Andrew Couch
Microsoft® Most Valuable Professional (MVP)—Microsoft Access

Microsoft

Microsoft® Access® 2010 VBA Programming

INSIDE OUT

- **The ultimate, in-depth reference**
- **Hundreds of timesaving solutions**
- **Supremely organized, packed with expert advice**

Includes
YOUR BOOK + SAMPLE FILES—ONLINE!
See back

INSIDE OUT

Conquer Microsoft® Access® VBA Programming—from the inside out!

Intermediate/Advanced

You're *beyond* the basics, so dive right in and customize, automate, and extend Access—using Visual Basic® for Applications (VBA). This supremely organized reference is packed with hundreds of timesaving solutions, troubleshooting tips, and workarounds. It's all muscle and no fluff. Discover how the experts use VBA to exploit the power of Access—and challenge yourself to new levels of mastery!

- Enhance your application with VBA built-in functions and SQL code
- Use the Access Object Model to work with data in forms and reports
- Manipulate data using SQL, queries, and recordsets with Data Access Objects (DAO)
- Create classes for handling form and control events
- Connect your Access database to different sources of data
- Effectively plan how to upsize an existing Access database to Microsoft SQL Server®
- Dynamically update Microsoft Excel® spreadsheets from your database
- Migrate your Access database directly to the cloud using SQL Azure™

Your book + sample files—online!

- **Fully searchable online edition of the book**—with unlimited access on the web. *See inside back; free online account required.*
- **Download sample database files**—ready to put to work, with examples for Access 2003 and later. *See http://go.microsoft.com/FWLink/?Linkid=223727*

About the Author

Andrew Couch, Microsoft MVP for Access, has been programming with VBA since it was introduced in Access and Microsoft Office. He uses VBA on a daily basis in commercial applications. An experienced instructor, Andrew has also taught VBA programming courses.

microsoft.com/mspress

ISBN: 978-0-7356-5987-2

U.S.A. $49.99
Canada $57.99
[*Recommended*]

Microsoft Office/ Microsoft Access

Microsoft® Office

Microsoft®

Microsoft®

Microsoft® Access® 2010 VBA Programming Inside Out

Andrew Couch

ISBN: 978-0-7356-5987-2

2 3 4 5 6 7 8 9 10 LSI 7 6 5 4 3 2

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at *mspinput@microsoft.com*. Please tell us what you think of this book at *http://www.microsoft.com/learning/booksurvey*.

Acquisitions and Developmental Editor: Kenyon Brown
Production Editor: Teresa Elsey
Editorial Production: Octal Publishing, Inc.
Technical Reviewer: Alan Cossey
Indexer: Denise Getz
Cover Design: Twist Creative • Seattle
Cover Composition: Karen Montgomery
Illustrator: Robert Romano

[2012-10-12]

pour Pamela, ma raison d'être

Contents at a Glance

Table of Contents

Part 3: Working with Forms and Reports

Introduction

Microsoft Visual Basic for Applications (VBA) is an exceptional programming language and environment. The language has grown out of a need to have a programming language that would allow more business-focused individuals to write programs, but equally support the programming features that developers look for in a product. The environment is as important as the language because of its unique features, allowing code to be quickly modified while being debugged.

The Access Basic language in early product versions evolved into the VBA language, which provided a cross-product language for the Microsoft Office products. This all coincided with the revolution of an event-driven approach to programming, which was very important, because the emphasis on being a programmer shifted from writing thousands of lines of code to writing snippets of code in response to events. This also led to a change of emphasis from writing large libraries of code to understanding how to manipulate the object models in the environment—a focus which has progressed with .NET, albeit using namespaces instead of object models.

Even with the introduction of object-oriented programming, VBA has kept pace with the expectations of modern programming. The two products that have shaped VBA the most are Microsoft Excel and Microsoft Access; Excel introduced VBA and originally gained VBA programming features in advance of these becoming available within Access.

A significant strength of VBA is that it is universal to the Microsoft Office suite of programs; all the techniques we describe in this book can be applied to varying degrees within the other Office products. A major turning point for these products was the ability through OLE Automation to be able to drive one product from another, and to cut and paste code between the different environments with a minimum amount of change to the code. This was a revolutionary feature introduced with the programming language of Access Basic, conforming to the new VBA standard established in Excel. VBA suddenly provided the long-awaited platform for the simple integration of the Office products and building solutions that could easily exploit the strengths of each component product in the Office suite. The combination of Access and VBA offers an extremely productive environment within which to construct applications.

VBA has often been criticized for its simplicity as a language when compared to languages such as C++ and C#. Quite to the contrary, the big advantage of VBA is that this simplicity leads to more easily maintainable and reliable code, particularly when developed by people with a more business-focused orientation to programming. Looking toward the future, the emphasis in modern programming has moved from the language syntax to the intricacies of understanding the objects that the language manipulates, so the emphasis on the specific syntax of languages is starting to blur.

In the .NET world, the conflict between using VB.NET, which originates from VBA, and C# continues, because even though the objects being manipulated are now common, there are subtle differences between the languages, which means that developers moving from VBA to C# can often feel that they are being led out of their comfort zone, especially when they need to continue to use VBA for other applications.

Access has often been criticized for creating poor performance applications where a prototype turns into a business critical system, propagating a support nightmare for information technology departments, and leading to applications that eat up network bandwidth. It has also been stated that the product is never used for mission-critical applications. The truth is that both Access and Excel are pivotal to many organizations, but the people answering that mission-critical question are often not willing to admit to this because it is perceived as vulnerability. The problem with using Access and Excel is that Rapid Application Development (RAD) can often come to mean final application without recourse to a more structured oversight of what is being developed, and as data volumes and user communities grow, so too the inevitable flaws in not having designed a scalable solution are exposed.

