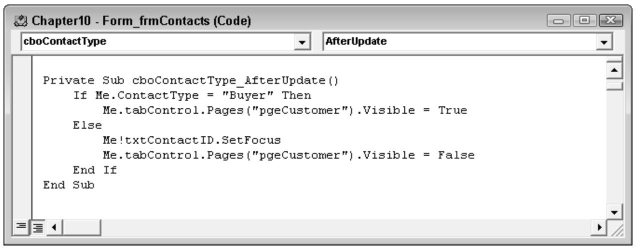
#### The If...Then...Else...End If construct

The `If...Then...End If` and `If...Then...Else...End If` construct checks a condition and, based on the evaluation, perform an action. The condition must evaluate to a Boolean value (`True` or `False`). If the condition is `True`, the program moves to the line following the `If` statement. If the condition is `False`, the program skips to the statement following the `Else` statement, if present, or the `End If` statement if there is no `Else` clause.

In Figure 10.18, the code examines the value of the `ContactType`, and if the value is `"Buyer"`, the Customer page is made visible; otherwise, the Customer page is made invisible.

FIGURE 10.18

The VBA code decides whether to display the Customer page of a tab control.



```
Private Sub cboContactType_AfterUpdate()
    If Me.ContactType = "Buyer" Then
        Me.tabControl.Pages("pgeCustomer").Visible = True
    Else
        Me!txtContactID.SetFocus
        Me.tabControl.Pages("pgeCustomer").Visible = False
    End If
End Sub
```

The `Else` statement is optional. Use `Else` to perform an alternative set of actions when the `If` condition is `False`:

```
If Condition Then
    [Action to perform when Condition is True]
Else
    [Action to perform when Condition is False]
End If
```

The `Then` and `Else` clauses can contain virtually any valid VBA statements, including another `If... Then... Else...End If`:

```
If Condition1 Then
    [Action to perform when Condition1 is True]
Else
    If Condition2 Then
        [Action to perform when Condition2 is True]
    Else
        [Action to perform when Condition2 is False]
    End If
End If
```

Needless to say, nested `If... Then... Else...End If` constructs can become quite complicated and confusing. The `ElseIf` clause sometimes helps reduce this confusion:

```
If Condition1 Then
    [Action to perform when Condition1 is True]
ElseIf Condition2 Then
    [Action to perform when Condition2 is True]
Else
    [Action to perform when Condition2 is False]
End If
```

In this example, notice that there is only one `End If` statement at the bottom of the construct.

When you have many conditions to test, the `If...Then...ElseIf...Else` conditions can get rather unwieldy. A better approach is to use the `Select Case...End Select` construct.

### The Select Case…End Select statement

VBA offers the `Select Case` statement to check for multiple conditions. Following is the general syntax of the `Select Case` statement:

```
Select Case Expression
    Case Value1
        [Action to take when Expression = Value1]
    Case Value2
        [Action to take when Expression = Value2]
    Case ...
    Case Else
       [Default action when no value matches Expression]
End Select
```

Notice that the syntax is similar to that of the `If...Then` statement. Instead of a Boolean condition, the `Select Case` statement uses an expression at the very top. Then, each `Case` clause tests its value against the expression's value. When a `Case` value matches the expression, the program executes the block of code until it reaches another `Case` statement or the `End Select` statement. VBA executes the code for only one matching `Case` statement.

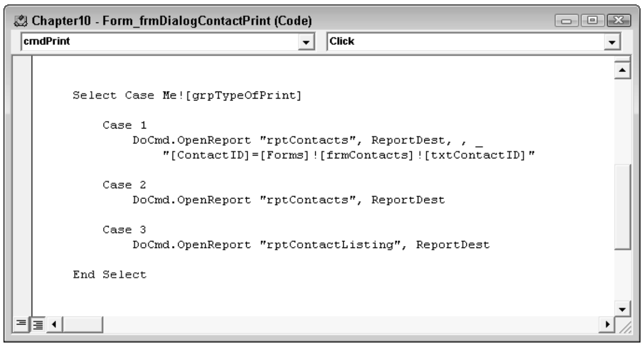
### Note

**If more than one `Case` statement matches the value of the test expression, only the code for the first match executes. If other matching `Case` statements appear after the first match, VBA ignores them.**

Figure 10.19 shows `Select...Case` used by `frmDialogContactPrint` to decide which of several reports to open.

