


# Microsoft® Access® 2010 Programmer's Reference

Teresa Hennig, Rob Cooper, Geoffrey Griffith, Jerry Dennison

# 7

# Using VBA in Access

**WHAT'S IN THIS CHAPTER?**

- Using events
- Employing good practices with VBA procedures
- How to evaluate expressions in VBA
- Using recordsets
- Using multiple recordsets
- Coding behind forms and reports
- The VBA debugging environment
- Determining the value of variables
- Handling common VBA challenges
- Concatenating strings
- Handling VBA errors

In the early days of programming, procedural languages ruled, meaning that the overall program execution was very structured and code was generally run in a very specific order. The main body of any of these programs had to cover every possibility: Display a screen to the user, gather input, perform edit checking, display messages, update the database (or simple files in those days), and close when everything was done. The main program also had to deal with every option or side request that the user might make. This made it difficult to understand the entire program, and it was tough to make changes because everything had to be retested when a modification was made. Those lumbering beasts included FORTRAN, COBOL, RPG, Pascal, and earlier forms of Basic. Millions of lines of code were written in these languages.

Fortunately, those days are over for VBA programmers. VBA is an *event-driven* language. In every Access form and report, a variety of events are waiting for you to use. They are available when the form opens and closes, when records are updated, even when individual fields on the screen are changed. They're all there at your fingertips. Each event can contain a procedure, which is where we get back to the procedural roots of standard programming. Although each procedure runs from top to bottom, just like in the old days, it only runs when the event *fires*. Until then, it sleeps quietly, not complicating your logic or slowing down your program.

Event-driven programming makes it much easier to handle complex programming tasks. Because your code will only run when an event occurs, each procedure is simpler and easier to debug.

In this chapter, you'll explore the nature of VBA events and see how the most common events are used, and you'll look at how two different sections of your VBA code can run at the same time. The chapter provides some guidelines about when and how to use Public and Private procedures, and data types, and also outlines structural guidelines for procedures, shows some common string and date handling techniques, and explains how to prevent rounding errors in your calculations. Class procedures, a powerful and useful tool, are covered in great detail in Chapter 8.

## WHEN EVENTS FIRE

Events are at the heart of event-driven programming — which is no surprise. What can be surprising to novice programmers is the sheer number of events available to use. They all beg to have some code behind them. In reality, however, very few events are used on a consistent basis. Most of them have absolutely no code behind them, and never will in normal usage. The trick is to know which ones are important and commonly used, and which ones are obscure and rarely ever used. They all appear equally important in Access Help.

### Common Form Events

The following table provides a list of common events and how you might want to use them. By knowing how to use this basic set of events, you're most of the way there to understanding event-driven programming in Access VBA.

| FORM EVENT | DESCRIPTION |
|---|---|
| `On Open` | Fires before the `On Load` event (so you can't reference any bound controls on your form yet because they haven't been instantiated) and before the recordset is evaluated for the form. This means you can use this event to change the recordset (by changing the `WHERE` or `ORDER BY` clause) before the form continues to load. Cancel this event by setting its intrinsic parameter `Cancel = True`, so the form will close without continuing to the `On Load` event. |
| `On Load` | Fires after the recordset for the form has been evaluated but before the form is displayed to the user. This offers you an opportunity to make calculations, set defaults, and change visual attributes based on the data from the recordset. |

| FORM EVENT | DESCRIPTION |
|---|---|
| `Before Update` | To perform some data edits before the user's changes are updated in the database, use this event. All the field values are available to you, so you can do multiple field edits (such as `HireDate` must be greater than `BirthDate`). If something doesn't pass your validity checks, you can display a message box and cancel this event by setting the intrinsic parameter `Cancel = True`. This event also fires before a new record is inserted, so you can place edits for both new and changed records here. |
| `On Double Click` | A non-intuitive, special-purpose event. If you build a continuous form to display records in a read-only format, your users may expect to drill down to the detail of the record by double-clicking anywhere on the row. But what if they double-click the record selector (the gray arrow at the left side of each row)? The event that fires is the form's `On Double Click` event. By using this event, you can run the code that opens your detail form. This gives your user a consistent experience and the confidence that your applications work no matter what. |
| `On Unload` | This event can be used to check data validity before your form closes. It can be canceled, which redisplays your form without closing it. It also has another useful behavior. If it is canceled during an unload that occurred because the user is closing Access (using the X button in the window heading), canceling the `Unload` event also cancels all other form closures and the closure of Access itself. This allows you to prompt the user with an "Are you sure?" message box when the user tries to close Access. |
| `On Current` | This is one of the most overused events by novice programmers, but it does have some good uses. It fires every time your form's "current" record changes. The current record is the one that the record selector (the gray arrow on the left side of each record) points to. It also fires when your form initially loads and positions to the first record in your recordset. One good place to use `On Current` is on a continuous form where one of the buttons below is valid for some records but not for others. In the `On Current` event, you can test the current record and set the Enabled property of the button to `True` or `False` as appropriate. Because this event fires so often, it can be hard to control and may cause performance issues. Use it only when you need to. |
| `On Delete` | Fires after each record is deleted, but before the delete is actually finalized, enabling you to display an "Are you sure?" message. Then the user has an opportunity to decide whether or not to delete this individual record. Use this in conjunction with the `Before Delete Confirm` event. |
| `Before Delete Confirm` | Fires before a group of deletes is finalized. If you cancel this event, none of the records in the group is actually deleted. This event also has a `Response` parameter; it can be used to suppress the normal Access message asking the user if he wants to delete the group of records. |
| `On Activate` | Fires after the form's `On Open` and `On Load` events, just before the form is displayed. It also fires whenever the form regains the focus, so it can be used to refresh or requery the data on the form after the user has returned from another form. |

## Common Control Events

The following table lists some commonly used events for controls on forms (such as text boxes, combo boxes, command buttons, and so on).

| CONTROL EVENT | DESCRIPTION |
|---|---|
| `On Click` | This one is obvious; it fires when the control (most likely a command button) is clicked. This is where you put the code to run when the user clicks a button. |
| `Before Update` | Useful for controls whose value or state can change, such as checkboxes, text boxes, and combo boxes. It fires just before a change to the control is committed, so you have a chance to validate the new value of the control. If this event is canceled, the control reverts to its previous value. You can ask the user a question in this event using a message box, such as "Are you sure you want to change the Invoice Number?" You can then continue normally or set `Cancel = True` based on the response. |
| `After Update` | Fires after a change to the control is made. This is a good time to control the next field to receive the focus, manipulate other fields in response to this one, or perform other actions (these techniques are used in Chapter 14). |
| `On Double Click` | Fires when a control is double-clicked. Useful when you want to provide a method of drilling down to a detail form from a read-only index form. Make sure you add the code to open the detail form to every double-click event of every field in the detail section. If your record selector arrow is visible, include your drill-down code to the form's `On Double Click` event (see previous section). |

## Common Report Events

The following table lists some commonly used report events. These events can run code to customize and add flexibility for your users when displaying reports.

| REPORT EVENT | DESCRIPTION |
|---|---|
| `On Open` | Fires before the recordset is evaluated for the report. As with forms, you can use this event to change the recordset (by changing the `WHERE` or `ORDER BY` clause) before the report continues to load. This can be especially helpful when you use a form to prompt the user for selection criteria before the report continues to load (described in detail in Chapter 15). This event can be canceled by setting the `Cancel` parameter to `True`, which will prevent the report from continuing to open. |

| REPORT EVENT | DESCRIPTION |
|---|---|
| `On Activate` | Fires after the `On Open` event and just as the report window is displayed to the user. The main thing this event is used for is to maximize the Access windows using `DoCmd.Maximize`. This allows the user to see more of the report. However, you'll probably want to restore the Access windows to their previous sizes when the report closes, which brings us to the `On Close` event. |
| `On Close` | Fires when the report closes. A common line of code to include here is `DoCmd.Restore` to restore the sizes of your form windows that were maximized in the `On Activate` event. |
| `On No Data` | Fires after the `On Open` event when the report evaluates the recordset and discovers that there are no records. This can easily happen if you allow users to specify the criteria for the report and they choose a combination of values that doesn't exist in the database. You can display a friendly message box to the user, and then set the intrinsic `Cancel` parameter to `True`, which closes the report. |
| `On Load` | Introduced in Access 2007. The `On Load` event fires after the `On Open` event. In this event, the recordset for the report has already been evaluated and data from the first record is available. |

## Asynchronous Execution

Sometimes, Access runs two areas of your VBA code simultaneously, even though you've placed the code into different events or even in different forms and reports. This ability for Access to start running one procedure of code before another one is finished is called *asynchronous execution*. Most of the time asynchronous execution happens without you (or your user) really noticing, but it can sometimes cause problems, so you should know when it happens and how to work with it.

### OpenForm

The most common asynchronous execution you'll encounter is when you open a form using the `OpenForm` method. Most of the time you won't notice it, but here's what really happens: When the `OpenForm` statement runs, the form you ask for starts to open, along with all of its `On Open`, `On Load`, and `On Current` events. However, any code after the `OpenForm` command also continues to run at the same time. Usually, not much happens at this point, so there's no harm done.

However, there are times when you would like the execution of the code in the calling form to stop until the user is done with the newly opened form. This is often the case when you are prompting the user for selection criteria during the `Open` event of a report (see Chapter 14), or when you open a form to add a new record from an index form.

In this latter case, you normally want to requery the index form to show the newly added record, but you must wait for the user to finish adding it. If you perform a requery right after the `OpenForm`, your code will continue merrily along and requery your first form, only within milliseconds after your second form has started to open. No matter how fast your user is, that's

not enough time for them to add the new record. So your requery runs before the new record is added, and the new record will not appear on your index form.

There is a simple solution to the normal asynchronous execution of the `OpenForm` command. It's called Dialog mode.

## Dialog Mode to the Rescue

To prevent asynchronous execution when a form opens, use Dialog mode. Instead of:

```
DoCmd.OpenForm FormName:="frmMyForm"
```

specify this:


```
DoCmd.OpenForm FormName:="frmMyForm", WindowMode:=acDialog
```

*code snippet Prevent Asynchronous Execution When A Form Opens in ch07_CodeSnippets.txt*

*Note the use of named parameters in these examples — `FormName:="frmMyForm"`, for instance. Functions and subroutines in VBA can receive parameters (often called arguments) using either positions or names. If the names are not specified, VBA assigns parameters based on their position: first, second, and so on. When you see extra commas indicating missing parameters, you know that positional parameters are being used. Named parameters are much clearer to read and understand, and experienced programmers often use them.*

Dialog mode accomplishes two things:

- It opens the form in Modal mode, which prevents the user from clicking on any other Access windows until they are done with this form. Modal forms are hierarchical in nature, meaning they can be opened one after the other with the most currently open form the only one accessible. As you close the current Modal form, the one opened immediately before it is now the only one accessible and so on. All Modal forms must be closed before you can navigate to any other database object.
- It stops the execution of the calling code until the newly opened form is either closed or hidden.

This second feature of Dialog mode is what is so helpful in preventing Access from trying to run two areas of your code at once.

Notice that the code stops until the form is closed or hidden. This is the basis for many clever uses of Dialog mode where values from the called form are used elsewhere. If you just hide the form (by setting its `Visible` property to `False`), the values on the form are still there and ready for you to reference, even though the code in the calling form now continues to run. This is the technique for gathering selection criteria and building SQL statements, which is described in Chapter 14.

*There is a disadvantage to using Dialog mode. While a form is open and visible in Dialog mode, any report that is opened will appear behind the form and won't be accessible. If you encounter this problem, you can use another technique to control the timing of form requeries. One technique is to open the second form normally and allow the code in the first form to complete. Then, put your requery code in the first form's* `On Activate` *event to fire when the focus returns to the first form.*

# VBA PROCEDURES

VBA code can be structured clearly and efficiently by breaking up sections of code into logical "chunks" called *procedures*. In this section, you'll see how to use the different types of VBA procedures and to employ good practices in their design.

## Function or Sub?

A common area of confusion among novice VBA programmers is whether to write a function or a sub (short for "subroutine"). Some developers create functions for every procedure they write, in the belief that they are better in some way. They aren't. Functions and subs are just two kinds of procedures, and they both have their purposes. A quick way to determine which one is more appropriate is to ask this question: Does my procedure *do* something or *return* something?

If the purpose of your procedure is to compute or retrieve a value and return it to the calling procedure, then of course you should use a function. After all, functions are designed to return a single value to the calling procedure. They do it efficiently and easily, and they can be used directly in queries and calculated controls on forms and reports. They can even be used directly in macros.

Functions tend to have names that are nouns, like `LastDayOfMonth` or `FullAddress`. For example, a control on a report might have this `Control Source` property value:

```
=LastDayOfMonth(Date())
```

The field would display the results of calling some function called `LastDayOfMonth` with the parameter value of today's date.

On the other hand, if the main purpose of your procedure is to do some action and there is no clear-cut value to return, use a sub. Many programmers think that they must return something, even if they have to make up some artificial return code or status. This practice can make your code harder for others to understand. However, if you really need a return code or status after the procedure finishes, it is perfectly okay to make it a function.

Subs tend to have names that are verbs like `LoadWorkTable` or `CloseMonth`. In practice, the code looks like this:

```
LoadWorkTable
```

Pretty easy, right? Any developer looking at this line of code can see the obvious: A sub called `LoadWorkTable` is being called, and it doesn't return a value.

It is possible to call a function as if it were a sub, without parentheses around the parameters. In that case, the function runs, but the return value is discarded. This usually is not a good coding practice, but you may encounter it in existing code.

## Public or Private?

Another decision that you have to make when you create procedures is whether to make them `Public` or `Private`. By default, Access makes procedures you create `Public`, but that's not necessarily what you want.

If you are working in a standalone module (those that appear in the Modules area of the Access Navigation Pane), the rules are a little different than if you are working in code that resides in a form or report. Form and report modules are intrinsically encapsulated as class modules so their `Public` procedures aren't as public as you might expect. Let's take a look at procedures in standalone modules first.

### Public and Private Procedures in Modules

Public functions and subs in standalone modules are just that — public property. Every area of your application can see them and use them. To do that, `Public` procedures in modules must have unique names. Otherwise, how would your code know which procedure to run? If you have two `Public` procedures with the same name, you'll get a compile error.

`Private` procedures in modules are very shy — they can't be seen or referenced by any code outside their own module. If you try to reference a `Private` procedure from a different module or another form or report, Access insists (at compile time) that no such procedure exists.

The hidden nature of `Private` procedures is their best feature. Because they are hidden, their names need to be unique only within their own module. Therefore, you can name them whatever you want — you don't have to worry about them conflicting with other procedures in your application.

This feature really comes into play when you reuse code by importing modules into other databases, maybe even ones you didn't create. If most of your module procedures are `Private`, you'll have a minimum of naming conflicts because the rest of the application can't see them. The `Public` procedures still need to have a unique name, which is why many procedures that are meant to be imported have interesting prefixes such as the author's initials or the company name.

### Public and Private Procedures in Forms and Reports

`Private` procedures in forms and reports behave just like `Private` procedures in modules. They can't be seen or referenced from outside the form or report. The event procedures that Access automatically builds behind your forms and reports are automatically set to `Private`. This makes sense because `Form_Open` and `OnClick` events are useful only inside that particular form or report. Also, these procedures need to have standard names, which could result in a big mess of duplicate names if they were `Public`.

In reality, this problem wouldn't occur. The code behind your forms and reports isn't like the code in normal modules. Access calls them class objects, but they behave like class modules, which are covered in Chapter 8. You can see this in the Visual Basic Editing window, as shown in Figure 8-1. Note the three headings: Microsoft Access Class Objects, Modules, and Class Modules.

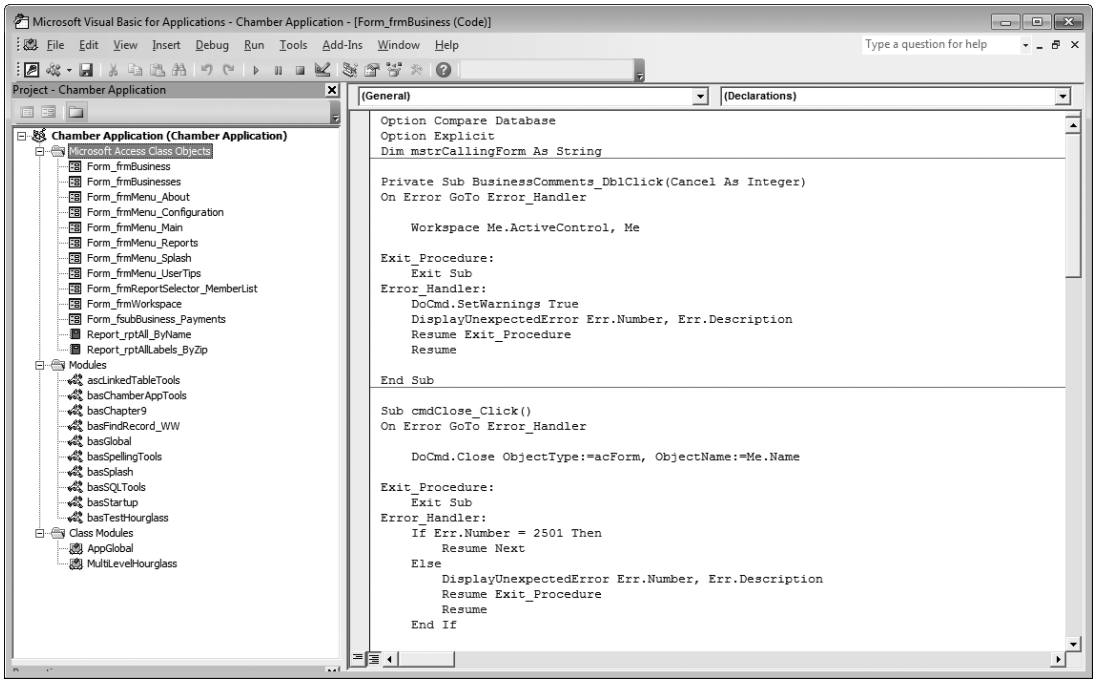


**FIGURE 7-1**

It turns out that even a `Public` procedure that you build in the code behind a form can be named the same as a procedure in another form. That's because class objects require that you specify the name of the class object (in this case, the form name) before the name of the procedure if you want to call it from outside the form. However, this is rarely needed. One possible situation might be some form initialization code that you want to run from outside the form, such as `InitializeForm`. If you want to do it, here's the syntax:


```
Form_frmMyFormName.InitializeForm
```

*code snippet Initialize Form in ch07_CodeSnippets.txt*

Notice that the prefix `Form_` and the name of the form qualify the `InitializeForm` procedure name. Because many forms could have the same procedure name, you need to tell the code which form's procedure you want to run.