This book details how Access and VBA are not a problem, although their success is often their downfall in the hands of those lacking some direction on how to effectively develop applications. The big problem with Access is that the underlying database engine is extremely efficient and can compensate for a design that normally would not scale. So if you convert your Access database data to be located in Microsoft SQL Server, Microsoft SQL Azure, or Microsoft SharePoint, you might find that the existing application design techniques for searching and displaying data need to be revised. Our advice is to take into account the mantra of Client-Server design, which is to minimize the amount of data being transferred in any operation.

In this book, we would like to make our contribution toward creating a better informed community of developers, and show how to better develop applications with VBA.

Who This Book Is For

This book is aimed at two types of reader. First, we want to enable the reader who has worked with Access and developed applications to move to the next level of development. We want to help that reader to more fully develop applications with a deeper understanding of what it means to program with VBA.

Our second target audience is the more experienced VBA programmer, who needs the assistance of a good instructional text to move up a gear and explore the more advanced aspects of VBA programming. As well, we have devoted a significant number of our pages to supporting you in developing with both SQL Server and cloud computing.

Assumptions About You

We make a basic assumption in this book that you are experienced either in working with Access or that you have a strong programming background, which means that you can learn VBA programming in Access very quickly. We will spend no time explaining how to create a table, form, or report, and if you cannot do this, you need to first learn these actions in more detail. We recommend our companion text *Microsoft® Access® 2010 Inside Out* by Jeff Conrad and John Viescas.

If you have some VBA Programming experience, you can skim over Chapters 1–3. If your experience level is not such that you are comfortable skipping chapters, Chapters 1–3 will, we hope, give you a key appreciation of the power of the VBA development environment.

How This Book Is Organized

This book allows you to either start at the beginning and work through each chapter or to dip into specific chapters or topics to investigate a particular feature of VBA. To enable dipping into the book, each part is designed to be self-contained.

Part 1, "VBA Environment and Language"

In Chapters 1, 2, and 3, we provide a foundation that demonstrates how to program with VBA. We start by showing you how to debug, write, and modify code (gaining confidence with the VBA environment is the first step to efficiently developing applications within it). Then we move on to an in-depth exposition of the VBA language, which can act both as a reference for coding syntax and a solid introduction to the language.

Part 2, "Access Object Model and Data Access Objects (DAO)"

Chapters 4 and 5 dig deep into programming with the objects that make up Access, including the DAO programming language, which is the bread and butter programming technique for any Access VBA developer.

Part 3, "Working with Forms and Reports"

Chapters 6, 7, and 8 illustrate how to apply VBA when working with forms, controls, and reports. This develops your core techniques in understanding how to apply VBA for building the key interface components in applications.

Part 4, "Advanced Programming with VBA Classes"

Chapters 9, 10, and 11 are for some developers more esoteric than the rest of this book, but they illustrate how you can exploit VBA to embrace the most advanced concepts of modern

computing by using object-oriented programming. There are a lot of cunning tricks and techniques in these chapters that are worth reading about, and many of the ideas in these chapters will take you forward in also handling development with .NET.

Part 5, "External Data and Office Integration"

In Chapters 12 and 13, we address the issue of how to link Access to external data and write VBA to communicate both with other Office applications and external data sources such as SQL Server and SharePoint.

Part 6, "SQL Server and SQL Azure"

Chapters 14, 15, and 16 provide a comprehensive description of how to extend the reach of Access applications by moving the back-end data into SQL Server, and then onto SQL Azure. Chapter 14 is dedicated to equipping developers with a solid understanding of how to develop code with SQL Server, during which we explain both how to use the SQL Server Management Studio and write programs using Transact SQL (T-SQL).

Chapter 15 moves on to look at converting Access Databases to SQL Server by using both the Upsizing Wizard and the SQL Server Migration Assistant (SSMA). Chapter 16 discusses how to move your databases into the cloud either by using the SQL Server Import and Export Wizard feature in the SQL Server Management Studio from a local SQL Server, or SSMA from an Access Database. We discuss how you can exploit the unique features of Office in directly constructing links to Azure, building multi-tenanted solutions and using the soon to be released new Data Sync features in SQL Azure.

Part 7, "Application Design"

The last part of this book, Chapters 17 and 18, shows you a number of ideas for helping you to create applications, including a discussion of how to design the user interface, building ribbons, utilizing the Windows API, and working with ADO and ADOX. In Chapter 17, we will step through the process of building applications. This chapter ties together all the lessons you learn throughout the book, making references back to other sections.

Features and Conventions Used in This Book

This book uses special text and design conventions to make it easier for you to find the information you need.

Text Conventions

Convention	Meaning
Boldface type	This indicates user input that you are instructed to type; for example, "Click the Save As command, name the file **NewFile_01**, and then click OK."
Ctrl+F	Keystroke combinations are presented as Ctrl+G, which means to hold down the Ctrl key and press the letter G on the keyboard, at the same time.
Object names	When we need to draw your attention to a specific technical term, program elements, or an object in the sample database, it will be presented in italic; for example, "Open the form *frmSample* and right-click the *ListBox* control."

Design Conventions

INSIDE OUT

This statement illustrates an example of an "Inside Out" heading

These are the book's signature tips. In these tips, you get the straight scoop on what's going on with the software—inside information about why a feature works the way it does. You'll also find handy workarounds to deal with software problems.

Note

Notes offer additional information related to the task being discussed.

About the Companion Content

You'll see references to the sample files and bonus content throughout the book. A complete list of the key database files follows (we have not listed all the smaller support files for each chapter).

We have also included in the bonus content (which is located within the file sets for Chapters 5, 7, and 18) additional application files that contain more code examples and provide useful utilities to add to your program libraries.

To access and download the companion content, visit: *http://www.microsoftpressstore.com/title/9780735659872*.