**FIGURE 10.19**

Using the `Select Case` statement



```
Select Case Me![grpTypeOfPrint]

    Case 1
        DoCmd.OpenReport "rptContacts", ReportDest, , _
            "[ContactID]=[Forms]![frmContacts]![txtContactID]"

    Case 2
        DoCmd.OpenReport "rptContacts", ReportDest

    Case 3
        DoCmd.OpenReport "rptContactListing", ReportDest

End Select
```

Using the `Case Else` statement is optional, but it's always a good idea. The `Case Else` clause is always the last `Case` statement of `Select Case` and is executed when none of the `Case` values matches the expression at the top of the `Select Case` statement.

In some procedures, you might want to execute a group of statements more than one time. VBA provides some constructs for repeating a group of statements.

## Looping

Another very powerful process that VBA offers is repetitive looping — the capability to execute a single statement or a group of statements over and over. The statement or group of statements is repeated until some condition is met.

VBA offers two types of looping constructs:

- `Do...Loop`
- `For...Next`

Loops are commonly used to process records within a recordset, change the appearance of controls on forms, and a number of other tasks that require repeating the same VBA statements multiple times.

### The Do...Loop statement

`Do...Loop` is used to repeat a group of statements *while* a condition is `True` or *until* a condition is `True`. This statement is one of the most commonly used VBA looping constructs:

```
Do [While | Until Condition]
    [VBA statements]
    [Exit Do]
    [VBA statements]
Loop
```

Alternatively, the `While` (or `Until`) may appear at the bottom of the construct:

```
Do
    [VBA statements]
    [Exit Do]
    [VBA statements]
Loop [While | Until Condition]
```

Notice that `Do...Loop` has several options. The `While` clause causes the VBA statements within the `Do...Loop` to execute as long as the condition is `True`. Execution drops out of the `Do...Loop` as soon as the condition evaluates to `False`.

The `Until` clause works in just the opposite way. The code within the `Do...Loop` executes only as long as the condition is `False`.

Placing the `While` or `Until` clause at the top of the `Do...Loop` means that the loop never executes if the condition is not met. Placing the `While` or `Until` at the bottom of the loop means that the loop executes at least once because the condition is not evaluated until after the statements within the loop has executed the first time.

`Exit Do` immediately terminates the `Do...Loop`. Use `Exit Do` as part of a test within the loop:

```
Do While Condition1
    [VBA statements]
    If Condition2 Then
        Exit Do
    End If
    [VBA statements]
Loop
```

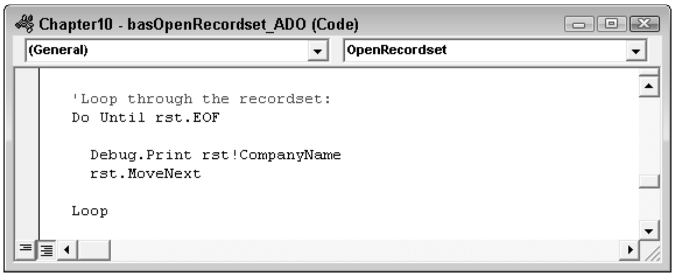
`Exit Do` is often used to prevent endless loops. An endless loop occurs when the condition's state (`True` or `False`) never changes within the loop.

In case you're wondering, *Condition1* and *Condition2* in this example may be the same. There is no requirement that the second condition be different from the condition used at the top of the `Do...Loop`.

Figure 10.20 illustrates how a `Do` loop may be used. In this particular example, a recordset has been opened and each record is processed within the `Do` loop. In this example, the company's name is printed in the Immediate window, but the data is not modified or used in any way.

**FIGURE 10.20**

Using the `Do...Loop` statement



```
'Loop through the recordset:
Do Until rst.EOF

  Debug.Print rst!CompanyName
  rst.MoveNext

Loop
```

The `While` and `Until` clauses provide powerful flexibility for processing a `Do...Loop` in your code.