## Coupling and Cohesion

The design of your procedures is important to delivering understandable, readable code. Two principles that guide the logical design of procedures (functions or subs) are *coupling* (bad) and *cohesion* (good). This topic is not specific to VBA, but it bears mentioning since we're working with procedures.

## Uncouple Procedures

Coupling is the tempting tendency to write long, complex procedures that do lots of things; in other words, *coupling* multiple tasks into one procedure. This should be avoided. As a guideline, write procedures that compute just one value or perform a single task. Some signs that you might have coupling in your procedures include:

- Procedure names that include multiple ideas, such as `ComputeValuesAndReloadWorkTables`
- Procedures with large blocks of code that have section header comments explaining what each section does
- Procedures that include "modes," with parameters that tell the procedure what to do

If your procedure couples multiple tasks together, you can run into problems like these:

- Your procedure is too complicated, making it harder to write and debug.
- The different tasks in your procedure can't be used separately; it's all-or-nothing.
- If you make a change to your procedure, the whole thing needs to be retested. You can't trust that your little change didn't affect other parts of the procedure. Remember the common programmer's lament: "But all I changed was . . ."

If you find yourself writing long procedures with these coupling problems, take a deep breath and step back from it for a minute. Try to identify chunks of code that do something simple and cohesive. As a rule, procedures should do or calculate one thing, and should do so independently using parameters that are passed to them.

You may wonder how to build procedures that must be complex. Sometimes there is no way to avoid complexity, but you can hide a lot of complexity by breaking your logic into smaller functions and subs, then calling them where appropriate. That way, each one of your procedures can be written and debugged separately. If you are working as a team, the procedures can even be written by different developers.

## Adhere to Cohesion

Cohesion means that each procedure should perform one function, and should be able to do its thing without a lot of help or knowledge from outside the procedure. It shouldn't rely on global variables or other objects to exist when the procedure is invoked. Some signs of a poor cohesion are:

- Procedures that include duplicate blocks of code
- Procedures that expect forms or reports with specific names
- Use of global variables, especially when they are expected to retain their value for a long time
- Hard coding of system environment information such as file paths
- Hard coding or special handling of certain records or values in tables

Hard coding is the practice of using values in code that would be more appropriate in a configurable lookup table or some other easy-to-change place. For example, many poorly written applications

hard code paths to files. The moment those applications are moved to another computer, they break. Another more insidious example is the use of values lists for combo boxes in forms. These seem so easy to set up, but they are just another instance of hard coding that makes your application less robust and more difficult to change over time. A better approach for a list of values that you don't think will change (or that you need to code against) is to put them in a table that doesn't have a maintenance form. This prevents your user from adding or removing the critical values your code depends on, but allows you flexibility over time. If you like, you can use a specific naming convention (as an extension of Reddick naming conventions) for these value list tables, such as `tval` instead of `tlkp` or `tbl`.

To improve cohesion, think of the old black-box principle of programming: You should need no knowledge of how a procedure produces its result, only that given valid input, it will produce the correct output. Along the same lines, the procedure should need little knowledge of the world outside to do its job. Each procedure you write should perform one task or calculation and need a minimum of special knowledge from outside its own boundaries. The best way to send information into a procedure is through parameters, not by using global variables or referring to specific forms or reports.

All this being said, cohesion is a spectrum, not a final black-or-white goal. Using VBA in Access sometimes calls for the use of global variables in controlled scenarios, or referring to an open form, or duplicating some code. It's best to be aware of coupling and cohesion principles so that you can make good coding decisions.

## Error Handling

All of your procedures should have at least a minimum level of error handling. There are easy ways to implement simple error handling that can help you debug your code and protect your users from errors (both expected and unexpected). This topic is covered in much greater detail later in this chapter.

## Using Variables

When using variables in your VBA code, there are several things to remember to ensure that your code runs smoothly. Choosing the appropriate data type for each variable is critical, and it's also important to use global variables correctly.

Naming conventions for variables are important. The Reddick naming conventions for variables are described in Appendix G. If you get into the habit of naming your variables consistently, your code will be easier to maintain over time, faster to debug, and look more professional.

### Using Appropriate Data Types and Sizes

First, make sure that your variable types will handle the size of data they are expected to store. Many overflow errors occur because an attempt was made to store an `AutoNumber` key value from a table in a variable defined as an Integer. This may work fine during testing because an integer can store numbers with values up to 32,767. Then, when a user starts adding more data, the application breaks on an overflow error.

It's a good idea to define variables with the maximum size that is possible to occur. AutoNumber fields should be stored in variables defined as Long (which is the same as the Long Integer in Access tables). Defining a variable as String allows it to store very long strings, whether they are defined as Text or Memo in a table.

If a variable can possibly contain a Null, then you must define it as a Variant, in which case it will be able to store just about anything that you throw into it — a messy approach, and one that takes Access a bit longer to process. It's usually better to decide what kind of data each variable is going to hold; then set the appropriate data type so that Access doesn't have to figure out what's in there every time it uses the variable. Sometimes, however, it's useful to allow a variable to contain a Null, especially when there might not always be data to load into the field. If you do use a Variant data type, use it because there's a specific reason that it might be passed a Null, not because you don't know what type it should be.

If you don't specify a variable's data type, it is a Variant by default. A common error is to define more than one variable on a single line of code, like this:

```
Dim strCallingForm, strReportTitle as String
```

Many novice VBA programmers think that both variables in this example are defined as Strings — they won't be. VBA requires that each variable have its data type explicitly defined. In this example, `strCallingForm` will be defined as a Variant because its data type wasn't specified.

You can define the two string variables on one line like this:


```
Dim strCallingForm as String, strReportTitle as String
```

*code snippet Define Two Strings On One Line Of Code in ch07_CodeSnippets.txt*

This style is technically correct (both variables are defined as Strings), but the second variable is easy to miss when you are looking at your code. The clearest and most consistent style for defining variables is to give each one its own line:

```
Dim strCallingForm as String
Dim strReportTitle as String
```

This may take an extra line of code, but it is much easier to read and understand.

## Using Global Variables

Global variables are variables that retain their value until they are changed or until the application stops. They can be handy, but they should be used in specific ways to avoid problems. To define a global variable, simply use `Global` instead of `Dim`, like this:


```
Global gstrCallingForm As String
```

*code snippet Dimensioning A Global Variable in ch07_CodeSnippets.txt*

Notice the naming convention: `g` for Global, `str` for String, and then the variable name.

A global can be defined in any standalone module; it doesn't matter which one. You can refer to it and set its value from anywhere in your application (that's why it's called global). However, you probably want to designate a module to store all your main reusable application code, which is where you should define your global variables. You could name this module `basGlobal` or something similar.

Global variables, however, have a problem. If your code is interrupted — after an error, for example — the global variables are cleared out. There are two ways to reduce the impact of this little problem. The best way is to use the value in global variables for a very short time, perhaps a few milliseconds. Globals can be used like parameters for objects that don't accept true parameters, such as forms. For example, the form daisy-chaining logic given in Appendix I uses a single global variable to pass the name of the calling form to the called form, but the called form immediately stores the name in a local module variable for safekeeping.

Another way to work around the problem with global variables is to create a wrapper function that first checks whether the variable has a value. If it does, it merely returns it. If it doesn't have a value (which will happen the first time the function is called, or if the value has been reset), the function then computes or retrieves the value, sets the global variable, and returns the value. This can be a good way to retrieve or compute values that take some time, such as connection string properties or other application-wide values that are retrieved from tables. You get the speed of a global variable and the reliability of computing the values when necessary.

Access 2007 introduced a new way of storing global values: `TempVars`. This is a collection of values that you can define and maintain, and it won't be reset if you stop your code during debugging. The values are retained as long as the current database is open. `TempVars` is explained in detail in Appendix I.

## EVALUATING EXPRESSIONS IN VBA

Expressions are one of the basic building blocks of any programming language. There are several ways to evaluate expressions in VBA that allow you to control the flow of your procedural logic.

### If ... Then

Almost every programming language has some way of asking `If`, and VBA is no exception. The `If..Then` structure is one of the most commonly used in VBA. Its usage is straightforward, but there are a couple of issues that warrant extra attention. First, the expression you are using needs to be formed correctly and completely. One common mistake is to use an expression like this:

```
If intOrderStatus = 1 Or 2 Then
 `some interesting code here
End If
```

The problem here is that a complete Boolean (`True` or `False`) expression needs to be on both sides of the `Or`. The literal way to interpret this expression is "if intOrderStatus = 1 or if 2 is `True`, then," which, of course, makes no sense. The constant `2` will always evaluate to `True`. In fact, the only value that evaluates to `False` is `0`. All other values are `True`. Internally, Access will store `-1` as `True`, but any value other than `0` evaluates to `True`.

The correct way to write this line of code is as follows:


```
If intOrderStatus = 1 Or intOrderStatus = 2 Then
  `some interesting code here
End If
```

*code snippet Correct 'If...Then' Syntax in ch07_CodeSnippets.txt*

It's repetitive, but you have to tell VBA exactly what you want to do.

*Instead of using multiple* `Or` *operators in SQL statements, you can use a much easier syntax: the* `In` *operator. In SQL, the equivalent to* `Where OrderStatus = 1 or OrderStatus = 2` *is merely* `Where OrderStatus In (1,2)`*. That's much easier to read and understand, and it only gets better the more values you have to compare.*

## Checking for Nulls

Another common area of confusion is checking for `Null`. The following statement is incorrect:

```
If varCustomerKey = Null Then
  `even more interesting code here
End If
```

An interesting fact about `Null`: It is, by definition, unknown and undefined. A variable containing a `Null` can't "equal" anything, including `Null`. In this example, the interesting code will never run, no matter how null the customer key field is.

To check for a Null in a field, you must use the `IsNull` function, like this:


```
If IsNull(varCustomerKey) Then
  `even more interesting code here
End If
```

*code snippet Checking For Nulls in ch07_CodeSnippets.txt*

The `IsNull` function is the only way VBA can evaluate a variable or recordset field and determine if it is Null. The `=` just can't do it. By the way, this is true in Access SQL, too — you need to use `IsNull` to test for Nulls in the `WHERE` clauses of queries and recordsets, or you can use the SQL specific syntax `WHERE [FieldName] IS NULL`.

Sometimes, you want to check to see if a field is either Null or contains an empty string (also known as a *zero-length string*). Empty strings can creep into your tables if you specify Yes to Allow Zero Length in the field definition during table design. To ensure that you are checking for both, use code such as this:

```
If IsNull(BusinessName) or BusinessName = "" Then
```

What a hassle — you have to type the name of the field twice, and the line is confusing to read. There's a much easier way:


```
If BusinessName & "" = "" Then
```

*code snippet Checking For Null Or Empty String -- Two Ways in ch07_CodeSnippets.txt*

This technique uses the concatenation behavior of the `&` operator. The `&` concatenates two strings, even if one of them is Null (see the section "String Concatenation Techniques" later in this chapter). In this case, it concatenates an empty string (`""`) onto the end of `BusinessName`. If `BusinessName` is Null, the result is an empty string. If `BusinessName` has any string value in it, it remains unchanged by tacking on an empty string. This behavior enables you to quickly check if a field has a Null or an empty string.

Notice that this example uses the `&` operator to concatenate strings. The `+` operator also concatenates strings, but there's an important difference: `+` propagates Null. That is, if either side (operand) is Null, the result is also Null. Concatenation of strings is discussed in more detail later in this chapter.

On the subject of Nulls, the `nz()` function converts a Null value to `0` (zero). It's built into VBA and can be helpful in math calculations when you don't want a Null to wipe out the whole result. For example, to calculate a price with a discount, you could use this code:

```
NetPrice = ItemPrice - (ItemPrice * DiscountPercent)
```

This works fine as long as `DiscountPercent` has a value. If it is Null, the `NetPrice` will also be set to Null, which is an error. The following code works correctly:

```
NetPrice = ItemPrice - (ItemPrice * nz(DiscountPercent))
```

Now, if `DiscountPercent` is Null, it is converted to `0` by the `NZ` function, and the `NetPrice` will be set to the full `ItemPrice`.

*The `nz()` function will accept two parameters separated by a comma. The second parameter is optional and allows you to define the replacement value if the first parameter is null. The syntax is:* `nz(TestValue, ReplacementValue)`. *The default replacement value is* `0` *except when used in a query where the default value (unless otherwise specified) is an empty string. When specifying the replacement value it must be treated as text. When using a literal replacement value, enclose the value in quotes. For example, if you would like to replace a null field value for a text field with the word "Unknown," use the following syntax:*

```
CurrentStatus = nz([StatusField], "Unknown")
```

## Select Case

Another way to evaluate expressions and run code based on them is the often under-utilized `Select Case` structure. It enables you to test for multiple values of a variable in a clean, easy-to-understand structure, and then run blocks of code depending on those values. Here's an example of a `Select Case` structure:


```
Select Case intOrderStatus
Case 1, 2
  `fascinating code for status 1 or 2
Case 3
  `riveting code for status 3
Case Else
  `hmm, it's some other value, just handle it
End Select
```

*code snippet The Select Case in ch07_CodeSnippets.txt*

Notice that there is no need for nested and indented `If` statements, and each `Case` block of code doesn't need a beginning or ending statement. Just to show the difference, the equivalent code using plain old `If` statements looks like this:

```
If intOrderStatus = 1 Or intOrderStatus = 2 Then
  `fascinating code for status 1 or 2
Else
  If intOrderStatus = 3 Then
    `riveting code for status 3
  Else
    `hmm, it's some other value, just handle it
  End If
Endif
```

This code is harder to read and understand. If you need to choose among multiple blocks of code depending on an expression's value, then `Select Case` is the preferred method.

# USING RECORDSETS

Recordset operations are one of the cornerstones of Access VBA, enabling you to directly read, update, add, and delete records in Access tables and queries. You explore all of this in the following sections.

## Opening Recordsets

Opening a recordset is easy, using either DAO or ADO (for more details about DAO and ADO, refer to Chapters 11 and 12). To open a recordset, you first need a reference to the current database, usually named `db`, and a `recordset` object. Here's how to accomplish that using DAO:

```
Dim db as DAO.Database
Set db = CurrentDB
Dim rst as DAO.Recordset
```

Now you need to actually open the recordset. There are three basic ways to open a recordset: by table, by query, and by SQL statement. Here's how to use a table directly:


```
Set rst = db.OpenRecordset(`tblMyTableName´)
```

*code snippet Opening A Recordset Using An Existing Table Or Query in ch07_CodeSnippets.txt*

If you have a query that already has some joined tables, selection criteria, or sort order, you can use it to open the recordset instead of using a table.


```
Set rst = db.OpenRecordset(`qryMyQueryName´)
```

*code snippet Opening A Recordset Using An Existing Table Or Query in ch07_CodeSnippets.txt*

Finally, you can open a recordset using your own SQL statement instead of using a preexisting query. Access evaluates and runs the query string on-the-fly.


```
Set rst = db.OpenRecordset(`Select * From tblMyTableName´)
```

*code snippet Opening A Recordset Using A SQL Statement in ch07_CodeSnippets.txt*

Now, you're probably thinking, "why is that last way any better than opening the table directly?" Your question is justified in this simple example. But using a recordset based on a SQL statement is much more flexible than using a table or query directly because you can modify the SQL statement in VBA code — like this:


```
Set rst = db.OpenRecordset(`Select * From tblMyTable Where MyKey = ` & Me!MyKey)
```

*code snippet Opening A Recordset Using SQL Statement With Filter in ch07_CodeSnippets.txt*

Now you're seeing some flexibility. This example opens a recordset limited to only those records that match the `MyKey` field on the form that contains this code. You can use values from your open forms or other recordsets as selection criteria, set flexible sort fields, and so on.

## Looping through Recordsets

When your recordset opens, it automatically points to the first record. One of the most common uses for a recordset is to loop through the records, top to bottom, and perform some action for each one. The action could be sending an e-mail, copying records across child tables, or whatever you need to do. Following is some example code to loop through all of the records in `tblBusiness`:


```
Dim db As DAO.Database
Dim rstBusiness As DAO.Recordset
```

```
Set db = CurrentDb

Set rstBusiness = db.OpenRecordset(`tblBusiness´)

Do While Not rstBusiness.EOF
    `do some code here with each business
    rstBusiness.MoveNext
Loop
```

*code snippet Looping Through Recordsets in ch07_CodeSnippets.txt*

Notice that the `EOF` (end of file) property of the recordset object is `True` when there are no more records in the recordset. It begins with a `True` value if there are no records in the recordset at all.

Remember to include the `.MoveNext` method before the `Loop` statement. If you omit it, your code drops into an infinite loop, repeatedly processing the first record, and not moving to the next one.