Chapter or topic	Content
Chapter 1	• VBAEnvironment.accdb
Chapter 2	• VBAExamples.accdb
Chapter 3	• VBAFeaturesExamples.accdb
Chapter 4	• AccessObjectModel.accdb
Chapter 5	• DAOExamples.accdb • CountryLibrary.accdb • Find_IT.accdb • DocDAO.accdb
Chapter 6	• FormExamples.accdb
Chapter 7	• Controls.accdb • TreeBuilder.accdb
Chapter 8	• Reports.accdb
Chapter 9	• BuildingClasses.accdb • BuildingClassesAfterExportImport.accdb
Chapter 10	• ClassesAndEvents.accdb
Chapter 11	• ClassesAndForms.accdb
Chapter 12	• Employees_be.accdb • Sample_fe.accdb • WebDatabase.accdb

Chapter or topic	Content
Chapter 13	• ExcelAnalysis.accdb • OfficeApplications.accdb • OutlookContacts.accdb • WordQuote.accdb
Chapter 14	• SQLServerExamples.accdb • SQL Server Script files
Chapter 15	• Northwind_ProblemsAndFixes.accdb • SQLServerCodeExamples.accdb • SQL Server Script files
Chapter 16	• Northwind_ForAzure.accdb • SQLAzureCodeExamples.accdb • SQL Azure Script files
Chapter 17	• ApplicationDevelopment.accdb • ApplicationDevelopment64Bit.accdb • ApplicationDevelopment_2007.accdb
Chapter 18	• ADOExamples.accdb • DocADOX.accdb • SQL Server Script files
Bonus Content	• Chapter 5: Find_IT.accdb, DocDAO.accdb • Chapter 7: TreeBuilder.accdb • Chapter 18: DocADOX.accdb

Your Companion eBook

The eBook edition of this book allows you to:

- Search the full text
- Print
- Copy and paste

To download your eBook, please see the instruction page at the back of this book.

Access Versions

All of the examples in the book are designed to run with Access 2010 32-bit.

If you are using Access 2010 64-bit, you should also be able to use the examples with the following revisions: in Chapter 17, use ApplicationDevelopment64Bit.accdb. The Bonus material databases have versions called Find_IT64Bit.accdb, DocADOX64Bit.accdb, and DocDAO64bit.accdb. The file TreeView.accdb has no equivalent 64-bit version, as this control is not supported in the 64-bit environment.

The majority of the code examples in this book will work on older versions of Access, and we have provided a set of .mdb files for this in Access 2002–2003 file format. However, the older the version that you use, the less likely will be the compatibility. There are several topics in Chapters 4, 5, 13, and 17 which were either not present in earlier versions of Access or have undergone a significant amount of change.

In some chapters, we have inevitably had to construct examples that rely on a hardwired path; in these situations you might find it easier either to construct your own example, as described in a chapter, or move the files to a path that matches the completed example. Where possible, we have provided assistance and advice in the sample databases to overcome any path problems.

Acknowledgments

A good technical book needs an author who is well informed and passionate, and I hope I can live up to that expectation. But it also needs contributions from a team of people to turn the idea into a reality.

First, my thanks to Kenyon Brown at O'Reilly Media; without his asking me to propose to write this book, it would have never have been started. Your diligence throughout the entire process has been splendid.

Next, I offer immense gratitude to Alan Cossey, who acted as technical reviewer on this book; having acted as a technical reviewer myself, I can greatly appreciate all of his time and recommendations made during the review process.

I would also like to thank Bob Russell at Octal Publishing, Inc., for acting as my copy editor; Bob has not only ensured that the flow of the book has a professional polish, but also caused me to reflect on the meaning of many parts of the text.

I would like to thank my good friend Jeff Conrad at Microsoft. Jeff is a great advocate for Access and helped wonderfully in answering and passing along many of my comments and questions to the Microsoft teams.

Numerous thanks also to those members of UK Access User Group for helping in testing my solutions to difficult technical issues. You can't beat a good community of developers!

My thanks also to Dianne Russell at Octal Publishing, Inc., for managing the copy editing and composition, and Betsy Waliszewski, senior marketing manager, for promotional activities.

Finally, I would like to thank my wife, Pamela, for her patience, and my son, Michael, for his assistance at various stages in helping with chapter layouts.

Andrew Couch
July 2011

Support and Feedback

The following sections provide information on errata, book support, feedback, and contact information.

Errata & Support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site:

http://www.microsoftpressstore.com/title/ 9780735659872

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in Touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*.

PART 1

VBA Environment and Language

CHAPTER 1

Using the VBA Editor and Debugging Code

The Microsoft Visual Basic for Applications (VBA) Editor is more than a simple editing tool for writing program code. It is an environment in which you can test, debug, and develop your programs. Understanding the unique way in which the editor allows you to make modifications to application code while the execution of the code is paused will help you to learn how to quickly develop your applications and master the techniques for debugging code.

In addition to changing code on-the-fly as it executes, you can switch across to the Microsoft Access 2010 application window while your code is paused, create a query, run the query, copy the SQL to the clipboard, and then swap back to the programming environment to paste the SQL into your code. It is this impressive flexibility during the development cycle that makes developing applications with VBA a productive and exhilarating experience.

In this chapter, you will work with examples of program code written in the VBA language. The VBA language itself is systematically explained in Chapter 2, "Understanding the VBA Language Structure," and in Chapter 3, "Understanding the VBA language Features." So, before reading this chapter (or while you're reading it) you might want to either skim read those chapters or simply refer to specific topics as they arise in this chapter. We have also included some examples of Data Access Object (DAO) programming code. In this chapter, we will be providing only limited explanations of the DAO development environment, just to place it into the context of building real applications. For more detailed information about it, see Chapter 5, "Understanding the Data Access Object Model."

To successfully work with VBA, you need an understanding of the language, the programming environment, and the objects that are manipulated by the code. Getting started means dipping into different topics as you begin to build sufficient knowledge to effectively use VBA.

By the end of this chapter, you will understand:

- The different ways that you can run and debug sections of program code.
- How to modify program code while it is paused and then resume execution.
- How to work with the different windows in the programming environment.
- Where code is stored in a VBA application.
- How procedures are created.