## The For...Next statement

Use `For...Next` to repeat a statement block a set number of times. The general format of `For...Next` is

```
For CounterVariable = Start To End
    [Statement block]
Next CounterVariable
```

We already saw an example of the `For...Next` loop. Earlier in this chapter, you saw a procedure named `BeepWarning` that looks like this:

```
Sub BeepWarning()
    Dim xBeeps As Integer
    Dim nBeeps As Integer
    nBeeps = 5
    For xBeeps = 1 To nBeeps
        Beep
    Next xBeeps
End Sub
```

In this procedure, `xBeeps` is the counter variable, `1` is the start, and `nBeeps` is the end. In this example, `xBeeps` starts at `1` and is incremented at the bottom of the `For...Next` loop at the `Next xBeeps` statement.

An alternate form of `For...Next` is

```
For CounterVariable = Start To End Step StepValue
    [Statement block]
Next CounterVariable
```

The only difference here is the `StepValue` added to the first statement. The `Step` keyword followed by an increment causes the counter variable to be incremented by the step value each time the loop executes. For example, if *Start* is `10` and *End* is `100` and *StepValue* is `10`, the counter variable starts at 10 and increments by 10 each time the loop executes.

Most of the time, a `For...Next` loop counts upward, starting at an initial value and incrementing the counter variable by the amount specified by the step value. In some cases, however, you might need a loop that starts at a high start value and steps downward to an end value. In this case, use a negative number as the step value.

The following section explains the special syntax to use when working with objects instead of simple variables.

## Working with Objects and Collections

Very often, you have to work with objects such as the controls on a form or a recordset object containing data extracted from the database. VBA provides several constructs specifically designed to work with objects and collections of objects.

### An object primer

Although Microsoft Access is not object oriented, it's often referred to as *object based.* Many of the things you work with in Access are objects and not just simple numbers and character strings. Generally speaking, an *object* is a complex entity that performs some kind of job within an Access application. Access uses *collections* to aggregate similar objects as a single group.

For example, when you build an Access form, you're actually creating a `Form` object. As you add controls to the form, you're adding them to the form's `Controls` collection. Even though you might add different types of controls (such as buttons and text boxes) to the form, the form's `Controls` collection contains all the controls you've added to the form.

You'll see many, many examples of working with individual objects and collections of objects in this book. Understanding how objects differ from simple variables is an important step to becoming a proficient Access developer.

Each type of Access object includes its own properties and methods, and shares many other properties (such as `Name`) and methods with many other Access objects.

Collections, however, have just a few properties and methods. These are the most important properties associated with Access collections:

- `Name`: The name of the collection. Most collection names are capitalized and are the plural form of the type of object contained within the collection. For example, a form's controls are contained within the form's `Controls` collection.
- `Count`: The number of items contained with the collection. A collection with a `Count` of `0` is empty. Collections can contain virtually any number of items, but performance degrades when the `Count` becomes very large (in excess of 50,000 objects).
- `Item`: Once you have objects stored in a collection, you need a way to reference individual objects in the collection. The `Item` property points to a single item within a collection.

The following example demonstrates setting a property on just one item in a collection:

```
MyCollection.Item(9).SomeProperty = Value
```

or:

```
MyCollection.Item("ItemName").SomeProperty = Value
```

where *MyCollection* is the name assigned to the collection, *SomeProperty* is the name of a property associated with the item, and *Value* is the value assigned to the property.

This small example demonstrates a couple of important concepts regarding collections:

- **There are different ways to reference the items stored in a collection.** In most cases, each item stored in a collection (such as a form's `Controls` collection) has a name and can be referenced using its name:

  ```
  MyForm.Controls("txtLastName").FontBold = True
  ```

  As a consequence, each object's name within a collection must be unique. You can't, for example, have two controls with the same name on an Access form.

  The alternate way to reference an object in a collection is with a number that indicates the item's ordinal position within the collection. The first item added to a collection is item 0 (zero), the second is item 1, and so on.

- **A collection might contain many thousands of objects.** Although performance suffers when a collection contains several tens of thousands of objects, a collection is a handy way to store an arbitrary number of items as an application runs. You'll see several examples of using collections as storage devices in this book.

## The With statement

The `With` statement enables you to loop through all the members of an object collection, setting or changing the properties of each member. Any number of statements can appear between the `With` and `End With` statements. `With` statements can be nested.