*Don't use recordset looping and updating to simply update a group of records in a table. It is much more efficient to build an update query with the same selection criteria to modify the records as a group.*

*If you need to perform an action on some of the records in a recordset, limit the recordset using a* `Where` *clause when you open it. Avoid testing the records with* `If` *statements inside your loop to determine which record(s) to perform the action against. It is much more efficient to exclude them from the recordset to begin with, rather than ignoring certain records in your loop.*

## Adding Records

To add a record using a recordset, the recordset type needs to be capable of updates. Most recordsets for Access (Access Control Entry, or ACE) tables , such as the one previously described, can be updated. However, if you need an updatable recordset for a SQL Server table opened via ODBC, you may need to also specify the `dbOpenDynaset` parameter value for the type. There's no harm in specifying it, even if it is an ACE table.


```
Set rst = db.OpenRecordset(`tblMyTable´, dbOpenDynaset)

With rst
  .AddNew
  !MyField1 = `A´
  !MyField2 = `B´
  .Update

End With
```

*code snippet Adding Records in ch07_CodeSnippets.txt*

The .AddNew method of the recordset object instantiates the new record in the table, and if the table is in ACE, also immediately assigns a new AutoNumber value to the record if the table contains one. Don't forget the final .Update, because without it, your record won't actually be added.

If the table is linked using ODBC (such as SQL Server), the AutoNumber/Identity value is not generated immediately when the .AddNew method runs. Instead, the Identity value is set after the .Update. This is discussed in the section "Copying Trees of Parent and Child Records" later in this chapter.

## Finding Records

To find a record in a recordset, use the FindFirst method. This is really just a way to reposition the current record pointer (cursor) to the first record that meets some criteria you specify. The criteria is specified like a WHERE clause in a SQL statement, except you omit the word WHERE. It looks like this:


```
rst.FindFirst `CustomerKey = ` & Me!CustomerKey
```

*code snippet Finding Records Using FindFirst in ch07_CodeSnippets.txt*

After you perform a FindFirst, you can check the NoMatch property of the recordset to determine whether you successfully found at least one matching record. You can also use the FindNext, FindPrevious, and FindLast methods to navigate to other records.

*In general, you shouldn't need to use the* Seek *method of a recordset. It may be slightly faster than* FindFirst*, but it won't work on a linked table without extra programming to open the table in a separate Workspace.*

## Updating Records

The code for updating records in a recordset is almost the same as for adding them. You may also need to find the correct record to update using FindFirst. If you find it successfully, you can update it. Here's an example:


```
Set rst = db.OpenRecordset(`tblMyTable´)

With rst
  .FindFirst `CustomerKey = ` & Me!CustomerKey
  If Not .NoMatch Then `we found the record

    .Edit
    !CustomerName = `ABC Construction´
    !CustomerStatus = 1
    .Update

  End If
End With
```

*code snippet Updating Records in ch07_CodeSnippets.txt*

*The* `With` *statement is purely a programming convenience. Instead of typing the name of the object every single time, you can use* `With <objectname>`*. After that, and until you use* `End With`*, any references with no object name, just a dot (.) or bang (!), are assumed to belong to the* `With` *object. You may want to improve the clarity of your code by not using it when you are trying to keep track of multiple recordsets.*

## USING MULTIPLE RECORDSETS

You can easily keep track of multiple open recordsets at once. Each one needs to be defined with a `Dim` statement and opened using the `OpenRecordset` method, and they are kept completely separate by Access. Each recordset has its own current record pointer (often called a cursor), end of file (`EOF`) and beginning of file (`BOF`) values, and so on.

This technique is necessary to perform the following trick: Copy a parent record and all of its child records into the same tables.

### Copying Trees of Parent and Child Records

Here's a task that can stump an Access programmer trying to tackle it for the first time. The problem is as follows: There are two tables, `tblPC` and `tblSpecification`. Each (parent) `PC` has many (child) `Specifications`. Many `PC`s have almost identical `Specifications`, but with slight variations. You need to write some code to copy one `PC` to another, along with all of its `Specifications`. The user will then manually update the copied `PC`'s `Specifications`.

At first, you might think that this seemingly simple problem can be performed using only queries. However, you soon run into a problem — you need to know the key of the newly copied `PC` so that you can assign the copied `Specifications` to it.

You can solve the problem by using multiple recordsets. Let's say that you have a continuous form showing a list of `PC`s and a Copy button at the bottom of the form. The desired functionality is to copy the `PC` record (with `` `Copy of ` `` as a prefix of the new `PC`) and also copy over all of its `Specification` records to the new `PC`:


```
On Error GoTo Error_Handler
Dim db As Database
Dim rstPC As DAO.Recordset
Dim rstSpecFrom As DAO.Recordset
Dim rstSpecTo As DAO.Recordset
Dim lngPCKey as Long

Set db = CurrentDb

If Not IsNull(Me.PCKey) Then

  Set rstPC = db.OpenRecordset(`tblPC", dbOpenDynaset)

  `copy the parent record and remember its key
  rstPC.AddNew
  rstPC!PCName = `Copy of` & Me!PCName
```

```
    rstPC.Update
    rstPC.Bookmark = rstPC.LastModified
    lngPCKey = rstPC!PCKey

    rstPC.Close
    Set rstPC = Nothing

    Set rstSpecTo = db.OpenRecordset(`tblSpecification", dbOpenDynaset)
    Set rstSpecFrom = db.OpenRecordset _
    (`Select * From tblSpecification Where PCKey = ` & Me!PCKey)

    Do While Not rstSpecFrom.EOF

      rstSpecTo.AddNew
      rstSpecTo!PCKey = lngPCKey `set to the new parent key
      rstSpecTo!SpecificationName = rstSpecFrom!SpecificationName
      rstSpecTo!SpecificationQty = rstSpecFrom!SpecificationQty
      rstSpecTo.Update

      rstSpecFrom.MoveNext
    Loop

    rstSpecTo.Close
    Set rstSpecTo = Nothing
    rstSpecFrom.Close
    Set rstSpecFrom = Nothing

    Me.Requery
  End If

  Exit_Procedure:
    On Error Resume Next
    Set db = Nothing
    Exit Sub

  Error_Handler:
    DisplayUnexpectedError Err.Number, Err.Description
    Resume Exit_Procedure
    Resume
```

*code snippet Copying Trees Of Parent And Child Records in ch07_CodeSnippets.txt*

It's important to understand the following key points about the preceding code:

- The variable `lngPCKey` stores the key of the newly created copy of the `PC` record. It's defined as a `Long` because this example assumes you are using AutoNumber keys, which are Long Integers.
- To find the record that was just created, you can use the `LastModified` property of the recordset. It returns a `Bookmark` to the record that was added. You can use this to find the new key.
- Setting the `Bookmark` property of a recordset positions it to that record.
- Use `Me.Requery` to requery the form's recordset so that the newly added record will be shown.

If your back-end database is Access (ACE), there's a simpler way to find the AutoNumber key of a newly added record. Anywhere between the `.AddNew` and the `.Update`, the AutoNumber key field of the table has already been set, so you can save it into a variable. Using this method, you don't need the `Bookmark` or `LastModified` properties. But be careful: If your back-end database is SQL Server or another ODBC database, the key won't be set until after the `.AddNew`, and your code won't work. The technique shown here is more flexible because it works for both ACE and ODBC databases.

Some developers are tempted to find the AutoNumber key with the highest value immediately after adding a record, thinking that this is a good way to find the new record. Don't do it! There are two problems with this approach. First, it fails in a multiuser environment if another user just happens to add a record in the fraction of a second after your code adds a new record but before it finds the "highest" value. Second, you shouldn't write code that depends on an AutoNumber key to have a certain value or sequence. If your database is ever switched to random keys (which can happen if it is replicated), this technique fails.

## Using Bookmark and RecordsetClone

In the previous example, there's one annoying behavior. After the form is requeried, the record selector is repositioned to the top of the list. That's disconcerting and can make it difficult to find the record that was just created.

It's easy to reposition the form to the new record — after all, you already know its key. Just after the `Me.Requery`, you add some code to find the new record in the just-requeried recordset and reposition the form to it.

To reposition the form, you use a `RecordsetClone`. This is a strange concept to developers when they first use it. Think of a `RecordsetClone` as a "twin" of the main recordset that the form is bound to. The nice thing about a `RecordsetClone` is that it has its own record cursor (with separate `FindFirst`, `EOF`, and so on), but it uses the exact same set of records as the form. You synchronize the "twin" recordsets with a `Bookmark`, which is essentially a pointer to an exact record in both recordsets.

If you find a record using a form's `RecordsetClone`, you can use the `Bookmark` to instantly reposition the form to that record. Here's the same code, with the extra repositioning section:


```
On Error GoTo Error_Handler
Dim db As Database
Dim rstPC As DAO.Recordset
Dim rstSpecFrom As DAO.Recordset
Dim rstSpecTo As DAO.Recordset
Dim lngPCKey as Long

Set db = CurrentDb

If Not IsNull(Me.PCKey) Then

  Set rstPC = db.OpenRecordset(`tblPC´, dbOpenDynaset)

  `copy the parent record and remember its key
  rstPC.AddNew
  rstPC!PCName = `Copy of ` & Me!PCName
 rstPC.Update
```

```
    rstPC.Bookmark = rstPC.LastModified
    lngPCKey = rstPC!PCKey

    rstPC.Close
    Set rstPC = Nothing

    Set rstSpecTo = db.OpenRecordset(`tblSpecification", dbOpenDynaset)
    Set rstSpecFrom = db.OpenRecordset _
    (`Select * From tblSpecification Where PCKey = ` & Me!PCKey)

    Do While Not rstSpecFrom.EOF

      rstSpecTo.AddNew
      rstSpecTo!PCKey = lngPCKey `set to the new parent key
      rstSpecTo!SpecificationName = rstSpecFrom!SpecificationName
      rstSpecTo!SpecificationQty = rstSpecFrom!SpecificationQty
      rstSpecTo.Update

      rstSpecFrom.MoveNext
    Loop

    rstSpecTo.Close
    Set rstSpecTo = Nothing
    rstSpecFrom.Close
    Set rstSpecFrom = Nothing

    Me.Requery

    `reposition form to new record
    Set rstPC = Me.RecordsetClone
    rstPC.FindFirst `PCKey = ` & lngPCKey
    If Not rstPC.EOF Then
      Me.Bookmark = rstPC.Bookmark
    End If
    rstPC.Close
    Set rstPC = Nothing

  End If

  Exit_Procedure:
    On Error Resume Next
    Set db = Nothing
    Exit Sub

  Error_Handler:
    DisplayUnexpectedError Err.Number, Err.Description
    Resume Exit_Procedure
    Resume
```

*code snippet Using Bookmark And RecordsetClone in ch07_CodeSnippets.txt*

You can reuse the `rstPC` recordset object for the repositioning logic because you are finished using it from earlier in the code, it has already been dimensioned, and it has an appropriate name. Of course, you need to close it and set it to `Nothing` again when you're done.

## Cleaning Up

Although Access VBA is supposed to automatically clean up local objects when a procedure ends, there is a history of errors and exceptions to this. So, programmers have learned that the safest

practice is to clean up everything themselves. It's boring, but it shows an attention to detail that is missing in many novice applications. To clean up recordsets, make sure that you:

- Close the recordset using the `.Close` method.
- Release the recordset object by setting it to `Nothing`.

These two easy steps may prevent strange problems and, more importantly, help you gain the respect of your peers.

## USING VBA IN FORMS AND REPORTS

Much of the power and flexibility of applications built using Access comes from the VBA code that you can use behind your forms and reports. Although code-less forms and reports can provide a lot of good functionality, they really shine when VBA coding techniques are added.

Access wizards provide a first look at VBA code behind forms and reports. However, wizard-built code is just scratching the surface. The following sub-sections offer some guidelines and techniques that will help you build extra functionality into your Access applications.

### All about Me

`Me` is a very special word in Access VBA. It is a reference to the form or report that your code is running in. For example, if you have some code behind the form `frmBusiness`, anytime you use `Me` in that code, you get a reference to the form object of `frmBusiness`.

This is a beautiful thing because there are many times that you need a reference to your own form or report, such as when you need to make it visible. You could refer to it directly, like this:

```
Forms!frmBusiness.Visible = True
```

Or, you can use the `Me` reference instead:

```
Me.Visible = True
```

Obviously, the `Me` reference is much shorter and easier to type. But there is a far greater reason to use `Me`: It enables you to move code from one form or report to another, where it automatically adapts to its new home.

The `Me` object is a full reference to a form object. Not only can you refer to it, but you can also pass it to other functions as a parameter. Simply define a function with a parameter with a `Form` data type, and you can pass the `Me` reference to it. You can see that used in the Better Record Finder technique shown in Appendix I.

It's good that you can pass `Me` as a parameter because it doesn't work outside the code of the form or report. Remember that `Me` refers to the form or report that it lives in, not the form or report that is currently active. So `Me` will not work in a standalone module (a module not behind a form or report).

## Referring to Controls

A control is any object that is placed on a form or report, such as a label, text box, combo box, image, checkbox, and so on. To refer to a control (for example, a bound text box named `BusinessName`) from the code behind a form or report, you use the following:

```
Me!BusinessName
```

So, if you want to clear out the `BusinessName` control, you use the following:

```
Me!BusinessName = Null
```

There has long been confusion in the VBA world about when to use a `!` (bang) and when to use a `.` (dot). There are more technical ways to describe it, but for the average VBA programmer there's a quick rule that works most of the time: If you (or any programmer) named it, you can use a bang. If Access named it, you use a dot. (Now, before all the VBA experts reading this get upset, please realize that it's only a general guideline. However, it does help.)

With that said, here's an exception. In the last few versions of Access, you can use either a bang or a dot when referring to controls on forms or reports, even though you named them. That's because of a little trick Access does: It turns all of your controls into properties of the form or report so they can be referred to with dots. This has a handy benefit: Access uses IntelliSense to prompt you with the possible properties and methods that are available for an object. So, in the `Me!BusinessName`, for example, you type **Me** and then . (dot); Access can then prompt you with every method and property for the object `Me`, including your control `BusinessName`.

*That little trick about using a dot instead of a bang for controls on forms and reports does not extend to fields in a recordset. To refer to them directly, you still need to use a bang, like this:* `rstMyRecordset!BusinessName`. *Or you can use other ways, such as the Fields collection:* `rstMyRecordset .Fields(˜BusinessName˝)`. *It is for that reason you should ensure that your control names are not the same as your field names. By default, a control bound to a field assumes the name of the field. A good programming practice is to immediately rename these controls using a Reddick naming convention prefix (*`txt` *for Text Box,* `cbo` *for combo box, and so on, depending on the type of control being used).*

## Referring to Subforms and Subreports

One of the most common questions about subforms and subreports is how to refer to their controls from the main form or report. Let's say that you have a form named `frmBusiness`, and on it you have a continuous subform named `fsubPayments`. Each `Business` record may have many `Payments`. You need to refer to a value of the calculated control `txtSumPaymentAmount` on the subform, but you want to do it from the main form `frmBusiness`.

The correct way to refer to `txtSumPaymentAmount` from `frmBusiness` is:

```
Me!fsubPayments.Form!txtSumPaymentAmount
```

The following table shows what each of the parts refers to:

| COMPONENT | DESCRIPTION |
|---|---|
| `Me` | The parent form where the code is running, which in this example is `frmBusiness`. |
| `!fsubPayments` | The control that contains the subform (its name usually defaults to the name of the subform object itself, but some programmers rename it). |
| `.Form` | This is the tricky piece. You need to drill down into the form that's in the control because that's where the controls in the subform live. The control on the main form named `fsubPayments` is just a Container — it doesn't contain the control you're looking for, but it does have this `Form` reference to use to get down into the subform itself. |
| `!txtSumPaymentAmount` | The control you want. You can even refer to controls that are on subforms on subforms (two levels down). |

Remember that you need to use the `Form` reference to get into the form that's in the subform control Container. For example, `frmAcontains` subform `fsubB` contains subform `fsubC`, which has control `txtC`. The full reference looks like this:

```
Me!fsubB.Form!fsubC.Form!txtC
```

You can also shift into reverse and refer to controls above a subform, using the `Parent` property. If some code in `fsubC` (at the bottom) needed to refer to control `txtA` on `frmA` (at the top), it would look like this:

```
Me.Parent.Parent!txtA
```

Note that you don't need the `Form` reference here because the `Parent` reference is already a `Form` reference.

## Closing Forms

Most of us will use the built-in Access wizards when creating certain types of buttons — one for closing your forms, for example. When you do, Access will create an Embedded Macro to perform this action. Often, you may not wish to use the macro but would like to use code so that you can run certain types of data validation or other operations before the form closes. While you could allow Access to convert the form's macros to code for you, you may just want to create the code yourself from scratch. The basic way to do that is with the following code:

```
DoCmd.Close
```

This method of the `DoCmd` object closes the active object, like your form. It doesn't get much simpler than that. Unfortunately, there is an obscure situation that will cause this code to fail to close the correct form. If you read the help documentation on `DoCmd.Close`, you'll see that if you don't

provide any parameters, it closes the active form. You might assume that the active form is the one containing this code; after all, you just clicked the Close button, so the form must be active. However, there are situations where another form is the active one.

You may, for example, have a hidden form on a timer that periodically does something. This is a technique that is often used in automatic log-off functionality, where a hidden form uses a timer to periodically check a table to determine whether it should shut down the application. The problem is that, when that timer fires and the code in the form checks the table, it becomes the active form. If you're unlucky enough for that to happen right when the Close button is clicked, the wrong form (the hidden one) will close instead of the form you intended.

Another situation is when the code in your closing routine reaches out and runs code in another form; this can make the other form active at that moment. The solution is to clarify the `DoCmd.Close` statement, like this:

```
DoCmd.Close ObjectType:=acForm, ObjectName:=Me.Name
```

This specifies that a form be closed, specifically the form to which this code belongs. If you get into the habit of using this syntax, the proper form will always close correctly.

## DEBUGGING VBA

Programming in VBA isn't easy. No matter how skilled you are there are times when you need help figuring out what the code is actually doing. Fortunately, VBA provides a rich and powerful debugging environment. You can stop the code at various times and for various reasons, view values of variables (and even change them), and step through your code line-by-line until you understand what's going on.