> **Note**
> **As you read through this chapter, we encourage you to also use the companion content sample database, VBAEnvironment.accdb, which can be downloaded from the book's catalog page.**

Debugging Code on a Form

To begin, open the sample database, VBAEnvironment.accdb, which opens the startup form, *frmVBAStartsHere*, shown in Figure 1-1.

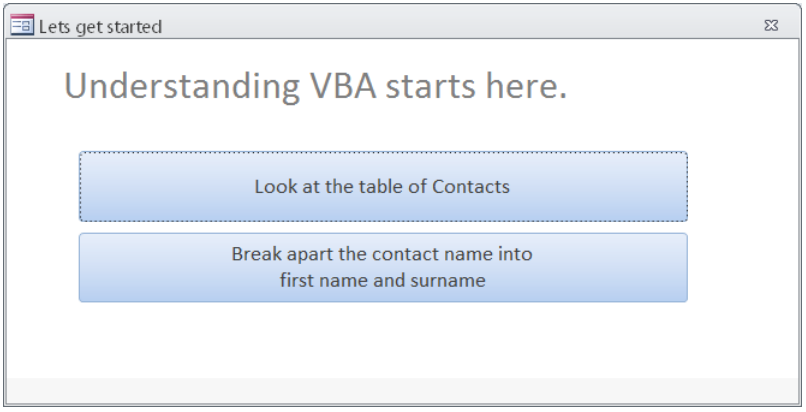

Figure 1-1 The startup form, *frmVBAStartsHere*.

The sample database contains program code with errors intentionally integrated into it. The *frmVBAStartsHere* form is designed to show how the code will break into Debug mode when it encounters an error. As you work through this chapter, you will fix these errors.

Click the button labeled Look At The Table Of Contacts. A pop-up window appears, as shown in Figure 1-2.

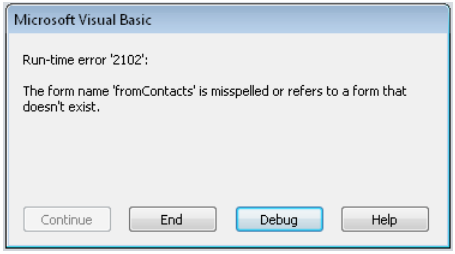

Figure 1-2 In this Access pop-up window, you can either end the code execution or click Debug to investigate the error.

If you click the End button, the program code stops executing. But as you want to debug the code, click the Debug button.

Entering the VBA Editor

When entering debugging mode, the program stops in the VBA editor and highlights the line of code at which it failed in yellow, as shown in Figure 1-3.

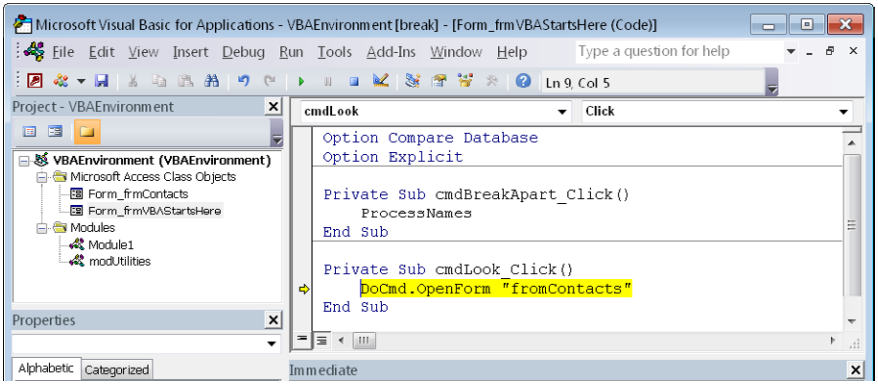

Figure 1-3 Choosing Debug opens the VBA Editor and highlights the program code line that generated the error.

In this example, the problem is a simple spelling error. The database contains a form called *frmContacts*, not *fromContacts*. Access displays an error message that fully describes the problem. It also provides you with the opportunity to edit the text to correct the misspelling, as shown in Figure 1-4.

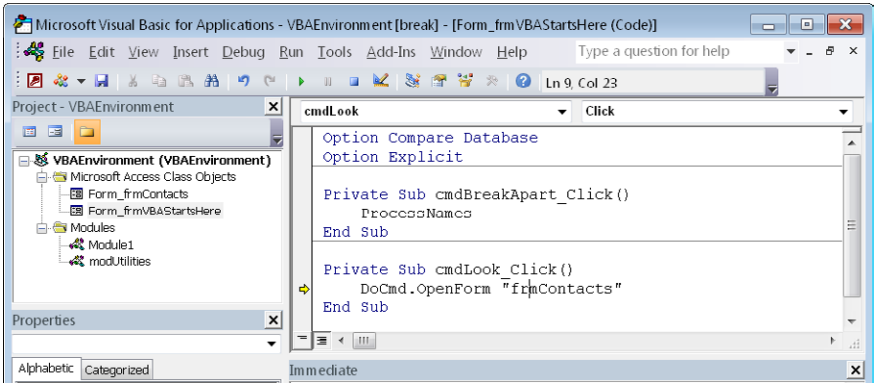

Figure 1-4 Code stopped at the error line. Notice in the Project Explorer pane on the left that the entry form *_frmVBAStartsHere* is highlighted. This tells you that you are viewing the form's code module.

DoCmd.OpenForm is a command that allows the program code to open the specified form. *DoCmd* is a shorthand way of saying, "do the macro command." After correcting the misspelling, you can either press the F5 key or click the Continue button on the toolbar to allow the program to continue execution. Figure 1-5 demonstrates the results after continuing to execute the code, which now opens the *frmContacts* form.