As an example, consider the code using the following `For...Next` looping construct. This code loops through all members of a form's `Controls` collection, examining each control. If the control is a command button, the button's font is set to 10 point, Bold, Times New Roman:

```
Private Sub cmdOld_Click()
  Dim i As Integer
  Dim c As Control
  For i = 0 To Me.Controls.Count - 1
    Set c = Me.Controls(i) 'Grab a control
    If TypeOf c Is CommandButton Then
      'Set a few properties of the control:
      c.FontName = "Times New Roman"
      c.FontBold = True
      c.FontSize = 12
    End If
  Next
End Sub
```

Don't be confused by the different expressions you see in this example. The heart of this procedure is the `For...Next` loop. The loop begins at zero (the start value) and executes until the `i` variable reaches the number of controls on the form minus one. (The controls on an Access form are numbered beginning with zero. The `Count` property tells you how many controls are on the form.) Within the loop, a variable named `c` is pointed at the control indicated by the `i` variable. The `If TypeOf` statement evaluates the exact type of control referenced by the `c` variable.

Within the body of the `If...Then` branch, the control's properties (`FontName`, `FontBold`, and `FontSize`) are adjusted. You'll frequently see code such as this when it's necessary to manipulate all the members of a collection.

Notice that the control variable is referenced in each of the assignment statements. Referencing control properties one at a time is a fairly slow process. If the form contains many controls, this code executes relatively slowly.

An improvement on this code uses the `With` statement to isolate one member of the `Controls` collection and apply a number of statements to that control. The following code uses the `With` statement to apply a number of font settings to a single control.

```
Private Sub cmdWith_Click()
  Dim i As Integer
  Dim c As Control
  For i = 0 To Me.Controls.Count - 1
    Set c = Me.Controls(i)  'Grab a control
    If TypeOf c Is CommandButton Then
      With c
        'Set a few properties of the control:
        .FontName = "Arial"
        .FontBold = True
        .FontSize = 8
      End With
    End If
  Next
End Sub
```

The code in this example (`cmdWith_Click`) executes somewhat faster than the previous example (`cmdOld_Click`). Once Access has a handle on the control (`With c`), it's able to apply all the statements in the body of the `With` without having to fetch the control from the controls on the form as in `cmdOld_Click`.

In practical terms, however, it's highly unlikely that you'll notice any difference in execution times when using the `With` construct as shown in this example. However, when working with massive sets of data, the `With` statement might contribute to overall performance. In any case, the `With` statement reduces the wordiness of the subroutine, and makes the code much easier to read and understand.

Think of the `With` statement as if you're handing Access a particular item and saying "Here, apply all these properties to *this* item." The previous example said, "Go get the item named *x* and apply this property to it" over and over again. The speed difference in these commands is considerable.

## The For Each statement

The code in `cmdWith_Click` is further improved by using the `For Each` statement to traverse the `Controls` collection. `For Each` walks through each member of a collection, making it available for examination or manipulation. The following code shows how `For Each` simplifies the example.

```
Private Sub cmdForEach_Click()
  Dim c As Control
  For Each c In Me.Controls
    If TypeOf c Is CommandButton Then
      With c
        .FontName = "MS Sans Serif"
        .FontBold = False
        .FontSize = 8
      End With
    End If
  Next
End Sub
```

The improvement goes beyond using fewer lines to get the same amount of work done. Notice that you no longer need an integer variable to count through the `Controls` collection. You also don't have to call on the `Controls` collection's `Count` property to determine when to end the `For` loop. All this overhead is handled silently and automatically for you by the VBA programming language.

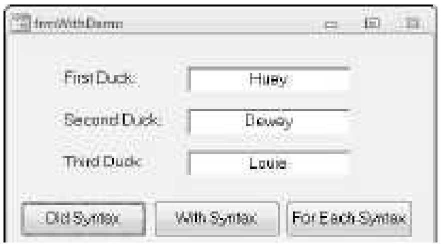
The code in this listing is easier to understand than in either of the previous procedures. The purpose of each level of nesting is obvious and clear. You don't have to keep track of the index to see what's happening, and you don't have to worry about whether to start the `For` loop at `0` or `1`. The code in the `For...Each` example is marginally faster than the `With...End With` example because no time is spent incrementing the integer value used to count through the loop and Access doesn't have to evaluate which control in the collection to work on.