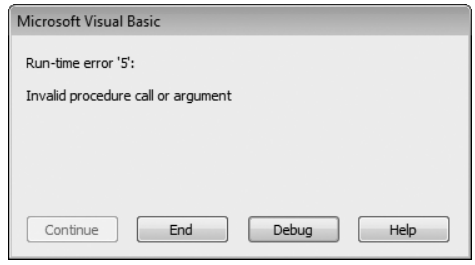


**FIGURE 7-2**

The main reason you need to debug your code is that Access has displayed an error message. (Hopefully you've put error handling in your code, which can make this activity easier.) Let's say you've coded a cool copy routine like the one shown earlier in this chapter. However, when you try it, Access displays an error. If you don't have error handling, a message box displays, as shown in Figure 7-2.

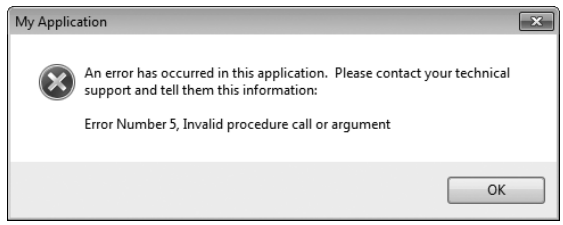


**FIGURE 7-3**

If you do have error handling, good job! Your error handling message box will display, as shown in Figure 7-3.

When Access displays your handled error message box, your code execution is suspended. To debug your code, press Ctrl+Break to interrupt code execution and display the dialog box shown in Figure 7-4.

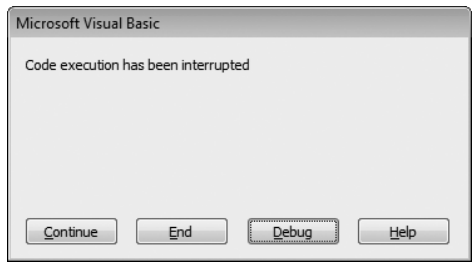


**FIGURE 7-4**

Whichever way you get there, you can finally click the Debug button. When you do, your code appears in the VBA code window. If you are not using error handling, the line of code that caused the error is indicated by an arrow and highlighted in yellow. If you are using error handling with the centralized `Msgbox` text and an extra `Resume` statement detailed later in this chapter, press F8 to step back to the procedure that contains the error. Then you can reposition to the specific line that caused the error, as shown in Figure 7-5.

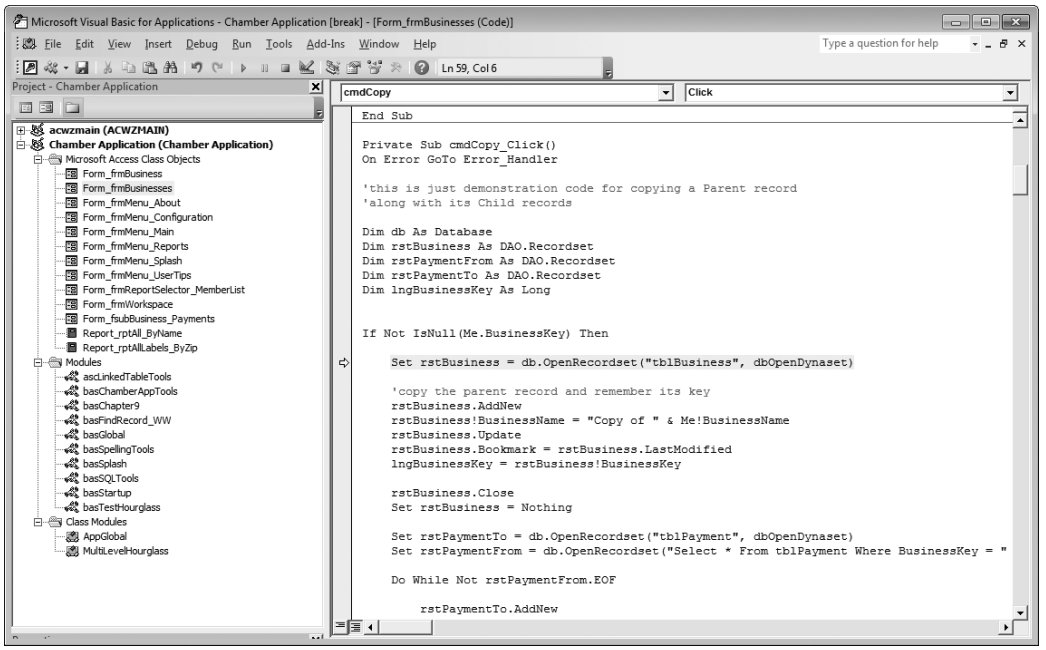


**FIGURE 7-5**

## INVESTIGATING VARIABLES

Now that you can see your code and the line that might be causing the problem, it's time to investigate. The error message — `Object variable or With block variable not set` — is a clue, but it doesn't tell you exactly what the problem is. The first step is to check the current values of the variables near the line that caused the error. Remember that your code is suspended, so all your variables are intact and able to report their values.

The quickest and easiest way to determine the value of a variable is to hover your mouse pointer over the variable name in the code window when your code is suspended. If the variable is part of a longer phrase, however, hovering may not work. For example, the variable `Me.BusinessKey` is simple enough to be "hoverable" (see Figure 7-6).

Because `BusinessKey` has a reasonable value, it doesn't seem to be the problem. To check variables or objects that are part of a more complex statement, highlight the portion you are interested in before you hover over it. In this example, just hovering over the object name `db` doesn't display anything, but after selecting `db`, hovering provides a value, as shown in Figure 7-7.

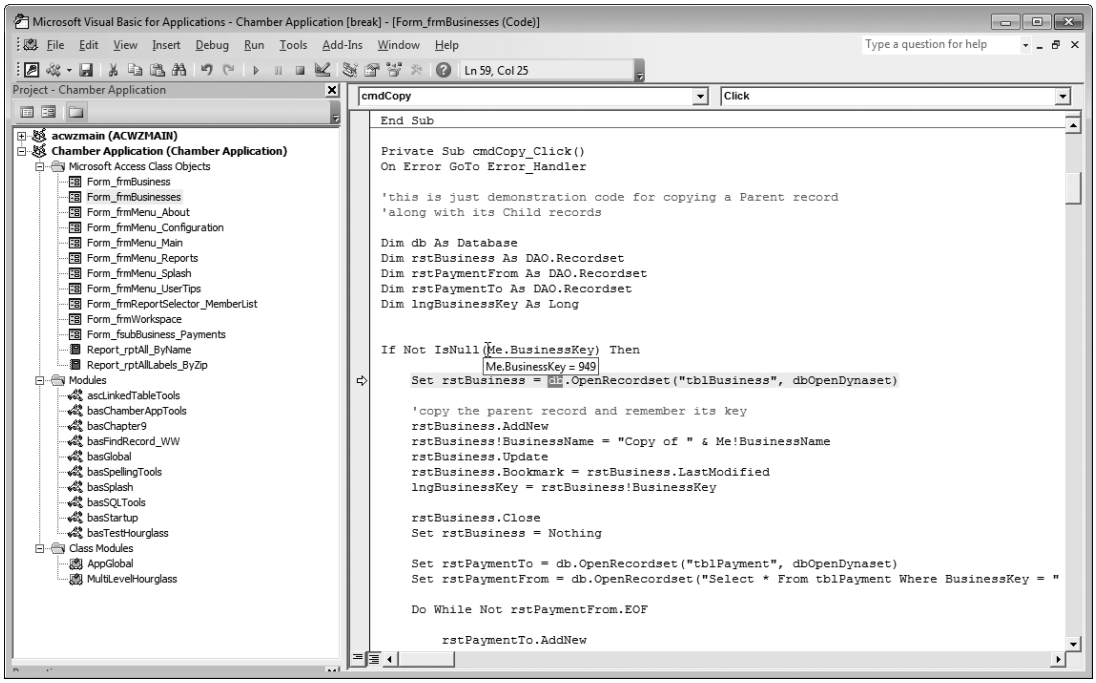


**FIGURE 7-6**

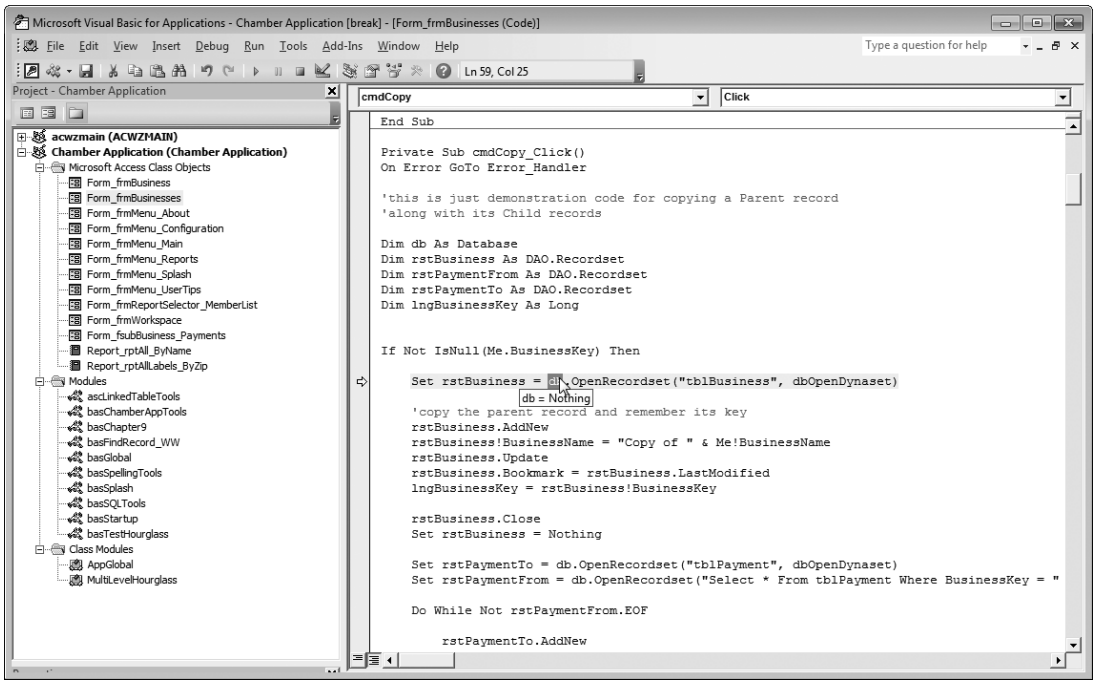


**FIGURE 7-7**

By checking the value of `db`, you can see that it is currently set to `Nothing`. This is Access's way of telling you that the `db` object reference hasn't been set to any value yet. Sure enough, when you look at the code, you can see that although you defined `db` using the line `Dim db As Database`, you forgot to include the line `Set db = CurrentDb`. Adding this line before the `OpenRecordset` line resolves the problem.

## When Hovering Isn't Enough — Using the Immediate Window

There are times when having the value of a variable pop up by hovering over it isn't sufficient. Perhaps the value doesn't fit in the pop-up, or maybe you need to copy the value to use it somewhere else. Or maybe you just want to look at it longer than the limited time the pop-up value displays. In those cases, you can use the Immediate window (instead of hovering) to view variable values.

If the Immediate window isn't already displayed, select View ➪ Immediate Window or press Ctrl+G to open it. Then you can ask Access to display the value of a variable using `?`, like this:

```
?Me.BusinessKey
```

When you press Enter, Access returns the value:

```
?Me.BusinessKey
949
```

*The `?` in the Immediate window is just a quick way of specifying `Debug.Print`.*

No matter how long this value is (it could be a very long string, for example), Access displays it here so that you can study it or even copy it into the clipboard to use somewhere else. This comes in handy when the variable contains a long SQL string that you want to try out by pasting it into a new query.

## Setting Breakpoints

Sometimes your code doesn't actually produce an error, but it still doesn't work correctly. In those cases, you need to stop the code yourself using breakpoints.

The easiest way to set a breakpoint is to click the gray area to the left of a line of code where you would like the code to suspend execution. This places a red dot to remind you where the breakpoint is set. Just before that line runs, your code will suspend and the code window will be displayed with that line highlighted in yellow, as shown in Figure 7-8.

At this point, you can investigate variable values as discussed previously in this chapter.

## Setting Watch Values

Sometimes you have no clue where the problem lies, so you don't know where to set the breakpoint. However, you may want to suspend your code and investigate whenever a certain variable is set to a certain value. To do this, you can use a watch value.

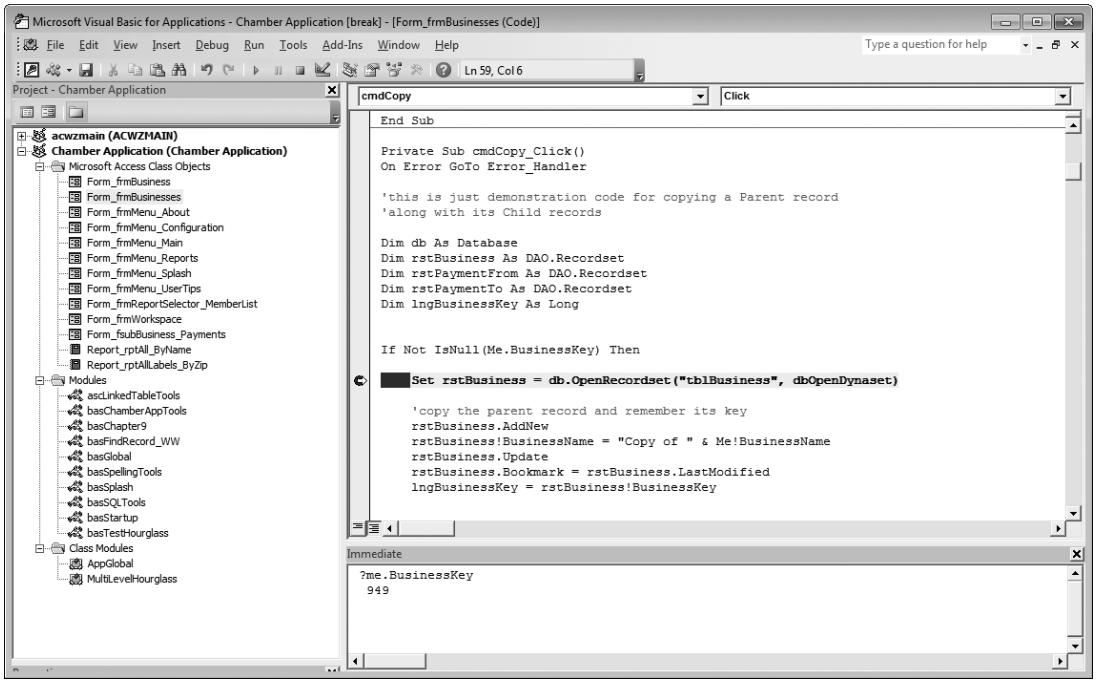


**FIGURE 7-8**

A watch value enables you to suspend execution of your code whenever a variable or object (or expression using a variable or object) changes or has a certain value. This is especially powerful in complex code scenarios where you are having trouble finding where your logic is going wrong. You create watch values using the Add Watch window (see Figure 7-9), which you can request using `Debug...Add Watch` or by right-clicking in the Watches window.

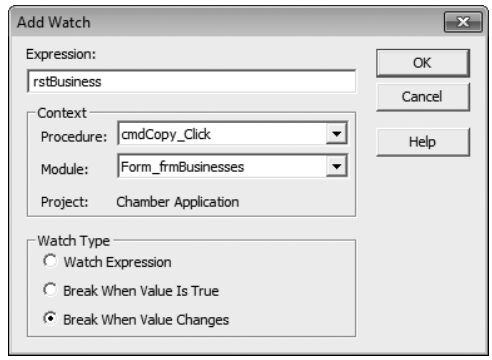


**FIGURE 7-9**

You can watch a single field, or you can type in an expression that uses multiple variables or values. Also, you can widen the context; it defaults to the procedure you are in, but you can expand on this to include all procedures. Finally, you can choose to merely watch the expression, to break (suspend your code execution) when the expression becomes `True` (for example, when `BusinessKey = 949`), or to break every time your expression changes. After you add your watch, it appears in the Watches window, as shown in Figure 7-10.

When the break condition you specify occurs, your code is displayed in the window. However, now you have an additional window, the Watches window. You can add more watch expressions here, if needed, and if you specify an object to watch (such as a form, report, recordset, and so on), you can even drill down to all of its properties using the plus sign (+) next to the object name.