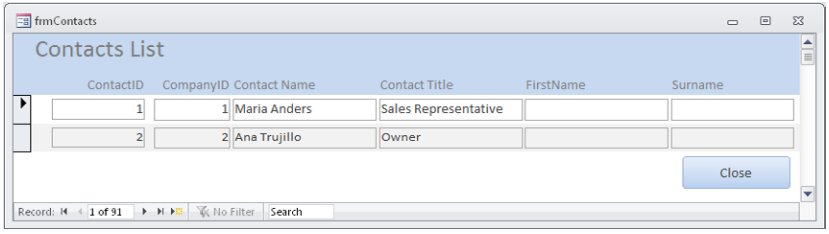

Figure 1-5 After correcting the programming error, you can see the result of executing *DoCmd.OpenForm*, which opens the requested Access form.

The Application and VBA Code Windows

Notice that in your Windows task bar there are two windows open: one window containing your Access application interface, and in the second window, the VBA Editor. When working with application code you can normally switch between the Editor and the application windows, as shown in Figure 1-6.

Figure 1-6 With the VBA editor open, you have two windows for Access, and you can switch between the application window and the VBA Editor window.

If you choose to close the forms you will be prompted to save the changes that you have made to the code on the form, as shown in Figure 1-7.

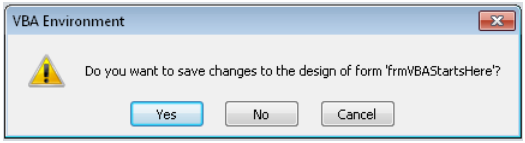

Figure 1-7 The prompt to save changes to the *frmVBAStartsHere* form.

CAUTION!

It is very easy to click the wrong button and lose your design changes. Ensuring that you click the Save button after making any changes to code means that you always know that your changes have been saved. If your program code closes objects as part of its execution, separate dialog boxes for saving changes can pop up, and you can easily forget to save a change. In the unlikely event that the Access application crashes and you have not been saving your design changes, any unsaved changes will be lost.

INSIDE OUT

Code behind a form or report is located in the class module of a form or report

The last example illustrates how program code can be located in a form's class module. Code is written behind a form ("Code Behind a Form" or CBF) to respond to events when the user interacts with the form and the form's controls, Figure 1-8 shows the relationship between controls on a form and the procedures in the form's class module.

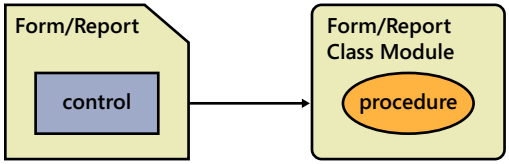

Figure 1-8 Code written in a form or report class module is normally related to events on the form or report, and not normally shared in any other part of the application.

The term *class module* relates to VBA classes discussed later in the book, the form's module is called a class module because it can handle events from the controls and form sections; this is a feature that you can construct within your own module classes.

When code is written behind a form's event, it is a subroutine, but it is also possible to have functions and subroutines on a form that are not directly associated with any single control. This normally occurs when you have a need for an operation to be performed by several controls. In this case, the code will be marked in the General section of the form's class module.

You have now learned that:

- When a code problem occurs, you can click Debug to display the code and fix the problem.
- VBA programs can be edited when the code is paused and then instructed to continue execution after you have fixed any errors.
- Regularly saving your changes after altering code is good practice.
- Program code can be stored in the class module of a form or report.

Creating Modules and Procedures

In the last section, you saw that when the program code goes into Debug mode, the Editor window is displayed. However, you can access the editing environment by using several different methods, as described in the following list:

- Press Alt+F11 (this applies to all Microsoft Office products).
- Press Ctrl+G. This displays the Immediate window in the Editor and automatically opens the Editor window, if it is not already open.
- On the ribbon, on the Create tab, click Module. This creates a new module and enters the Editor.
- In a form or report, on the ribbon, on the Design tab, click the View Code icon.
- Click any of the modules shown in the Navigation window.
- Right-click a Form/Report's sections or controls, and then select Build Event, where there is code written behind an event.

If you are not already in the Editor, then open the sample database and press Alt+F11 to go there.

The VBA Editor comprises a number of windows. If you accidently close one, or need to show a window that is not already displayed, click View on the menubar to open the window, as shown in Figure 1-9.

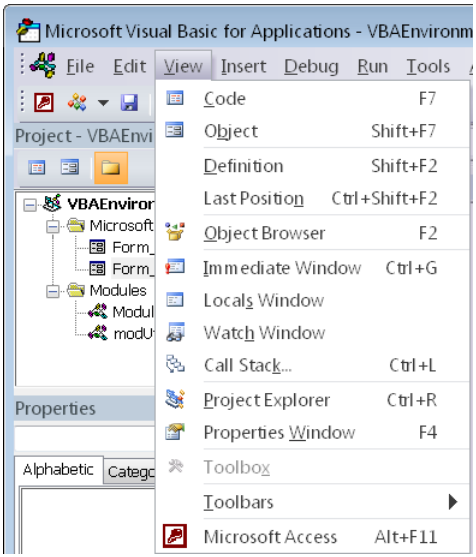

Figure 1-9 From the View menu, you can open different types of Editor windows. Note the Project window in the background with its expandable folders. This is a map of all the code modules in the application. Double-click any form or report to open the document's code module.

The Project pane normally contains two folders. The first folder, Microsoft Access Class Objects, contains your forms and reports (only objects with an associated code module are shown). Clicking one of these objects displays the existing code module. The term Class refers to the special nature of a Form/Report module; it handles the events for the object. These are sometimes simply called Form/Report modules. The separate Modules folder below the Form/Report modules contains general purpose program code that can be used in various parts of your application; these are sometimes called general or global modules (this folder is only shown after you have created a module).

Below the Project pane is the Properties pane for the project. You can use this window to change the name of the project or of a module (see Figure 1-10). The VBA project name property should be changed if you use the operating system to copy a database to create a new file, as the file copy operation does not change the VBA project name inside the database.