**On the CD-ROM**

**The `Chapter10.accdb` example database includes `frmWithDemo` (see Figure 10.21), which contains all the code discussed in this section. Each of the three command buttons along the bottom of this form uses different code to loop through the `Controls` collections on this form, changing the font characteristics of the controls.**

FIGURE 10.21

`frmWithDemo` is included in `Chapter10.accdb`.



## Looking at Access Options for Developers

Many of the most important features in Access affect only developers. These features are hidden from end users and benefit only the person building the application. Spend some time exploring these features so that you fully understand their benefits. You'll soon settle on option settings that suit the way you work and the kind of assistance you want as you write your VBA code.

### The Editor tab of the Options dialog box

The Options dialog box contains several important settings that greatly influence how you interact with Access as you add code to your applications. These options are accessed by opening a module in the VBA code editor, and choosing `Tools`➪`Options`.

### Auto Indent

Auto Indent causes code to be indented to the current depth in all successive lines of code. For example, if you inserted four spaces (or tabs) in front of the current line of code, each line of code following the current line will be automatically indented four spaces.

### Auto Syntax Check

When the Auto Syntax Check option is selected, Access checks each line of code for syntax errors as you enter it in the code editor. Many experienced developers find this behavior intrusive and prefer to keep this option disabled, instead letting the compiler point out syntax errors. Most of the syntax errors caught by Auto Syntax Check are the most obvious spelling errors, missing commas, and so on.

### Break on all Errors

Break on All Errors causes Access to behave as if `On Error GoTo 0` is always set, regardless of any error trapping you might set up in code. When this option is selected, Access stops on every error, making it easier to debug the code.

### Require Variable Declaration

This setting automatically inserts `the Option Explicit` directive into all VBA modules in your Access application. This option is *not* selected by default in recent versions of Access.

## Tip

**When you get used to having `Option Explicit` set on every module (including global and class modules), the instances of rogue and unexplained variables (which, in reality, are almost always misspellings of declared variables) disappear. With `Option Explicit` set in every module, your code is more self-explanatory and easier to debug and maintain because the compiler catches every single misspelled variable.**

### Compile on Demand

Compile on Demand instructs Access to compile modules only when their procedures are required somewhere else in the database. When this option is unchecked, all modules are compiled anytime any procedure is called.

### Auto List Members

This option pops up a list box containing the members of an object's object hierarchy in the code window. In Figure 10.11, the list of Application objects appeared as soon as I typed as the period following `Application` in the VBA statement. You select an item from the list by continuing to type it in or scrolling the list and pressing the spacebar.

### Auto Quick Info

When Auto Quick Info has been selected Access pops up syntax help (refer to Figure 10.12) when you enter the name of a procedure (function, subroutine, or method) followed by a period, space, or opening parenthesis. The procedure can be a built-in function or subroutine or one that you've written yourself in Access VBA.

### Auto Data Tips

The Auto Data Tips option displays the value of variables when you hold the mouse cursor over a variable with the module in break mode. Auto Data Tips is an alternative to setting a watch on the variable and flipping to the Debug window when Access reaches the break point.

**Cross-Reference**

**Debugging Access VBA is described in Chapter 14.**

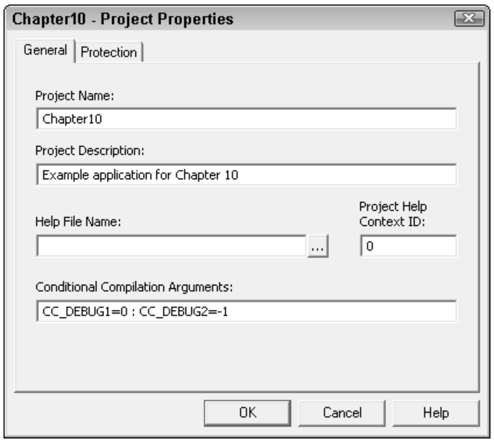
## The Project Properties dialog box

All the code components in an Access application, including all the modules, procedures, variables, and other elements are aggregated as the application's VBA project. The VBA language engine accesses modules and procedures as members of the project. Access manages the code in your application by keeping track of all the code objects that are included in the project, which is different than and separate from the code added into the application as runtime libraries and wizards.