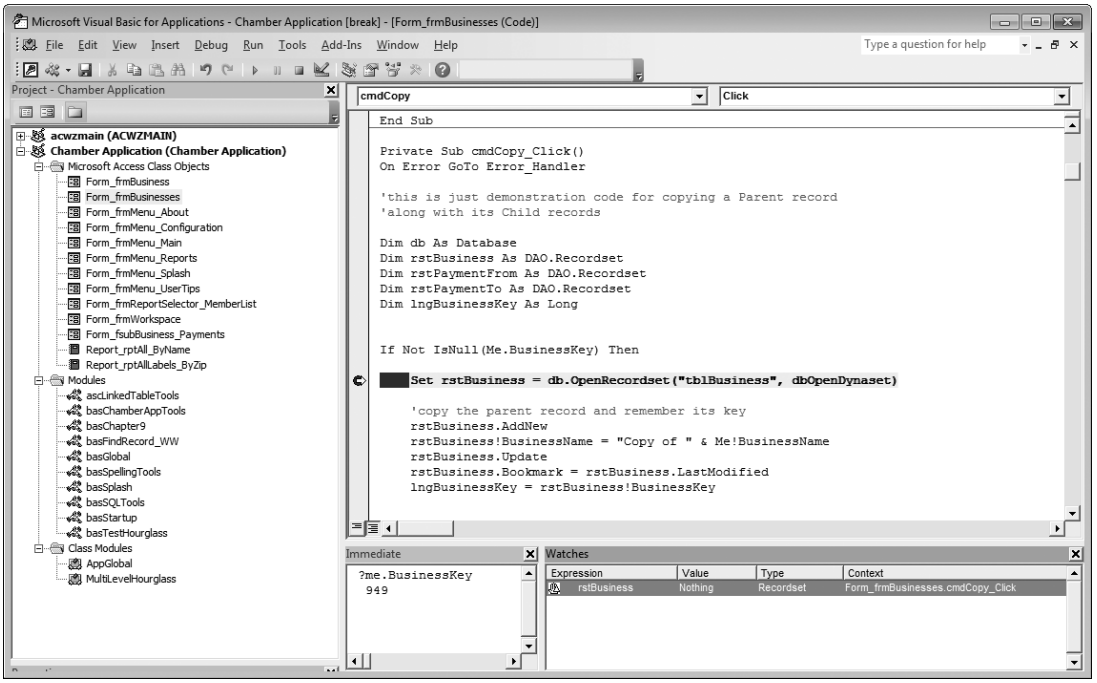


**FIGURE 7-10**

## Stopping Runaway Code

Everyone has done it. Every developer has created code that created an infinite loop. That's where your Access application just freezes, consuming all available computer power while it runs around the little race track that you accidentally created.

To stop your code in mid-execution, press Ctrl+Break. This suspends your code and drops you into the code window on whatever line that happens to be executing at that moment.

## Stepping through Your Code

Sometimes the only way to figure out a problem in your code is to actually run it line-by-line until you see where it goes wrong. You can use any of the preceding methods to stop your code, but there's nothing like getting your mind right into the logic by stepping through the code one line at a time.

You step through code by selecting Debug ➪ Step Into (or pressing F8). This debug command is the most common one to use because it's so basic. It runs the line of code that is highlighted, displays the next line that will be run, and awaits your next command. The `Step Into` and other `Step` commands are shown in Figure 7-11.

Sometimes the basic nature of `Step Into` is a problem. If the highlighted line of code is a call to another procedure (either a function or a sub), `Step Into` will do just that — it will dive into that procedure and highlight its first line.

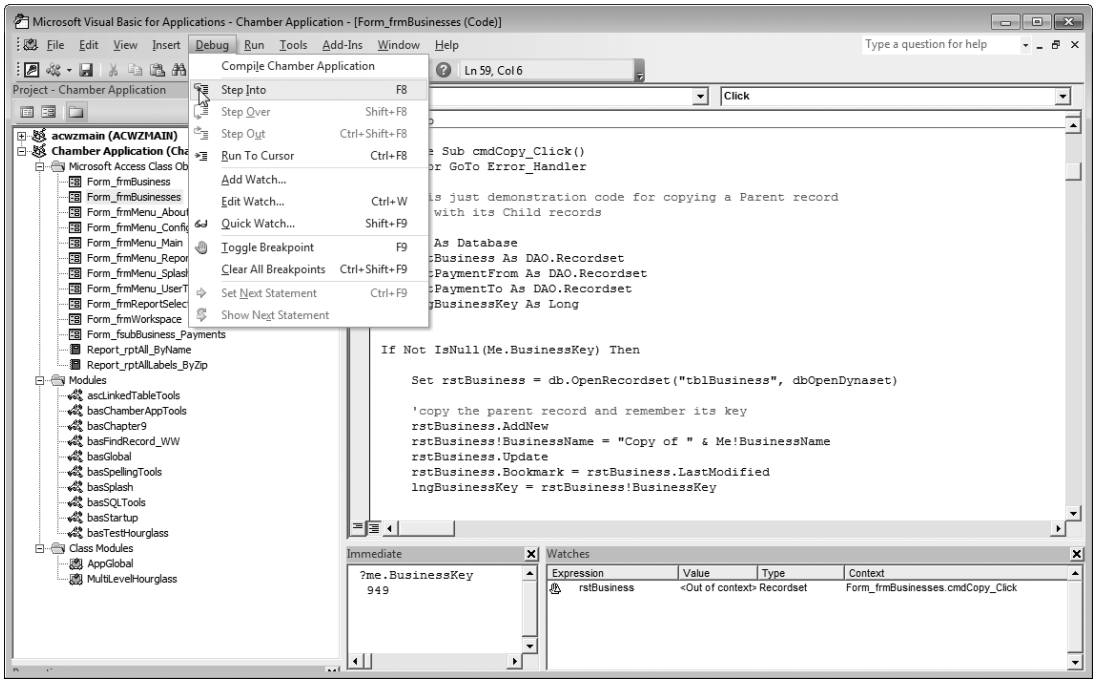


**FIGURE 7-11**

Now, maybe that's just what you want. But if you are following good programming practices, such as the coupling and cohesion guidelines presented earlier in this chapter, you have lots of small, fully tested functions that will be supremely boring and laborious to step through. After all, you know the error isn't in one of those, right?

The answer to this little problem is to use a cousin of `Step Into` called `Step Over` (Shift+F8). Its name isn't quite accurate, because when you use it, the highlighted line of code isn't really stepped over — it's actually stepped through. The line of code that's highlighted will run, even if it is a call to one of your functions or subs, and then the next line will be highlighted. The entire function or sub will run without stopping, so you don't have to step through all that boring code.

Also note that `Step Over` works exactly the same as `Step Into` for a normal line of code (not a call to another procedure). This means that you can get into the habit of leaning on the Shift key when you use F8, and you'll never need to step through called procedures unless you want to.

What if you accidentally use `Step Into` when you meant to use `Step Over`? Hope is not lost. By using the often-forgotten `Step Out` (Ctrl+Shift+F8), you can run the remainder of the current procedure without stopping, and automatically stop on the next line after your code returns to the calling procedure.

# COMMON VBA TECHNIQUES

Every Access developer will face some common VBA challenges at some point. There are simple and easy ways to handle drilling down to detail records, date math, rounding issues, and tricky string concatenation problems.

## Drilling Down with Double-Click

It's a good design practice to use read-only continuous forms to display multiple records and then allow your user to drill down to the detail of a single selected record. This action should have a button at the bottom of the form (called Detail, for example) that opens the detail form for the currently selected record.

For convenience and to compliment Windows standard behavior, it's also good to allow the user to drill down using double-click. Because you already have code behind the Detail button that opens the detail form, you can easily reuse that code:


```
Private Sub cmdDetail_Click()
On Error GoTo Error_Handler
    Dim stLinkCriteria As String
    If IsNull(Me!BusinessKey) Then
        EnableDisableControls
        GoTo Exit_Procedure
    End If

    gstrCallingForm = Me.Name
    stLinkCriteria = "[BusinessKey]= " & Me![BusinessKey]
    DoCmd.OpenForm FormName:="frmBusiness", _
        wherecondition:=stLinkCriteria
    Me.Visible = False

Exit_Procedure:
    On Error Resume Next
    Exit Sub
Error_Handler:
    DisplayUnexpectedError Err.Number, Err.Description
    Resume Exit_Procedure
    Resume

End Sub
```

*code snippet Drilling Down With Double-Click in ch07_CodeSnippets.txt*

Because this code is already written and tested, you only need to call it by name (`cmdDetail_Click`) when the user double-clicks a record. This is quite simple to do: You just add a double-click event to each text box on your detail form and add one line of code to each double-click procedure:


```
Private Sub txtBusinessName_DblClick(Cancel As Integer)
On Error GoTo Error_Handler

  cmdDetail_Click

Exit_Procedure:
  On Error Resume Next
  Exit Sub
```

```
Error_Handler:
  DisplayUnexpectedError Err.Number, Err.Description
  Resume Exit_Procedure
  Resume
End Sub
```

*code snippet Calling Code In Another Procedure in ch07_CodeSnippets.txt*

Here's a case where your actual code (1 line) is a lot shorter than all the error handling, but that line allows you to reuse the code you already have behind the Detail button.

Just because Access creates and names an event procedure (`cmdDetail_Click` in this case) doesn't mean you can't use it yourself. Just call it by typing its name as a statement in VBA.

To support double-click all the way across your row, you need to add the same code to each field's `Double-Click` event. That way, whichever field your user double-clicks, they'll drill down to the detail record.

Now, there's only one more thing to add. Users will often double-click the Record Selector itself (the arrow to the left of the current record) when they want to drill down to the record's detail. Surprisingly, the event that fires in this case is not related to the detail section of the continuous form; instead, the Form's double-click event will fire. To support double-click of the Record Selector, you can use this code behind the Form's `On Double Click` event:

```
Private Sub Form_DblClick(Cancel As Integer)
On Error GoTo Error_Handler

  cmdDetail_Click

Exit_Procedure:
  On Error Resume Next
  Exit Sub

Error_Handler:
  DisplayUnexpectedError Err.Number, Err.Description
  Resume Exit_Procedure
  Resume
End Sub
```

## Date Handling

The way Access stores and manipulates dates can be a source of confusion to developers, especially those who remember the older database methods of storing days, months, and years in date fields. Access handles dates in an elegant, easy-to-use manner.

### How Access Stores Dates and Times

Access stores a particular date as the number of days that have elapsed since an arbitrary starting "zero date" (December 30, 1899). You can prove this to yourself by typing the following in the Immediate window (you can bring up the Immediate window in Access using Ctrl+G).

```
?CLng(#12/31/1899#)
1
```

The `CLng` function converts an expression to a Long Integer. To this question, Access will answer with 1, meaning that 1 day elapsed since December 30, 1899. Of course, Access can handle dates before this date; they're stored as negative integers. If you want to see how many days have elapsed since that special zero date, try this:

```
?CLng(Date)
```

Access can perform date math very easily because internally it doesn't store a date as days, months, and years. It just stores the number of days since the zero date and converts that value to an understandable date format only when the date needs to be displayed. But the date storage technique that Access uses goes even farther. Access can also store the time of day in the same date field. To do this, Access uses the decimal portion (the numbers after the decimal point) to store a fraction of a day. For example, 12:00 noon is stored as .5 (half way through the day), and 6 a.m. is stored as .25. Again, you can see this for yourself by typing this into the Immediate window:

```
?CDbl(Now)
```

There are a couple of things to note here. The first is that you now need to use `CDbl` (Convert to Double Precision Number) so that you can see the decimal portion (the time portion) that is returned by the `Now` function. The other is that each time you run this command, you'll see that the decimal portion changes because time is elapsing.

*When you're storing the current date in a table, be sure to use the* `Date` *function. If you use* `Now`*, you'll also get a time component, which may cause incorrect results when you use dates in your query criteria. For example, if your query selects records where a date field is* `<=4/28/2007`*, then any records with a date of 4/28/2007 should be returned. However, if they were stored with a decimal time component (by using* `Now` *instead of* `Date`*), the value will be fractionally greater than 4/28/2007 and that date won't be returned.*

## Simple Date Math

To add or subtract calendar time from a date field, use the `DateAdd` function. For example, to add 1 month to today's date, use:

```
?dateadd(˝m˝,1,Date)
```

To subtract, use a negative number for the second parameter, `Number`. You can use different units of calendar time for the Interval parameter, such as `˝d˝` for days, `˝ww˝` for weeks, `˝q˝` for quarters, and so on. Be careful when adding or subtracting years; you have to use `˝yyyy˝`, not just `˝y˝`. The interval of `˝y˝` is day of year, which acts just like day in the `DateAdd` function.

Here's an example of date math. It computes the last day of a month by finding the first day of the next month, and then subtracting 1 day.


```
Public Function LastDateofMonth(StartDate As Date)

On Error GoTo Error_Handler

  Dim dtNextMonth As Date
  Dim dtNewDate As Date

  `add a month to the start date
  dtNextMonth = DateAdd(`m`, 1, StartDate)

  `build a date
  dtNewDate = CDate((DatePart(`m`, dtNextMonth)) & _
  `/01/` & (DatePart(`yyyy`, dtNextMonth)))

  `subtract a day
  LastDateofMonth = dtNewDate -1

Exit_Procedure:
  Exit Function
Error_Handler:
  DisplayUnexpectedError Err.Number, Err.Description
  Resume Exit_Procedure
  Resume

End Function
```

*code snippet Function To Find Last Date Of Month in ch07_CodeSnippets.txt*

Note the use of `CDate`, which converts any expression that can be interpreted as a date into an actual date data type. You can use `IsDate` to check whether an expression can be interpreted as a date. Also note how the `DatePart` function is used to break up a date into string components for Month, Year, and so on.

## Handling Rounding Issues

Rounding problems are among the most difficult to understand and debug. They usually occur when adding up money values, but they can also happen in any math where a series of values is expected to add up correctly.

### Rounding of Sums

One basic issue is not Access-related at all, but rather an issue whenever you add up a list of rounded numbers. For example, take a list of numbers that each represent one third of a dollar. If you add them up, you'll get 99 cents because the value of each portion (.33333333...) was truncated to .33.

```
.33
.33
.33
.99
```

A common place for this to show up is in a list of percentages that are supposed to total 100 percent. They often don't because some precision was lost in the list. Then, you are faced with a decision — add up the actual numbers and show a total that's not 100, or just hard-code 100 percent so that it looks right. Most of the time, you will want to ensure that your final value is correct and eliminate an accumulated rounding error. This can be accomplished by eliminating rounding of the

individual components of your formula and reserving rounding to the final outcome. For display purposes, each component can be formatted in the displayed control to the number of decimal places you wish to show but the underlying control will still contain the full value it represents.

## Rounding Errors Caused by Floating Point Numbers

Another kind of rounding error comes from the way Access stores numbers in floating-point fields. These fields cannot store certain numbers without losing some precision, so totals based on them may be slightly wrong. The best way to avoid this kind of rounding error is to use the `Currency` data type for fields when they need to hold money values (as you might expect), or the `Decimal` type for any other numeric values that you want to use in calculations. The `Currency` data type is somewhat misnamed; it really can hold any decimal value.

*Access uses the word "Currency" for both a data type and a format. This is unfortunate because they really are two different things. The Currency data type is a method of storing the numeric values in a table. The Currency format affects only the display of numeric data. The two can be used independently or together.*

## Access Rounding Functions

Access has a built-in function (`Round`) to round numbers, but it may not work the way you expect. Most people think that any decimal ending in 5 should round up to the next higher number. However, Access uses a form of scientific rounding that works like this:

- If the digit to be rounded is 0 through 4, round down to the lower number.
- If the digit to be rounded is 6 through 9, round up to the higher number.
- If the digit to be rounded is 5, round up if digit to the left is odd, and round down if the digit to the left is even.

This last rule is what surprises a lot of developers. Using this rule, `Round` gives the following results:

```
?round(1.5)
2
?round(2.5)
2
```

Yes, that's right. Both 1.5 and 2.5 round to 2 using the built-in `Round` function in Access VBA, because 1 is odd (round up) and 2 is even (round down). Here's another example:

```
?round(1.545,2)

1.54
?round(1.555,2)

1.56
```

In this example, .545 rounds down, but .555 rounds up, for the same reason. Because this can cause some trouble in business applications, developers have taken to writing their own rounding functions that behave the way business people expect. Here's an example of a function that rounds a trailing 5 upward to a specified number of decimal places:


```
Public Function RoundCurr(OriginalValue As Currency, Optional _
NumberOfDecimals As Integer) As Currency
On Error GoTo Error_Handler

`returns a currency value rounded to the specified number of decimals of
`the Original Value

If IsMissing(NumberOfDecimals) Then
  NumberOfDecimals = 0
End If

RoundCurr = Int((OriginalValue * (10 ^ NumberOfDecimals)) + 0.5) _
  / (10 ^ NumberOfDecimals)

Exit_Procedure:
  Exit Function
Error_Handler:
  DisplayUnexpectedError Err.Number, Err.Description
  Resume Exit_Procedure

End Function
```

*code snippet Substitute Rounding Function in ch07_CodeSnippets.txt*

This function can be placed in any module in your application and used whenever you want the business-style rounding that most users expect. Note that if you don't specify the number of decimals you would like, the function will assume that you want none and will return a whole number.

## STRING CONCATENATION TECHNIQUES

Sooner or later, you'll need to join (concatenate) two strings together. The operator for performing concatenation is `&`. You may be tempted to say "and" when you see this symbol, but it really means "concatenate with." A classic example is joining First Name with Last Name, like this:

```
strFullName = FirstName & " " & LastName
```

This results in the first name and last name together in one string, as in "Tom Smith."

### The Difference between & and +

There are times when you may need to concatenate something to a string, but only if the string actually has a value. For example, you may want to include the middle initial in a person's full name. If you write code like this:

```
strFullName = FirstName & " " & MiddleInitial & " " & LastName
```

you will have a small problem. People with no middle name (Null in the table) will have two spaces between their first and last names, like this:

```
Tom  Smith
```

Fortunately, there is another concatenation operator: +. The technical explanation of this operator is "concatenation with Null propagation." That's a great phrase to impress your friends with at parties, but an easier explanation is that it concatenates two strings just as the & operator does, but only if both strings have a value. If either one is Null, the result of the whole concatenation operation is Null.

Using the FullName example, the goal is to have only one space separating first and last names if there is no middle initial. Using +, you can tack on the extra space only if the middlenName is not null:

```
MiddleName + ` `
```

The whole thing looks like this:

```
strFullName = FirstName & ` ″ & (MiddleInitial + ` `) & LastName
```

As shown, you can use parentheses to ensure that the operations happen in the correct order. In this case, the inner phrase — `(MiddleInitial + ` `)` — will evaluate to the middle initial plus a space, or to Null (if there is no middle initial). Then, the rest of the statement will be performed.