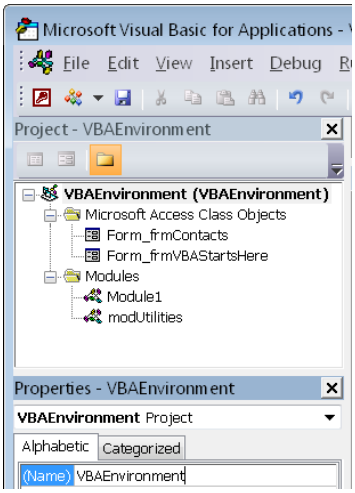

Figure 1-10 The Project pane displays all forms and reports that have code modules. You can use the Modules tab for writing code that is not tied to a particular form or report.

Creating a Module

You can use the Project window to create a new module. There are several different ways to add a new module; the method shown in Figure 1-11 involves right-clicking the Modules tab, and then selecting Insert | Module from the shortcut menu that appears. This method is used when you want to concentrate on setting up new modules when you are in the middle of writing and debugging code.

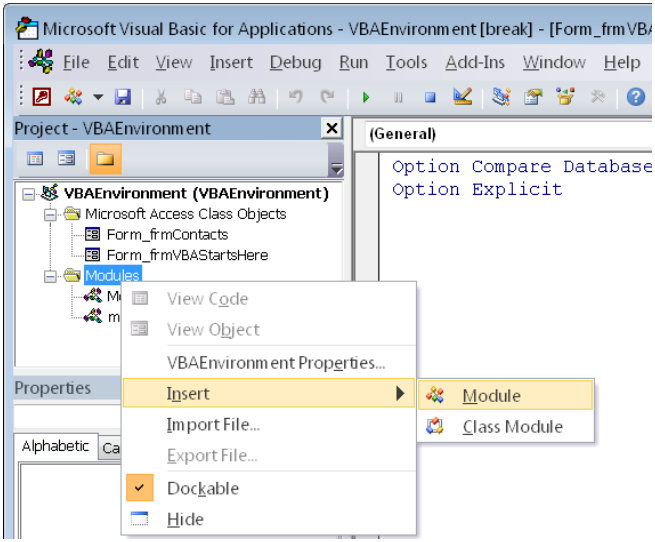

Figure 1-11 Creating a new module or class module from the Project pane.

When you create a new module, it is automatically assigned a default name (for example Module1). When you click the save button, you will be prompted to give the module a permanent, more meaningful name. Figure 1-12 shows the new module before it has been saved with an alternative name. You might also notice that when you save the new module, it contains two special *Option* keyword lines of text. This is explained in detail in Chapter 2, but for the moment, you can ignore this.

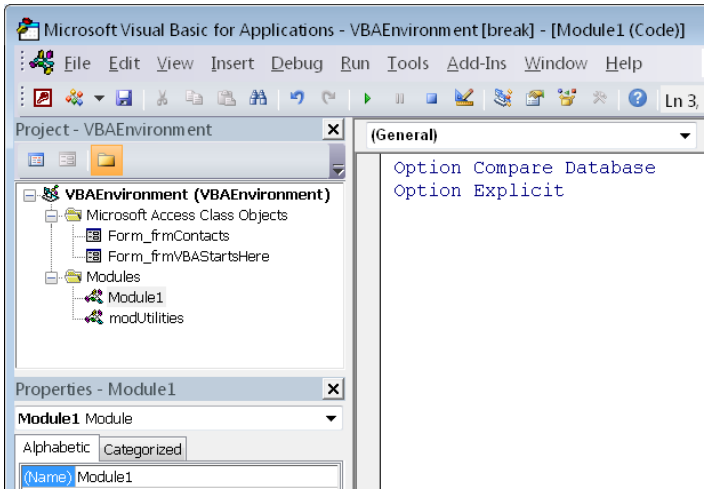

Figure 1-12 After creating a new module, it will be displayed using a default name such as Module1, Module2, Module3, and so on.

When you click the save option on the toolbar or close the database, you are prompted to replace the default module name with something more meaningful.

Creating a Procedure

Modules contain *procedures*, and the procedures contain program code. Use the Insert menu to open the Add Procedure dialog box (see Figure 1-13), in which you can add a new Sub (subroutine), Function, or Property (class modules only). There is also an option to prefix the procedure with the keyword *Static*, which makes variables hold their value when repeatedly executing the procedure (static variables are described in Chapter 2).

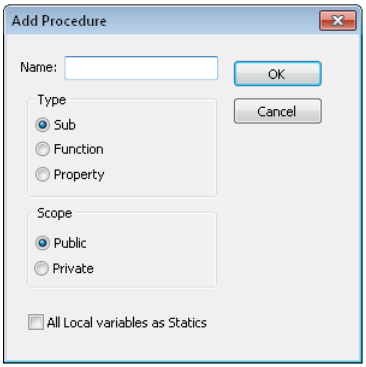

Figure 1-13 The Add Procedure dialog box.

There is another, quicker mechanism for creating a new procedure: click any empty area, type the keyword **Sub** {name} or **Function** {name} (be sure you are not inside an existing sub or function), and then press the Enter key. The VBA environment adds an *End Sub* keyword automatically to complete the procedure block, as shown in Figure 1-14).

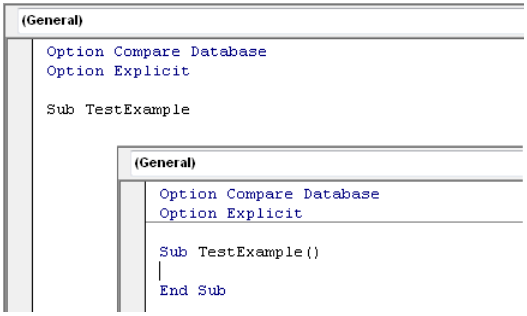

Figure 1-14 Creating a new procedure by using the *Sub* keyword. The window in the background shows the keyword and the procedure name typed in; the foreground window shows the result after pressing return.

Type the word **MsgBox**, enter a space, and then type a double quotation mark. As you do this, IntelliSense assists you as you type in each part of the syntax for the *MsgBox* procedure, as shown in Figure 1-15.