Each Access project includes a number of important options. The Project Properties dialog box (shown in Figure 10.22) contains a number of settings that are important for developers. Open the Project Properties dialog box by opening a module in the code window, and choosing Tools➪*Project Name* Properties (where *Project Name* is the name of your database's project).

FIGURE 10.22

The Project Properties dialog box contains a number of interesting options.



Project Name: Chapter10

Project Description: Example application for Chapter 10

Project Help Context ID: 0

Conditional Compilation Arguments: CC_DEBUG1=0 : CC_DEBUG2=-1

## Project Name

Certain changes in an application's structure require Access to recompile the code in the application. For example, changing the code in a global module affects all statements in other modules using that code, so Access must recompile all the code in the application. Until the code is recompiled, Access "decompiles" the application by reverting to the plain-text version of the code stored in the `.accdb` file and ignoring the compiled code in the `.accdb`. This means that each line of the code must be interpreted at runtime, dramatically slowing the application.

Sometimes insignificant modifications, such as changing the name of the project itself, are sufficient to cause decompilation. This happens because of the hierarchical nature of Access VBA. Because all objects are "owned" by some other object, changing the name of a high-level object might change the dependencies and ownerships of all objects below it in the object hierarchy.

Access maintains a separate, independent project name for the code and executable objects in the application. Simply changing the name of the `.accdb` file is not enough to decompile the code in an Access application. By default, the project name is the same as the name of the `.accdb`, but it's not dependent on it. You can assign a unique name to the project with the Project Name text box in the General tab of the Project Properties dialog box.

## Project Description

The project description is, as its name implies, a description for the project. Because this area is so small, it isn't possible to add anything of significance that might be helpful to another developer.

## Conditional Compilation Arguments

Compiler directives instruct the Access VBA compiler to include or exclude portions of code, depending on the value of a constant established in the module's declarations section.

One of the limitations of using compiler directives is that the constant declaration is local to the module. This means that you have to use the `#Const` compiler directive to set up the constant in every module that includes the `#If` directive. This limitation can make it difficult to remove all the `#Const` compiler directives to modify the code at the conclusion of development.

For example, consider a situation in which you want to use conditional compilation to include certain debugging statements and functions during the development cycle. Just before shipping the application to its users, you want to remove the compiler directives from the code so that your users won't see the message boxes, status-bar messages, and other debugging information. If your application consists of dozens of forms and modules, you have to make sure you find every single instance of the `#Const` directive to make sure you successfully deactivated the debugging code. (This is why it's such a good idea to apply a naming convention to the identifiers you use with the `#Const` directive.)

Fortunately, Access provides a way for you to set up "global" conditional compilation arguments. The General tab of the Project Properties dialog box contains the Conditional Compilation Arguments text box, where you can enter arguments to be evaluated by the conditional compilation directives in your code.

As an example, assume you've set up the following sort of statements in all the modules in your application:

```
#If CC_DEBUG2 Then
  MsgBox "Now in ProcessRecords()"
#End If
```

Instead of adding the constant directive (`#Const CC_DEBUG2 = True`) to every module in the application, you might enter the following text into the Conditional Compilation Arguments text box:

```
CC_DEBUG2 = -1
```

This directive sets the value of `CC_DEBUG2` to `-1` (`True`) for all modules (global and form and report class modules) in the application. You need to change only this one entry to `CC_DEBUG2=0` to disable the debugging statements in all modules in the application.

**Note**

**You don't use the words `True` or `False` when setting compiler constants in the Project Properties dialog box, even though you do use these values within a VBA code module. You must use `-1` for `True` and `0` for `False` in the Project Properties dialog box.**

Separate multiple arguments with colons — for example: `CC_DEBUG1=0 : CC_DEBUG2=-1.`

## Command-line arguments

The Options dialog box you open from the File menu (click on the large round Microsoft Office Button in the upper-left corner of the main Access window, and choose File➪`Access Options`) provides a number of interesting options. Select the `Advanced` tab and scroll down to the Advanced section near the bottom of the dialog box. Notice the `Command-Line Arguments` text box at the very bottom of the Advanced section.