## String Concatenation Example

Here is another example that you can use in your code. It concatenates the city, state, postal code (ZIP Code), and nation into one text field. This can be handy if you want to show a simulation of an address label on a form or report.


```
Function CitySt ZIPNat(City As Variant, State As Variant, ZIP As Variant, _
  Nation As Variant) As Variant
On Error GoTo Error_Handler

CitySt ZIPNat = City & (`, ` + State) & (` ` + ZIP) & _
(IIf(Nation = `US″ Or Nation = `CA″, Null, (` ` + Nation)))

Exit_Procedure:
  Exit Function
Error_Handler:
  MsgBox Err.Number & `, ` & Err.Description
  Resume Exit_Procedure
  Resume

End Function
```

*code snippet String Concatenation Function in ch07_CodeSnippets.txt*

You can try it out by calling it in the Immediate window like this:

```
?CitySt ZIPNat(`Seattle″, `WA″, `98011″, `US″)

Seattle, WA 98011
```

Notice that this code also tacks on the Nation at the end of the string, but only if it isn't US or CA (the ISO standard nation codes for USA and Canada, respectively). This enables you to use this function for both domestic and foreign addresses.

# VBA ERROR HANDLING

When programmers use the term "error handling," they really mean graceful or planned error handling. After all, Access takes some kind of action for any error that it encounters in your code. *Graceful* error handling includes the following:

- Quietly absorbing expected errors so the user never sees them
- Displaying a "friendly" message to the user for unexpected errors, and closing the procedure properly

Error handling in Access VBA involves adding code to every procedure — both subroutines and functions — to take specific actions when Access encounters an error. This is called handling or trapping the error. (Some developers call the encounter with an error: *throwing* an error. Error handling is the code that *catches* the error and handles it properly, either by hiding it from the users or by explaining it to them.)

This section provides techniques to handle several types of expected and unexpected errors so that your applications look and feel more professional to your users. But first, you'll explore why you should use error handling at all. Many Access developers see it as a mundane chore, but there are good reasons for including error handling in every procedure you write.

## Why Use Error Handling?

Without error-handling code, Access treats all errors equally, displaying unfriendly or vague error messages and abruptly ending procedures. Even worse, if you are using the runtime mode of Access, the entire application closes. This is not what you want users to experience.

Figure 7-12 shows an example of an error message that Access displays if you attempt to divide a number by zero in your application. Sure, technically it indicates what happened, but what is the user supposed to do about it? And what if he clicks the Debug button? If he's running an MDB/ACCDB instead of an MDE/ACCDE, he'll be looking at your code!

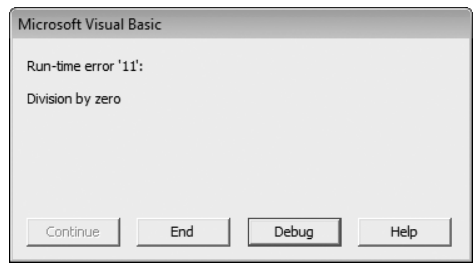


**FIGURE 7-12**

When Access encounters an error, it abruptly ends the procedure. It does not run another line of code; it just terminates the function or sub that contains the error. So, it can often leave things hanging — open objects, open forms, the mouse pointer in hourglass mode, warnings turned off, and so on.

*Amateur or pro? When your code is being evaluated by another programmer, one of the easiest things for him to check is whether you have proper error handling. No matter how good your code is, without error handling you may look like a beginner. It's worth making sure that every procedure has error handling.*

Now for the good news: Error handling isn't difficult. By using some easy techniques and code templates, you can make sure that your application never suffers an unhandled error. If you establish a standard way to handle errors, you can make it easy to implement in every procedure you write. It may not be fun or glamorous, but it will certainly make your application better.

## Two Kinds of Errors: Unexpected and Expected

All errors that your Access application may encounter fall into one of two categories: unexpected and expected. The following sections explain these two categories and what your application should do when errors occur in each of them.

### Handling Unexpected Errors

Unexpected errors are ones that you have no way of predicting, and that under normal circumstances should not occur. When your application encounters an unexpected error (for example, divide by zero or a missing object), and no error handling is in effect, Access displays an error message like the one shown earlier and abruptly ends the procedure.

The goal of error handling in this case is not to solve the problem the error is indicating — there's nothing you can do about that now. Your code has tripped on an error and fallen down. The only thing you can do is let the user know what happened calmly and in plain language. Figure 7-13 is an example of what your error message might look like.

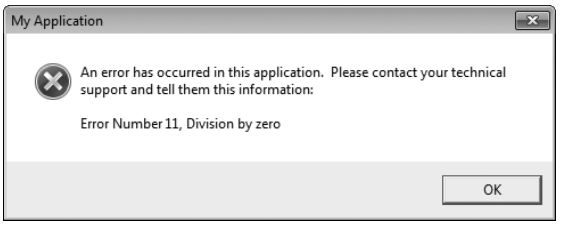


**FIGURE 7-13**

There are several differences between the error message Access shows and the "handled" error message you can show:

- You can specify the title of the message box instead of displaying "Microsoft Visual Basic" or "Microsoft Access."
- You can show an icon to have a stronger impact.
- You can add a text explanation. You can even mention your phone number or other contact information.
- You can format the error message with blank lines, commas, and so on.
- Your user can't enter debug mode and look at your code.

### Absorbing Expected Errors

Some errors can be expected during normal operation. One such error happens in your application whenever the `On Open` event of a report is canceled. This occurs when you display a form to prompt the user for selection criteria during the `On Open` event, and the user decides to cancel the report. This report criteria technique is described in Chapter 14.

There are other errors that you can expect. Maybe you expect a certain file to be on the hard drive, but it isn't. Maybe you expect a form to be open, but somehow it has been closed. These kinds of errors can usually be absorbed by your application, never allowing the user to see them.

In these situations, your code should just ignore the error and keep going. Whenever Access encounters an error, it makes an error number available to your code. To absorb an expected error, you add an `If` statement to check if the error number matches the number you expect. If it matches, you can just `Resume Next` to continue to the next line of code without bothering the user with an error dialog box. If it doesn't match, you can drop into your normal error handling.

Next, we explore some basic error-handling code that can be used to handle both expected and unexpected errors in your application. Then we'll look more specifically at expected errors in the section "More on Absorbing Expected Errors."

## Basic Error Handling

Let's start with the basics. Here's some code that you could add to every procedure to build in easy, no-frills error handling:


```
Public Function MyFunction
On Error GoTo Error_Handler

  `your function code goes here

Exit_Procedure:
  Exit Function

Error_Handler:
  MsgBox "An error has occurred in this application. " _
  & "Please contact your technical support and " _
  & "tell them the following information:" _
  & vbCrLf & vbCrLf & "Error Number " & Err.Number & ", " _
  & Err.Description, _
  Buttons:=vbCritical

  Resume Exit_Procedure
End Function
```

*code snippet Basic Error Handling in ch07_CodeSnippets.txt*

Let's take a look at some important lines in the code, beginning with the following:

```
On Error GoTo Error_Handler
```

The `On Error GoTo` statement in VBA tells the code to jump to a particular line in the procedure whenever an error is encountered. It sets up this directive, which remains in effect until it is replaced by another `On Error` statement or until the procedure ends. In this example, when any error is encountered, the code execution jumps to the line named `Error_Handler`.

*In the early days of Basic and other procedural languages, lines were numbered, not named. For example, your code might have a line* `GOTO 1100`. *In VBA, you still have the* `GoTo` *statement, but instead of numbering the lines, you can give them meaningful names like* `Exit_Procedure`.

If no error occurs throughout the main body of the procedure, the execution of the code falls through to this point:

```
Exit_Procedure:
  Exit Function
```

and the `Exit Function` will run. As its name implies, the `Exit Function` statement exits this function immediately, and no lines after it will be executed. Note that if this procedure is a sub instead of a function, you use `Exit Sub` instead.

This same `Exit_Procedure` line is also executed after any unexpected errors are handled:

```
Error_Handler:
  MsgBox "An error has occurred in this application. " _
  & "Please contact your technical support and " _
  & "tell them this information:" _
  & vbCrLf & vbCrLf & "Error Number " & Err.Number & ", " _
  & Err.Description, _
  Buttons:=vbCritical
```

If an error occurs, execution jumps to the `Error_Handler` line, and a message box is displayed to the user. When the user clicks OK (her only choice), the code execution is redirected back to the `Exit_Procedure` line:

```
Resume Exit_Procedure
```

and your code exits the procedure.

With this technique, execution of the code falls through to the `Exit_Procedure` code and the function exits normally, as long as no errors are encountered. However, if an error is encountered, the execution is redirected to the error-handling section.

*In early versions of Access, the labels for the* `Exit_Procedure` *and* `Error_Handler` *sections had to be unique in the entire module. This forced programmers to use labels such as* `Exit_MyFunction` *and* `Error_MyFunction`*. In recent versions of Access, these labels may be duplicated in different procedures. This is a great improvement because now you can copy and paste error-handling code into each procedure with almost no modification.*

This is the most basic error handling you can include in your code. However, there's one word that you can add to make your code much easier to debug: `Resume`. Yes, it's just one word, but it can work wonders when you are trying to make your code easier to debug.

## Basic Error Handling with an Extra Resume

One of the problems with basic error handling is that when an error does occur, you have no easy way of knowing the exact line that caused the error. After all, your procedure may have dozens or hundreds of lines of code. When you see the error message, the execution of your code has already jumped to your error handler routine and displayed the message box; you may not be able to tell

which line caused the problem. Many programmers rerun the code, using debug mode, to step through the code to try to find the offending line.

But there is a much easier way to find that error-producing line of code: Just add a `Resume` line after the `Resume Exit_Procedure`.

You're probably thinking, "Why would you add an extra `Resume` right after another `Resume Exit_Procedure`? The extra `Resume` will never run!" You're right. It will never run under *normal* circumstances. But it will run if you ask it to. If your application encounters an error, you can override the next line that will run. In debug mode, you can just change the next line to be executed to your extra `Resume`. The `Resume Exit_Procedure` statement is skipped entirely. The following code is identical to the basic code shown previously, but with that one extra `Resume`.


```
Public Function MyFunction()
On Error GoTo Error_Handler

  Dim varReturnVal As Variant

  `your function code goes here

Exit_Procedure:
  Exit Function `or Exit Sub if this is a Sub

Error_Handler:
  MsgBox "An error has occurred in this application. " _
  & "Please contact your technical support and tell them this information:" _
  & vbCrLf & vbCrLf & "Error Number " & Err.Number & ", " _
  & Err.Description, _
  Buttons:=vbCritical, title:="My Application"
  Resume Exit_Procedure
  Resume

End Function
```

*code snippet Basic Error Handling With An Extra Resume in ch07_CodeSnippets.txt*

Under normal operation, the extra `Resume` never runs because the line before it transfers execution of the code elsewhere. It comes into play only when you manually cause it to run. To do this, you can do something that is rarely done in debug mode: Move the execution point in the code to a different statement.

Here's how the extra `Resume` works. Say your code is supposed to open a report, but there's a problem: The report name you specified doesn't exist. Your code might look like this:

```
Private Sub cmdPreview_Click()
On Error GoTo Error_Handler

  If Me!lstReport.Column(3) & "" <> "" Then
    DoCmd.OpenReport ReportName:=Me!lstReport.Column(3),
    View:=acViewPreview
  End If

  `Update the Last Run Date of the report
  DoCmd.SetWarnings False
  DoCmd.RunSQL "UPDATE tsysReport " _
  & "SET tsysReport.DtLastRan = Date() " _
  & "WHERE tsysReport.RptKey = " & Me.lstReport
  DoCmd.SetWarnings True
```

```
Exit_Procedure:
  On Error Resume Next
  DoCmd.SetWarnings True
  Exit Sub

Error_Handler:
  MsgBox "An error has occurred in this application. " _
  & "Please contact your technical support and " _
  & "tell them the following information:" _
  & vbCrLf & vbCrLf & "Error Number " & Err.Number & ", " &
  Err.Description, _
  Buttons:=vbCritical, title:="My Application"
  Resume Exit_Procedure
  Resume

End Sub
```

When you run your code, an error message appears, as shown in Figure 7-14.

Instead of clicking OK as your user would do, press Ctrl+Break on your keyboard. A Visual Basic dialog box appears, as shown in Figure 7-15.

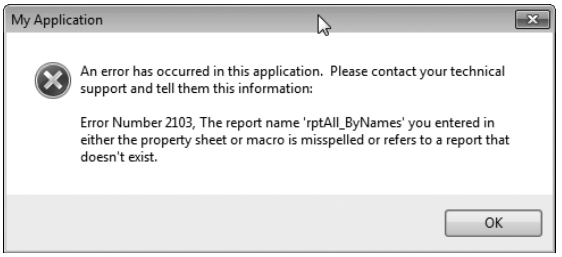


**FIGURE 7-14**

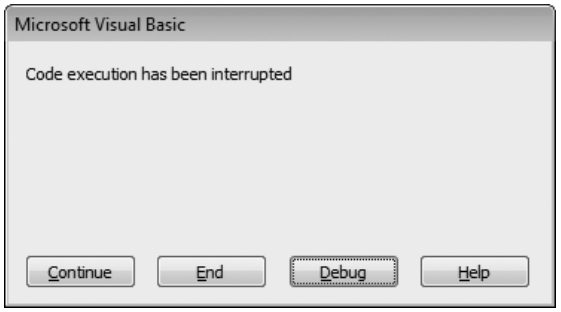


**FIGURE 7-15**

*This extra* `Resume` *technique won't work in an Access runtime application because in runtime versions no design modes are allowed, including the VBA code editor. It also won't work in an Access MDE or ACCDE because all VBA source code is not accessible from within those applications.*

Now click the Debug button. The code displays in the Code window, as shown in Figure 7-16.

The `Resume Exit_Procedure` statement will be indicated by an arrow and highlighted in yellow. This is the statement that will execute next if you continue normally. But instead of letting it run,

you take control, using your mouse to drag the yellow arrow down one line to the extra `Resume` line. By doing this, you indicate that you want the `Resume` line to run next.

*Instead of using the mouse, you can click or arrow down to the* `Resume` *line, and then use* `Debug...Set Next` *statement (Ctrl+F9 on your keyboard). As usual in Access, there are several ways to do the same thing.*

Now, the yellow arrow will be pointed at the `Resume` statement, as shown in Figure 7-17.

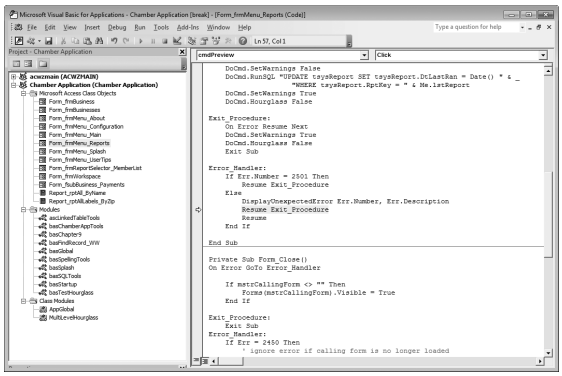

**FIGURE 7-16**

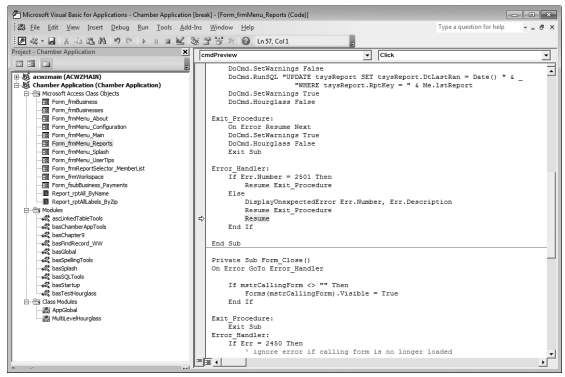

**FIGURE 7-17**

Now, you want the `Resume` statement to run in order to retry the statement that caused the error. Press F8 to run the next line of code (your `Resume`) and stop. Or, you can choose `Debug...Step Into` from the menu.

The exact line that caused the error will now be indicated by an arrow, as shown in Figure 7-18. That was easy, wasn't it?

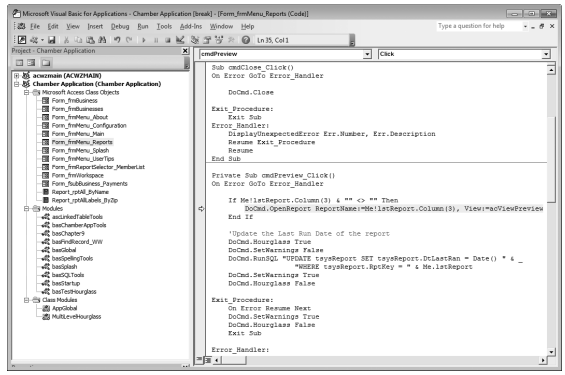

**FIGURE 7-18**

Now, admittedly, this is a simple example. You probably could have determined which line caused the error just by looking at the error description. However, when your procedures contain pages of code, often with coding loops, complex logic, and similar statements, this extra `Resume` technique comes in handy. It can save you many hours of time while you are debugging your VBA code.

The extra `Resume` doesn't cause any harm in your code, so you can leave it in every procedure even when you deliver your application. Also, if a technically savvy client encounters an unexpected error

and she's running an MDB or ACCDB (not an MDE or ACCDE), you can walk the client through this process to help determine what caused the problem in the client's environment. As you know, what works on your PC doesn't always work when your user is running it.

## Basic Error Handling with a Centralized Message

There's one more thing you can do to make your error-handling code even easier to maintain. Instead of repeating the code to display the message box in every procedure, you can move it to a reusable function that handles it consistently every time.