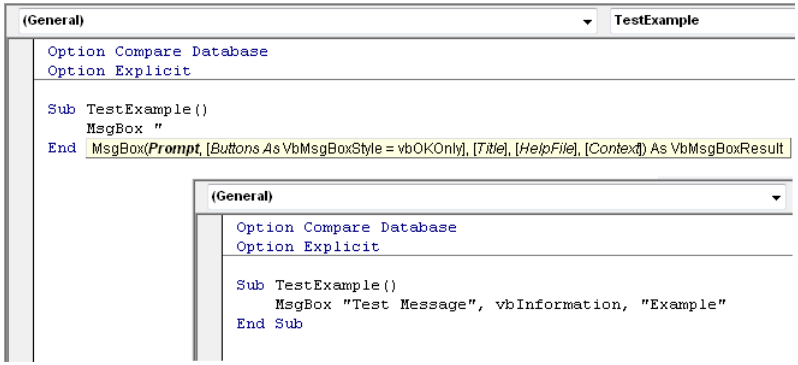

Figure 1-15 The built-in pop-up *MsgBox* procedure has three parts: the text to display; a constant that is used to indicate what buttons and images to display; and finally, the title for the window.

Executing a Subroutine

The *subroutine* code you created can be executed two ways. The first way is to click the green Continue button on the toolbar menu or press the F5 key (you need to have the cursor positioned inside the procedure on any part of the code). This should then display the message box.

The second way is to type the name of the subroutine into the Immediate window, and then press Return, as demonstrated in Figure 1-16.

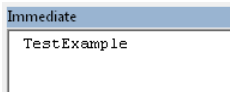

Figure 1-16 You can type a subroutine name into the Immediate window, and then press the Return key to execute it.

The second type of procedure in VBA is called a *function*. The key difference between a function and a subroutine is that functions are always expected to return a value. Functions are fully explained in Chapter 2.

To create a function, you can type **Function *{name}***, similar to the way you entered your subroutine (you should try this).

INSIDE OUT

Changing a procedure type from a subroutine to a function or from a function to a subroutine.

VBA allows you to quickly change a subroutine into a function, and vice versa. After you change the first line of the procedure, the VBA Editor automatically changes the *End Sub* statement to an *End Function* (and all other *Exit Sub* statements to *Exit Function* statements), thereby converting the subroutine into a function. This is very useful if you have larger blocks of code (spotting all the changes to make would be difficult) and leads to improved productivity when developing code. Figure 1-17 shows the original subroutine in the first window (background). In the second (middle) window, you can see the word Sub has been edited to Function. Finally, as shown in the foreground window, when you click off the line of code, the VBA Editor automatically changes the code *End Sub* to *End Function*.

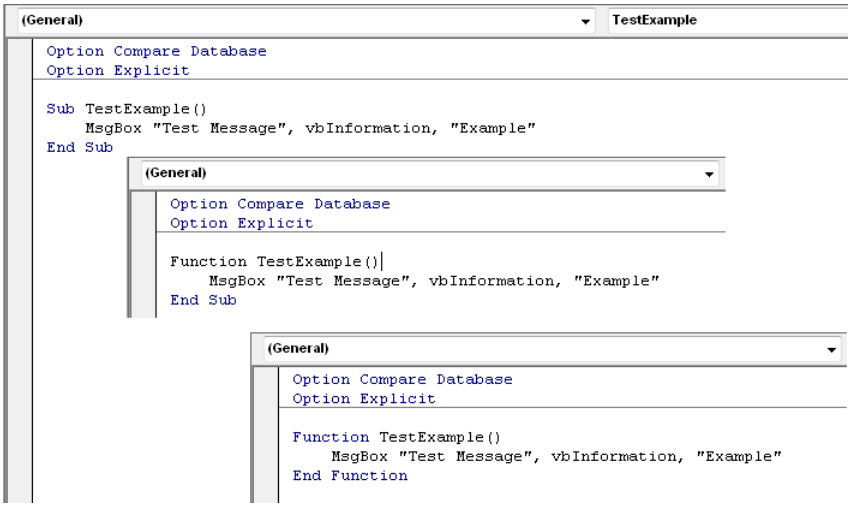

Figure 1-17 As soon as you click off where you replaced the keyword *Sub* with *Function*, VBA changes the *End Sub* to *End Function*.

Because a function returns information, you are going to modify the program code to match Figure 1-18 so that it returns a value.

The *MsgBox* statement can be written in two different ways: the first is to write it when you want to display a message with an OK button (where it looks like a Sub [see Figure 1-17]); the second way is illustrated in Figure 1-18, where you want to gather input from a user (it behaves like a function).

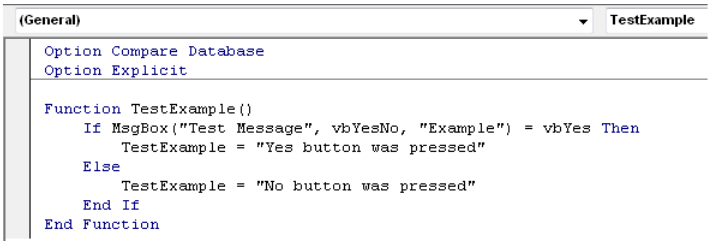

Figure 1-18 The *MsgBox* function prompts the user with two buttons (Yes and No), and then tests to see which button the user pressed.

After you have typed in a call to either a built-in procedure or your own procedure, you can right-click the shortcut menu to display information on the parameters for the procedure or get assistance with selecting constant values (see Figure 1-19). The *MsgBox* function has alternative constants for the second parameter (*vbYesNo*) shown in Figure 1-18, which control the buttons and graphics displayed in a message box. To change a constant value in the *MsgBox* routine, hover the mouse over the existing value, right-click to display the shortcut menu, and then select List Constants. This simplifies entering a new constant value.