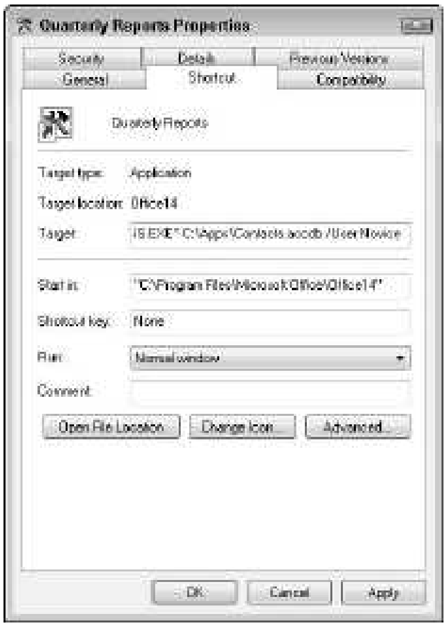
Many applications use command-line arguments to influence how the application behaves at runtime. You could, for example, add a command-line argument to an Access database application that indicates whether the user was an experienced or novice user. The application might display help and other assistance that is appropriate for the user's experience level. (Use the `Command` function to return the arguments portion of the command-line used to start Access or the Access runtime environment.)

Passing a Windows application command-line arguments during development has always been difficult. Windows requires command-line arguments to be passed as text in the Target text box of a program icon's Property Sheet. Figure 10.23 shows such a Property Sheet. The text `/User Novice` in the Target text box is the command-line argument passed to the Access application as it starts up.

Before the `Command-Line Arguments` option was available, there was no easy way to test the effect of command-line arguments in your application. Use this option to test and debug the command-line argument code you build into your applications.

**FIGURE 10.23**

Adding a command-line argument to a shortcut pointing to a Windows application



Quarterly Reports

Target type: Application

Target location: Office14

Target: IS.EXE" C:\Apps\Contacts.accdb /User Novice

Start in: "C:\Program Files\Microsoft Office\Office14"

Shortcut key: None

Run: Normal window

Comment:

## Caution

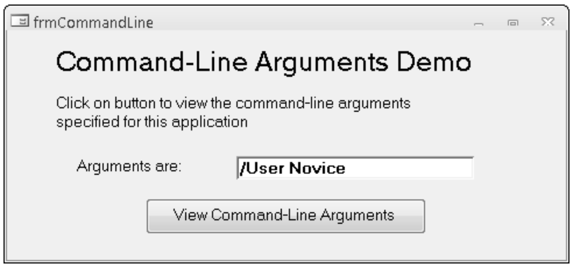
**Don't forget to remove the text from this option before distributing your application to end users. The text you enter in this option setting is persistent and will remain there until it's removed or changed.**

## On the CD-ROM

**Use the** `Command` **function to return the arguments portion of the command line used to start Access or the Access runtime environment. The** `Chapter10.accdb` **example database includes** `frmCommandLine` **(see Figure 10.24), a demonstration of the** `Command` **function. Use the Options dialog box to set some command-line arguments for** `Chapter10.accdb`**; then click on the button on** `frmCommandLine` **to see how the** `Command` **function retrieves the arguments.**

**FIGURE 10.24**

`Chapter10.accdb` includes `frmCommandLine` to demonstrate using command-line arguments in your applications.


frmCommandLine

Command-Line Arguments Demo

Click on button to view the command-line arguments specified for this application

Arguments are: /User Novice

## Summary

This chapter reviewed some of the important topics as you work with Access VBA. I showed you the fundamental concepts of creating VBA modules and procedures and touched on the important topic of event-driven programming in Microsoft Access.

You also read that Access provides a large number of options and settings that influence how you work with your modules and procedures. The good news is that you have a lot of options controlling the appearance and behavior of the Access code editor. There is *no* bad news about writing code in Microsoft Access!

The `With...End`, `With`, and `For Each` constructs make it easy and efficient to traverse the members of object collections. Named arguments give you a lot more flexibility in passing parameters to functions and subroutines.

You continue your exploration of the VBA programming language in the next several chapters. In Chapters 11 through 15 you learn virtually every fundamental skill necessary to succeed as a VBA programmer. One important aspect of VBA programming is that it's a skill with no barriers — your abilities as an Access VBA programmer are completely transferable to any of the other Microsoft Office products like Word and Excel.