The following code basically tells the user that an unexpected error occurred. It is needed in every procedure in your application.


```
MsgBox "An error has occurred in this application. " _
& "Please contact your technical support and " _
& "tell them this information:" _
& vbCrLf & vbCrLf & "Error Number " & Err.Number & ", " &
Err.Description, _Buttons:=vbCritical, title:="My Application"
```

Instead, you can centralize this in a function that's called using one line:

```
DisplayUnexpectedError Err.Number, Err.Description
```

Much cleaner! This mundane bit of business is handled with just one line. Now you just need the function that it calls, `DisplayUnexpectedError`. Here it is:

```
Public Sub DisplayUnexpectedError(ErrorNumber As String, _
ErrorDescription As String)
'Note: since this is a universal error handling procedure,
'it does not have error handling

MsgBox "An error has occurred in this application. " _
& "Please contact your technical support and " _
& "tell them this information: " _
& vbCrLf & vbCrLf & "Error Number " & ErrorNumber & ", " &
ErrorDescription, _Buttons:=vbCritical, title:="My Application"

End Sub
```

*code snippet Error Handling With A Centralized Message in ch07_CodeSnippets.txt*

In this code, `Err.Number` is replaced with `ErrorNumber`, and `Err.Description` with `ErrorDescription`. That's necessary because you're calling a different function and sending in those two values as parameters.

This technique cleans up and shortens your code a lot, but there is an even better reason to use it. If you ever want to change the text of the message displayed to the user, you have only to change it in one place — the `DisplayUnexpectedError` function — instead of searching and replacing it throughout all your procedures.

Note that if you use this centralized message technique, you'll have one more step in your code when you debug using the extra `Resume` statement shown earlier. After you click Debug, the `End Sub` statement in the subroutine `DisplayUnexpectedError` will be highlighted. Press F8 (Step Into) once to get back to the procedure that caused the error. (This is a minor inconvenience compared to the benefit of the centralized error message.)

## Cleaning Up after an Error

Errors often occur in the middle of a lengthy procedure, when all kinds of things are happening. Many settings or values persist after an error occurs, and it's up to you to make sure they are reset back to their appropriate values. For example, these situations may be true when an unexpected error occurs in your procedure:

- Objects are open.
- The hourglass is on.
- You have set the status bar text or a progress meter.
- Warnings are off.
- You are in a transaction that should be rolled back if an error occurs.

Although your code may clean up all these settings under normal circumstances, a common mistake is to leave a mess when your code encounters an error. You don't want to leave a mess, do you?

Neglecting to clean up can cause problems, ranging in severity from annoying to serious. For example, if you don't turn the hourglass off, it will remain on while your users continue their work in Access. That's just annoying.

More seriously, if you don't turn `DoCmd.SetWarnings` back to `True`, any action queries (such as an `Update` or `Delete` query) will modify or delete data without any warning. Obviously, that can cause some serious problems that neither you nor your users will appreciate.

*Have you ever seen an Access application that won't close? Even when you click the X button, or run a* `DoCmd.Quit` *in your code, Access just minimizes instead of closing. This can be quite mysterious. Many reasons have been identified for this behavior, but one of them is related to cleaning up. Normally, Access automatically closes and releases objects when they fall out of scope, typically when your procedure ends. However, some versions of Access have issues where this normal cleanup doesn't occur. Because Access won't close if it thinks that some of its objects are still needed, it just minimizes instead. To prevent this, make sure you close the objects you open, and then set them equal to Nothing. Although later versions of Access, including Access 2010, do a better job of automatically releasing each object when its procedure ends, it is good practice to clean them up explicitly.*

To prevent these issues, make sure your code cleans everything up even if it encounters an error. Even as it is failing and exiting the procedure, its last actions can save you some trouble. Here's an example:

```
Public Function MyFunction
On Error GoTo Error_Handler
  Dim varReturnVal as Variant
  `your function code goes here
```

```
Exit_Procedure:
  Exit Function

Error_Handler:
  On Error Resume Next
  DoCmd.Hourglass False
  DoCmd.SetWarnings True
  varReturnVal = SysCmd(acSysCmdClearStatus)

  DisplayUnexpectedError Err.Number, Err.Description

  Resume Exit_Procedure
  Resume

End Function
```

Note that the first line in the `Error_Handler` section is `On Error Resume Next`. This overrides the normal error handling and forces the code to continue even if an error is encountered.

Programmers have different styles and preferences for cleaning up after an error. For example, some programmers prefer to put all the cleanup code in the `Exit_Procedure` section because they know that section will run whether the procedure ends normally or abnormally. Other programmers prefer to clean everything up as they go along in the main body of the code and then add additional cleanup code in the `Error_Handler` section. Either style is fine. The important thing to remember is that your procedure won't necessarily end normally. Look through your code to see what will happen if an error occurs, and make sure it is cleaned up.

One last point: Don't let your error handling trigger an infinite error loop. When your code is already in an error-handling situation, or if it is just trying to finish the procedure, set your error trapping to `On Error Resume Next`. That way, your code continues, ignoring any errors that occur. If you don't add that statement, you might end up in an infinite loop where an error in your error handler triggers the error handler again and again.

## More on Absorbing Expected Errors

As stated earlier in this chapter, sometimes a normal activity in your application results in Access encountering an error. For example, if the code behind a report cancels the `On Open` event, Access displays an error message. Because this is a normal event, it isn't necessary for your user to see an error message. Your application should continue as though nothing happened.

The code in the `Open` event of the report looks something like this:

```
Private Sub Report_Open(Cancel As Integer)
On Error GoTo Error_Handler

  Me.Caption = "My Application"

  DoCmd.OpenForm FormName:="frmReportSelector_MemberList", _
  WindowMode:=acDialog

  'Cancel the report if "cancel" was selected on the dialog form.

  If Forms!frmReportSelector_MemberList!txtContinue = "no" Then
    Cancel = True
    GoTo Exit_Procedure
  End If

  Me.RecordSource = ReplaceWhereClause(Me.RecordSource,
```

```
    Forms!frmReportSelector_MemberList!txtWhereClause)

Exit_Procedure:
    Exit Sub

Error_Handler:
    DisplayUnexpectedError Err.Number, Err.Description

    Resume Exit_Procedure
    Resume

End Sub
```

An open selection criteria form is shown in Figure 7-19.

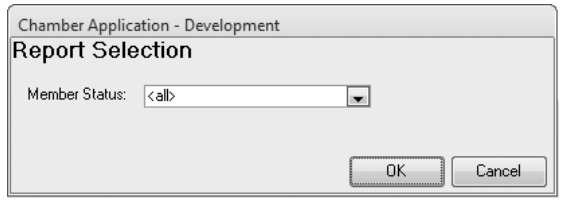


**FIGURE 7-19**

If the user clicks OK, the form is hidden and the report's `On Open` code continues. It adds the selection criteria to the report's `RecordSource` property and displays the report. However, if the user clicks Cancel, the form sets a hidden `Continue` text box to `no` before it is hidden. If the report sees a "no" in this text box, it cancels itself by setting `Cancel = True`.

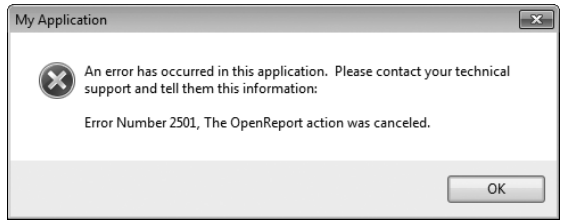


**FIGURE 7-20**

If you set the `Cancel` parameter to `True` in a report's `On Open` procedure, an error is returned to the calling code, and if it isn't handled, you see an error, as shown in Figure 7-20.

Now that is one unnecessary error message! For Access to continue without inflicting it on your poor user, you must check for this particular error (in this case, 2501) and absorb it by doing nothing but exiting the procedure. The following code shows how to absorb the error:

```
Private Sub cmdPreview_Click()
On Error GoTo Error_Handler

  If Me!lstReport.Column(3) & "" <> "" Then
    DoCmd.OpenReport ReportName:=Me!lstReport.Column(3), _
    View:=acViewPreview
  End If

  `Update the Last Run Date of the report
  DoCmd.Hourglass True
  DoCmd.SetWarnings False
  DoCmd.RunSQL _
  "UPDATE tsysReport SET tsysReport.DtLastRan = Date() " & _
  "WHERE tsysReport.RptKey = " & Me.lstReport
  DoCmd.SetWarnings True
  DoCmd.Hourglass False

Exit_Procedure:
  Exit Sub
```

```
Error_Handler:
  If Err.Number = 2501 Then
    Resume Exit_Procedure
  Else
    On Error Resume Next
    DoCmd.SetWarnings True
    DoCmd.Hourglass False
     DisplayUnexpectedError Err.Number, Err.Description
    Resume Exit_Procedure
    Resume

  End If
End Sub
```

In this code, you tell Access to ignore error 2501, should it be encountered. Access will not display an error message, and will instead exit the procedure immediately. If errors other than 2501 occur, the code will continue through to the `Else` statement and use your normal error-handling logic.

If you have several expected error codes that you want to quietly absorb, you can either add them to the `If` statement using `Or`, like this:

```
If Err.Number = 2501 Or Err.Number = 2450 Then
```

or, if you want to take a different action for each error, you can use a `Select Case` statement, like this:

```
Select Case Err.Number
Case 2501 `report was cancelled
  Resume Exit_Procedure
Case 2450 `form is no longer loaded
  Resume Next
Case Else
  ...normal error handling
End Select
```

In this example, when the report is canceled (error 2501), Access will jump directly to `Exit_Procedure`, but if it encounters a form that is not loaded (error 2450), it will use `Resume Next` to ignore the error and continue with the next line of code.

While you are becoming familiar with including error handling in every procedure, or if you aren't sure which error numbers need special handling, just include the basic error handling with the extra `Resume`. As specific expected errors pop up during your development and testing, you can add the code to quietly handle and absorb them.

## Issues in Error Handling

Some developers try to enhance their error handling with writing log files or sending e-mail. There are some issues involved with these error-handling techniques, as explained in the following sections.

### Don't Use Error Handling with Logging

Some developers write code to insert an error log record into a table or text file when an error occurs. The idea is to be able to analyze when and where errors have occurred by

querying this table long after the errors happened. However, this technique has some issues.

- Access does not provide a way to determine the name of the procedure that is currently running. Because any error logging routine needs to know which procedure caused the error, you need to manually code the name of the current procedure into each error routine. That is labor intensive and prone to errors.
- The benefit of error logging is questionable because few errors should be happening after your code has been tested and delivered. Errors should be rare enough that your users will let you know when they happen. You can always ask them to capture a screenshot if you want to see the details.
- Some types of errors cause the attempt to log them to fail. Examples include loss of network connectivity, and disconnected storage hardware. Your user may see additional unrelated errors, or you could be lulled into thinking that all errors are logged, when they may not be.

If your code is running in a managed environment, it may be beneficial to log errors to the System Event Log. For more information on this, refer to Chapter 20. The bottom line is that spending the time to log unexpected errors to a table or text file is not recommended. This is one of those cases where the benefits usually don't outweigh the costs.

### Don't Use Error Handling That Sends E-Mail

Another interesting way to track the errors that are occurring in an application is to add code to the error-handling routines that "phone home" when an error occurs. Specifically, the application builds and sends an e-mail to you (the developer) whenever an unexpected error occurs. This is usually done with the `SendObject` method, although there are other ways to utilize MAPI (mail application programming interface) directly.

This approach has the same problems listed in the preceding section, plus a few more:

- Your code needs to be able to send an e-mail using an installed e-mail client. There is always a possibility that there is no e-mail client installed, or it is not compatible with your e-mailing code.
- Some e-mail clients (for example, Microsoft Outlook) have code to protect against viruses using the e-mail program to propagate themselves to other computers. If an outside program (in this case, yours) tries to send an e-mail, a warning message displays, alerting the user that a virus may be attempting to send e-mail. That isn't what you want your user to see when your application is running.

As with error handling with logging, this technique is probably more trouble than it is worth.

## SUMMARY

The only way to really learn how to execute VBA in your Access applications is to jump in there and try it. Using the techniques explained in this chapter — how to prevent problems with asynchronous execution, how to use recordsets and recordset clones, how to debug VBA, and more — you can tackle many of the common programming tasks that your users will need.

Every procedure you write in Access VBA should have error handling. Keep error handling simple and easy to implement, with one basic copy-and-paste code block. Then do a few quick steps if necessary to adapt the error handling to your procedure:

- Change `Exit Function` to `Exit Sub` if the procedure is a sub.
- Add code to quietly absorb expected errors, if any.
- Make sure you perform any necessary cleanup if an error occurs.

By following these error-handling guidelines, you'll build VBA code that is easier to debug and maintain, so you can focus on building great features into your application.

# CONTENTS


*INTRODUCTION* *xxxiii*

**CHAPTER 1: INTRODUCTION TO MICROSOFT ACCESS 2010** **1**

**A Brief History of Access** **2**
**When to Use Access** **2**
Microsoft Office Access 2010 3
SQL Server 2008 Express Edition 3
SQL Server 2008 4
How Do You Choose? 4
**Access Database Basics** **5**
Getting Started in Access 2010 5
Access 2010 Database Templates 5
The Access Navigation Pane 6
The Access Ribbon 7
The Access Security Bar 7
**Access Database Objects** **7**
Creating Tables 7
Creating Queries 9
Creating Forms 10
Creating Reports 10
Creating Macros 11
Creating Modules 11
**Summary** **12**

**CHAPTER 2: NEW FEATURES** **13**

**New Look** **14**
**Development Environment** **15**
64-Bit 15
Office Backstage 15
Calculated Columns 16
Integration with Office Themes 18
New Macro Designer 19
Expression Builder 20
Web Service Expressions 21
Application Parts 21

**Forms** **22**
Web Browser Control 22
New Navigation Control 23
Subreports in Forms 24
**Macros** **25**
UI Macros 25
Data Macros 25
**Integration with SharePoint** **26**
Working with Data on SharePoint 27
Publish the Database to SharePoint 27
Additional SharePoint Features 27
**Browser Interface/Applications** **28**
Introducing the Web Form Designer 29
Introducing Web Report Designer 29
Feature Restrictions (Features Disabled in Web Applications) 30
**What's Gone or Deprecated** **30**
Calendar Control 30
ISAMs 30
Replication Conflict Resolver 31
Snapshot Format 31
**Summary** **31**

**CHAPTER 3: UPGRADING AND CONVERTING TO ACCESS 2010** **33**

**To Convert or To Enable** **34**
Common Terminology 34
Key Decision Factors 35
Feature Sets and File Extensions: What's New, What's Replaced, What Happens 36
Other Things to Consider 43
**Installing Multiple Versions of Access on One PC** **46**
**Changing File Formats** **48**
Selecting the Default File Format 48
Overriding the Default File Format 48
ACCDE and MDE Files 49
Steps for Converting or Enabling 49
File Conversion Using Access 2010: A One-Stop Shop 49
Other Considerations When Converting 50
Converting to Access 97 or Earlier Is a Two-Version Process 51
**Converting a Secured Database** **51**
Converting a Password-Protected Database 52
Converting a Database with Password-Protected VBA 53

**Converting a Replicated Database** **53**
**Enabling a Database** **56**
Enabling the Experience: Opening 95 or 97 Files with Access 2010 56
**Access 2010: 64-Bit Considerations** **57**
Porting an Access application to a 64-bit Platform 58
**Summary** **61**

**CHAPTER 4: MACROS IN ACCESS 2010** **63**

**VBA versus Macros in Access** **63**
Benefits of Using VBA 64
Benefits of Using Macros 65
**Types of Macros** **66**
Macro Objects 66
Embedded Macros 66
Data Macros 67
**Creating Macros in Access 2010** **67**
New Macro Designer 69
Additional Macro Changes 75
Sharing Macros Using Access 2010 76
Running Macros 76
Debugging Macros 77
**Macro Objects and Embedded Macros** **78**
Error Handling 78
Variables 79
Macro Actions and Arguments 82
Macro Scenarios 87
**Data Macros** **93**
Types of Data Macros 94
Running Data Macros 95
Data Macro Blocks 95
Data Macro Properties 97
Data Macro Actions and Arguments 98
Error Handling 99
Variables 101
Data Macro Scenarios 101
**Summary** **117**

**CHAPTER 5: USING THE VBA EDITOR** **119**

**Anatomy of the VBA Editor** **119**
**Using the Object Browser** **121**

Object Browser Components 122
Show Hidden Members 124
**Testing and Debugging VBA Code 124**
Immediate Window 125
The Debug.Print Statement 127
The Debug.Assert Statement 128
Breakpoints 128
Stepping through Code 130
Call Stack 132
Run to Cursor 134
Locals Window 135
Watch Window 135
Edit and Continue 137
**Using Option Statements 137**
**Summary 139**

**CHAPTER 6: VBA BASICS 141**

**The Mindset of a Programmer 142**
**Anatomy of VBA Procedures 142**
**VBA Keywords 145**
**VBA Operators 146**
**Variables and VBA Syntax 147**
Variables 147
Naming Your Variables 149
Variable Scope and Lifetime 151
Overlapping Variables 153
**Other VBA Components 156**
Option Statements 156
Comments 156
Line Continuation 157
Constants 159
Enums 160
**VBA Objects 161**
Properties 162
Methods 162
Events 162
**Using Code Behind Forms and Reports 163**
**Using VBA Code to Call Macros 165**
**Writing Code in Modules 166**
**Example: User-Defined Function 169**
**Summary 171**

**CHAPTER 7: USING VBA IN ACCESS** 173

**When Events Fire** 174
Common Form Events 174
Common Control Events 176
Common Report Events 176
Asynchronous Execution 177
**VBA Procedures** 179
Function or Sub? 179
Public or Private? 180
Coupling and Cohesion 181
Error Handling 183
Using Variables 183
**Evaluating Expressions in VBA** 185
If ... Then 185
Checking for Nulls 186
Select Case 188
**Using Recordsets** 188
Opening Recordsets 188
Looping through Recordsets 189
Adding Records 190
Finding Records 191
Updating Records 191
**Using Multiple Recordsets** 192
Copying Trees of Parent and Child Records 192
Using Bookmark and RecordsetClone 194
Cleaning Up 195
**Using VBA in Forms and Reports** 196
All about Me 196
Referring to Controls 197
Referring to Subforms and Subreports 197
Closing Forms 198
**Debugging VBA** 199
**Investigating Variables** 200
When Hovering Isn't Enough — Using the Immediate Window 202
Setting Breakpoints 202
Setting Watch Values 202
Stopping Runaway Code 204
Stepping through Your Code 204
**Common VBA Techniques** 206

Drilling Down with Double-Click 206
Date Handling 207
Handling Rounding Issues 209
**String Concatenation Techniques 211**
The Difference Between & and + 211
String Concatenation Example 212
**VBA Error Handling 213**
Why Use Error Handling? 213
Two Kinds of Errors: Unexpected and Expected 214
Basic Error Handling 215
Cleaning Up after an Error 221
More on Absorbing Expected Errors 222
Issues in Error Handling 224
**Summary 225**

**CHAPTER 8: CREATING CLASSES IN VBA 227**

**Why Use Classes? 228**
**A Touch of Class 228**
**Creating a Class Module 230**
Adding a Class Module to the Project 230
A Brief Word on Naming the Class 230
Instantiating Class Objects 231
Creating Class Methods 231
Creating Property Procedures 234
**Naming and Identifying Objects 241**
What Does the Object Do? 242
Naming Techniques 242
Identifying a Class Instance 243
**Using Class Events 243**
Initialize and Terminate Events 243
Creating Custom Class Events 244
Responding to Events 245
Handling Errors in Classes 248
**Forms and Reports as Objects 251**
**Variable Scope and Lifetime 257**
**The Me Property 260**
Subclassing the Form 260
Creating the Subclassed Form 261
Creating a Parent Property 261
**Creating a Clone Method 262**

**Creating and Using Collection Classes** **264**
The Collection Object 264
Collection Class Basics 266
**The Three Pillars** **275**
Encapsulation 275
Inheritance 276
Polymorphism 276
Inheriting Interfaces 277
Instancing 280
**Summary** **281**

**CHAPTER 9: EXTENDING VBA WITH APIS** **283**

**Introducing the Windows API** **284**
Finding API Functions 284
Why You Need the API 285
**Introducing Linking** **287**
Static Linking 287
Dynamic Linking 287
**Declaring APIs** **288**
The Declare Keyword 288
The PtrSafe Keyword 289
Naming the Procedure 289
Specifying the Lib(rary) and Argument List 290
**Understanding C Parameters** **293**
Signed and Unsigned Integers 293
Numeric Parameters 294
Object Parameters 297
String Parameters 298
Variant Parameters 298
Pointers to Numeric Values 298
Pointers to C Structures 299
Pointers to Arrays 299
Pointers to Functions 300
Pointers in 64-Bit Windows 301
The Any Data Type 304
**Err.LastDLLError** **304**
**Distributing Applications That Reference Type Libraries and Custom DLLs** **306**
**Useful API Functions** **306**
Returning the Path to the Windows Folder 306

Determining Whether the System Processor Is 32-Bit or 64-Bit 307
Determining Whether Windows Is 32-Bit or 64-Bit 308
Determining Whether Office Is 32-Bit or 64-Bit 308
Displaying the Windows Open Dialog Box 309
Finding the Position of a Form 311
Finding the Temp Directory 312
Generating a Unique Temp Filename 312
Finding the Login Name of the Current User 313
Finding the Computer Name 314
Opening or Printing Any File 314
Delaying Code Execution 315
Getting the Path to a Special Folder 315
Locking the Computer 317
**Summary 317**

**CHAPTER 10: WORKING WITH THE WINDOWS REGISTRY 319**

**About the Registry 320**
What the Registry Does 320
What the Registry Controls 321
Accessing the Registry 321
Registry Organization 322
Registry Organization on 64-Bit Windows 328
**Using the Built-In VBA Registry Functions 329**
SaveSetting 330
GetSetting 330
GetAllSettings 331
DeleteSetting 332
Typical Uses for the Built-In VBA Registry Functions 332
**Using the Windows Registry APIs 335**
Getting Started 335
Creating a Registry Key 341
Setting the Value for a Key 342
Getting the Value for a Key 343
Deleting a Registry Value 344
Deleting a Registry Key 345
Testing the Function Wrappers 346
Opening an Existing Registry Key 348
Connecting to the Registry on a Remote Computer 349
Enumerating Registry Keys and Values 350
**Summary 353**

**CHAPTER 11: USING DAO TO ACCESS DATA 355**

**Data Access Objects 355**
**Why Use DAO? 356**
**New Features in DAO 357**
Multi-Value Lookup Fields 357
Attachment Fields 358
Append-Only Fields 359
Database Encryption 359
Calculated Fields 359
**Referring to DAO Objects 359**
**The DBEngine Object 361**
The Workspaces Collection 361
The Errors Collection 365
**The Database Object 366**
The Default (Access) Database 366
The CurrentDb Function 367
Opening an External Database 368
**DAO Object Properties 370**
DAO Property Types 370
Creating, Setting, and Retrieving Properties 370
**Creating Schema Objects with DAO 374**
Creating Tables and Fields 375
Creating Indexes 378
Creating Relations 380
Creating Multi-Value Lookup Fields 382
Creating Calculated Fields 385
**Data Access with DAO 386**
Working with QueryDefs 386
Working with Recordsets 389
Filtering and Ordering Recordsets 391
Navigating Recordsets 394
Bookmarks and Recordset Clones 399
Finding Records 402
Working with Recordsets 405
Working with Attachment Fields 409
**Append-Only Fields 414**
**Summary 417**

**CHAPTER 12: USING ADO TO ACCESS DATA** 419

**Introduction to ADO in Access** 420
**Adding ADO References** 420
Referring to ADO Objects 420
**Connecting to ADO Data Sources** 421
Creating an Implicit Connection 421
Creating an ADO Connection Object 422
Creating a Connection String 423
Creating a Data Link Connection 424
Closing a Connection 425
Working with Cursors 426
Using Transactions 427
**Data Access with ADO** 429
Overview of the ADO Object Model 429
Using the Execute Method 430
Creating Recordsets 434
Navigating Recordsets 437
Working with Data in Recordsets 441
**Using ADO Events** 448
Declaring WithEvents 448
Implementing ADO Event Methods 449
Implicitly Triggering Events 449
Explicitly Calling Events 450
Testing the State Property 451
**Schema Recordsets with ADO** 451
ADO Schema Recordsets 451
Specifying Constraint Columns 452
Using ACE Specific Schemas 452
**Creating Schema with ADOX** 453
Adding References to ADOX 453
The ADOX Object Model 453
Working with Tables 454
Working with Views (Queries) 455
Managing Security with ADOX 457
**Summary** 457

**CHAPTER 13: USING SQL WITH VBA** 459

**Working with SQL Strings in VBA** 460
Building SQL Strings with Quotes 460
Using Single Quotation Marks instead of Double Quotation Marks 461

Concatenating Long SQL Strings 462
**Using SQL When Opening Forms and Reports 463**
**Using SQL to Enhance Forms 464**
Sorting on Columns 464
Selections on Index Forms 466
Cascading Combo Boxes 473
Using SQL for Report Selection Criteria 475
Altering the SQL inside Queries 480
**The ReplaceOrderByClause and ReplaceWhereClause Functions 481**
**Summary 488**

**CHAPTER 14: USING VBA TO ENHANCE FORMS 489**

**VBA Basics 490**
Properties 490
Event Properties: Where Does the Code Go? 491
Naming Conventions 493
**Creating Forms the 2010 Way 494**
Columnar and Tabular Layouts 495
Anchoring 496
The Modal Dialog Box Mode 499
Control Wizards — Creating Command Buttons Using VBA or Macros 499
Command Button Properties 500
Attachment Controls 501
Combo Boxes 505
Using the BeforeUpdate Event 516
Saving E-mail Addresses Using the Textbox AfterUpdate Event 519
Outputting to PDF 522
OpenArgs 523
IsLoaded() 524
On Timer () 525
Late Binding 527
On Click(): Open a Form Based on a Value on the Current Form 531
Multiple Form Instances 534
Displaying Data in TreeView and ListView Controls 540
**Summary 548**

**CHAPTER 15: ENHANCING REPORTS WITH VBA 549**

**Introduction to Reports 549**
How Reports Are Structured 550

New in Access 2007 551
New in Access 2010 551
**Creating a Report 554**
**Working with VBA in Reports 555**
Control Naming Issues 555
The Me Object 556
**Important Report Events and Properties 556**
Opening a Report 556
Section Events 558
Closing a Report 562
**Report Properties 562**
Section Properties 563
Control Properties 564
**Working with Charts 565**
**Common Report Requests 565**
Changing the RecordSource at Runtime 565
Gathering Information from a Form 566
Changing the Printer 567
Dictionary-Style Headings 568
Shading Alternate Rows 570
Conditional Formatting of a Control 572
Creating a Progress Meter Report 573
**Layout View 575**
**Report View 576**
Considerations When Designing for Report View 576
Interactivity 577
**Summary 579**

**CHAPTER 16: CUSTOMIZING THE RIBBON 581**

**Ribbon Overview 582**
**Custom Menu Bars and Toolbars 582**
Custom Menu Bars 582
Shortcut Menu Bars 583
**Ribbon Customization Using the Options Dialog Box 583**
**Ribbon Customization 584**
**Saving a Custom Ribbon 584**
**Specifying the Custom Ribbon 585**
Defining a Ribbon Using XML 586
Writing Callback Routines and Macros 597
More Callback Routines 599
Displaying Images 604

Refreshing Ribbon Content 607
**Creating an Integrated Ribbon 609**
Building the Report Manager 609
Building the Custom Filter Interface 613
**Creating a Ribbon from Scratch 616**
Defining the Tabs and Groups 616
Building the Home Tab 619
Building the Settings Tab 625
Building the Administration Tab 626
**More Ribbon Tips 628**
Setting Focus to a Tab 628
Additional Resources 629
**Summary 630**

**CHAPTER 17: CUSTOMIZING THE OFFICE BACKSTAGE 631**

**Introducing the Office Backstage 632**
Access 2010 Backstage 632
Parts of the Backstage 633
Uses for the Backstage in Custom Applications 633
**Writing a Backstage Customization 634**
**Controls in the Backstage 634**
Tab 634
Group 635
TaskGroup, Category, and Task 637
TaskFormGroup 638
Button 640
GroupBox 641
Hyperlink 641
ImageControl 641
LayoutContainer 642
RadioGroup 643
**Designing the Layout of Components 643**
Single-Column Layout 644
Two-Column Layout 644
Column Widths 644
Creating a Grid 645
**Extending the Existing Backstage 649**
Adding a Group to an Existing Tab 649
Adding a Category to an Existing TabFormGroup 650
Adding a Task to an Existing TaskGroup 651
**Backstage-Specific Callbacks 653**

onShow 653
onHide 653
getStyle 654
getHelperText 654
getTitle 654
getTarget 655
**Backstage Scenarios 655**
Access Runtime Experience 655
Setting the Title of a Tab to the Application Title 656
About Page and Contact Form 656
Warning for a Missing Reference 660
Custom Database Information 662
Creating a Bulleted or Numbered List 664
Welcome Page with Image 665
Other Possible Examples 668
**Summary 668**

**CHAPTER 18: WORKING WITH OFFICE 2010 671**

**Working with Outlook 2010 672**
Setting References to Outlook 672
Creating Outlook Application Objects 672
Working with MailItem objects 674
Outlook Security Features 678
Other Outlook Objects 679
More Information about Outlook 684
**Working with Excel 2010 684**
Setting References to Excel 684
Creating Excel Application Objects 685
Working with Excel Workbooks 686
Working with Sheets in Excel 689
Working with Data in Excel 691
More Information about Excel 695
**Working with Word 2010 695**
Setting References to Word 695
Creating Word Application Objects 696
Working with Document Objects 698
Working with Data in Word 701
More Information about Word 705
**Summary 705**

**CHAPTER 19: WORKING WITH SHAREPOINT 707**

**SharePoint 2010 707**
What Is SharePoint? 708
SharePoint 2010 Requirements 708
SharePoint 2010 Versions 708
Access Services on SharePoint Server 709
**Access Features Overview 709**
SharePoint Features in Access 2010 709
Access Features in SharePoint 2010 710
**SharePoint Features in Access 711**
Access Web Applications 711
Linked Tables to SharePoint 724
Migrating a Database to SharePoint 731
Publishing a Database to SharePoint 736
**Access Features on SharePoint 740**
SharePoint 2.0 Shows Access Features 740
Access Web Datasheet 741
Open with Access 742
Importing from SharePoint 748
Access Views on SharePoint 752
**Summary 756**

**CHAPTER 20: WORKING WITH .NET 757**

**Overview 758**
Example Files 758
**Visual Studio .NET 2010 758**
Getting Visual Studio 2010 759
Installing Visual Studio 2010 760
.NET Terminology 760
Writing Code in Visual Studio 2010 762
Debugging Code in Visual Studio 2010 769
The MSDN Library 770
**Using Access Databases in .NET 771**
Working with ADO.NET 771
Types of .NET Applications 776
Building Client Applications 776
Building Web Applications 779
Other Methods of Using Access Databases 781

**Automating Access with .NET** **782**
The Access PIA 782
Setting References 782
Creating Code to Automate Access 783
Running the Automated Application 785
**Creating COM Add-Ins for Access** **785**
The Benefits of COM Add-Ins 786
Creating a New COM Add-In Project 786
Setting References to the Access PIA 787
Adding Custom Code to the Add-In 788
Installing the COM Add-In 791
Running the COM Add-In 791
**Using .NET Code in Access** **791**
Creating a Managed Library in .NET 791
Calling a Managed Library from VBA 796
**Summary** **797**

**CHAPTER 21: BUILDING CLIENT-SERVER APPLICATIONS WITH ACCESS** **799**

**Database Application Evolution** **800**
**Client-Server Applications** **800**
Using the Sample Files 801
Installing the Sample Database 802
Choosing the Correct File Format 803
**The ACCDB/MDB File Format** **803**
Linking to External Data 804
Creating a DSN via Access 805
DSN Connection Types 807
Using ACE with ODBC Data Sources 807
Increasing ODBC Performance 810
**The ADP File Format** **813**
Using ADPs to Link to Data 814
Query Options on SQL Server 816
**ACCDB/MDB versus ADP** **817**
Recordset Differences 818
Security Differences 818
Local Data Storage 819
Sharing Application Files 819
**Controlling the Logon Process** **820**
Controlling Login for ACCDB/MDB Files 820
Controlling the Login Process for ADPs 823

**Binding ADO Recordsets** **826**
Binding to a Form, ComboBox, or ListBox 826
Binding to a Report 827
Using Persisted Recordsets 830
**Working with Unbound Forms** **831**
When to Use Unbound Forms 832
Creating Unbound Forms 832
**Summary** **840**

**CHAPTER 22: THE ACCESS 2010 TEMPLATES** **841**

**Access 2010 Template Features** **841**
**Access 2010 Templates Types** **842**
Standalone Database Templates 842
Access Web Application Templates 843
Templates for SharePoint Applications 844
**Application Parts** **844**
**Table Field Templates** **844**
**Save As Template** **845**
Creating ACCDT Files 845
Features Not Supported in ACCDT Files 846
Deploying ACCDT Files 847
**The ACCDT File Format** **849**
The ACCDT File Parts 851
The Template's Root Folder 851
The Template Folder 852
The Database Folder 854
The Objects Folder 857
**Summary** **860**

**CHAPTER 23: ACCESS RUNTIME DEPLOYMENT** **861**

**The Access 2010 Runtime** **861**
Why Use the Access Runtime? 862
Access 2010 Runtime Versions 862
Getting the Access 2010 Runtime 862
Using the Access Runtime 863
**Deploying the Access Runtime** **865**
Manual Installation of the Runtime 865
The Package Solution Wizard 866
Using the Package Solution Wizard 866
**Summary** **874**

**CHAPTER 24: DATABASE SECURITY 875**

**ACCDB File Security 876**
Shared-Level Security for ACCDBs 877
Securing VBA Code in ACCDB 883
ACCDB Security Summary 887
**MDB File Security 888**
Shared-Level Security for MDBs 889
Encoding an MDB File 891
Securing VBA Code for MDBs 891
User-Level Security 893
Working with User-Level Security 896
Using the User-Level Security Wizard 896
Using the Access User Interface 896
User-Level Security Using DAO 897
User-Level Security Using ADO 906
User-Level Security Using ADOX 916
**Summary 917**

**CHAPTER 25: ACCESS 2010 SECURITY FEATURES 919**

**The Office Trust Center 920**
What Is the Trust Center? 920
Trust Center Features 920
**Disabled Mode 926**
Why Do We Have Disabled Mode? 926
Enabling a Database 927
Modal Prompts 928
AutomationSecurity 930
Macros in Access 2010 931
**Digital Signatures and Certificates 934**
Types of Digital Certificates 935
Using Self-Certification 937
Signed Packages 939
**Access Database Engine Expression Service 941**
Sandbox Mode in Access 2010 942
Sandbox Mode Limitations 942
Workarounds 943
**Summary 944**

APPENDIX A: THE ACCESS OBJECT MODEL 945

APPENDIX B: DAO OBJECT METHOD AND PROPERTY DESCRIPTIONS 999

APPENDIX C: ADO OBJECT MODEL REFERENCE 1035

APPENDIX D: 64-BIT ACCESS 1095

APPENDIX E: REFERENCES FOR PROJECTS 1103

APPENDIX F: RESERVED WORDS AND SPECIAL CHARACTERS 1113

APPENDIX G: NAMING CONVENTIONS 1127

APPENDIX H: THE ACCESS SOURCE CODE CONTROL 1137

APPENDIX I: TIPS AND TRICKS 1145

*INDEX* *1191*