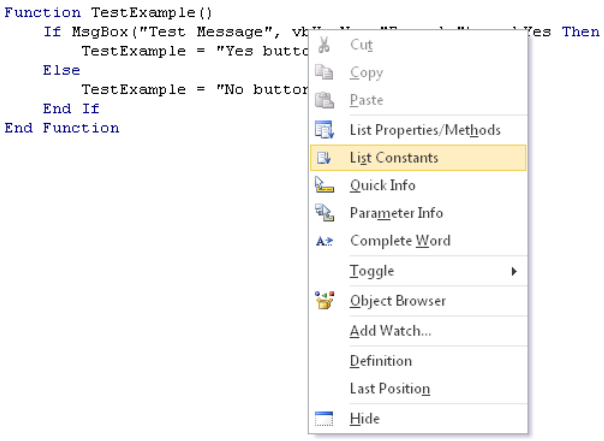

Figure 1-19 Accessing the shortcut menu to display information about the parameters for the procedure. Other options on this menu include providing quick information on the function.

Executing a Function

To run a function, you can press the F5 key, but this will not display the returned value. (In Chapter 2, you will see that functions can be used to assign a returned value to a variable.) You can also call the function from the Immediate window by using the "?" (question mark) symbol adjacent to the function name to display the returned value, as shown in Figure 1-20. Notice that when you execute a function you need to add parentheses "( )" after the

function name; a function needs to show that it accepts parameters even when it has no parameters.

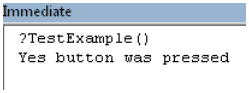

Figure 1-20 Executing a function from the Immediate window. Use the ? (question mark) character to return a value from the function.

In this section, you have seen how program code can be written in a module that is not connected to a form or report. These code units are called *standard modules*, or sometimes general modules or global modules. Figure 1-21 illustrates how a standard module is an object that is independent of any form or report. Compare this to Figure 1-8, which showed a class module of a form or report that is attached to the Form/Report. Code written in these procedures can link to other procedures in the same or different modules. The code will normally not be specific to a single form. Form-specific code is better written inside a form's class module

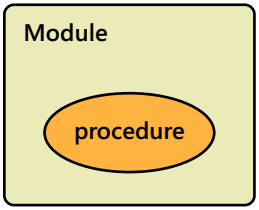

Figure 1-21 A schematic view of a module, which can contain one or more procedures. The procedures can be a combination of functions and subroutines.

You should now understand that program code can be written either in the class module of a form or report (when the code is specific to the Form/Report), or it can be written in a standard module (when it is not specific to a Form/Report).

Viewing and Searching Code

Module code can be viewed either showing the code for a single procedure (Procedure view) or the entire module (Full Module view), using the scrollbars to browse through its contents, as shown in Figure 1-22.

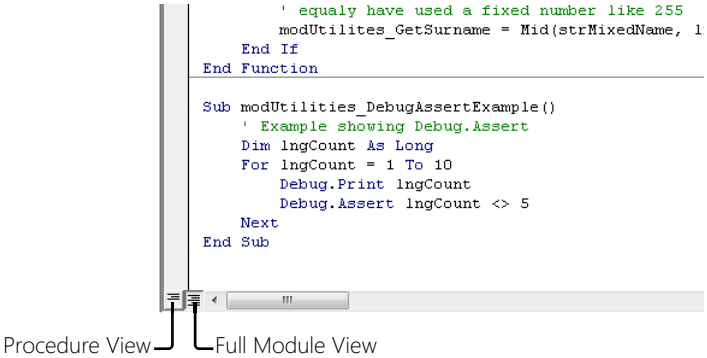

Figure 1-22 Using the buttons in the lower-left corner of the code window, you can display either a single procedure or a scrollable list of all the procedures in the module.

Split Window

The module code window can also be switched to a Split view (see Figure 1-23). This gives you the ability to compare code in two different procedures, one above the other.

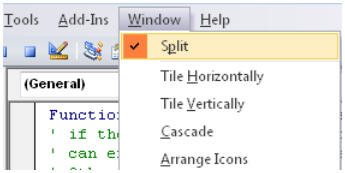

Figure 1-23 Use the Window menu to enable the Split view option.

Drag the splitter bar in the center of the screen up or down to change the proportion of the screen that is used to display each procedure. The scrollbars and the PgUp/PgDown buttons can be used independently in each window to browse through the procedures in the module. Figure 1-24 illustrates the split window view.

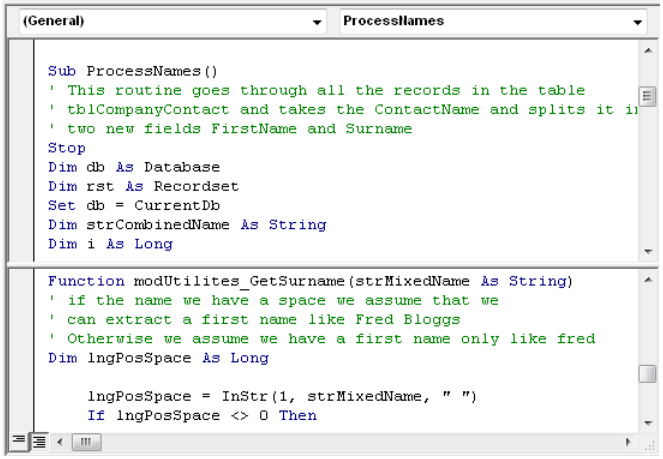

Figure 1-24 Viewing two procedures at the same time in Split view mode.

Dragging the splitter bar to the very top of the screen and releasing it will remove the split view. Similarly, by moving the mouse to the top right, just above the vertical scroll bars, the mouse pointer will change shape and you can drag down the splitter bar (this can be a little tricky to do and you will find the Window menu easier to use for this).

Use the drop-down menu located on the upper-right portion of the window to select any procedure within a module (see Figure 1-25). This applies to each of the windows when using the split view, as well.

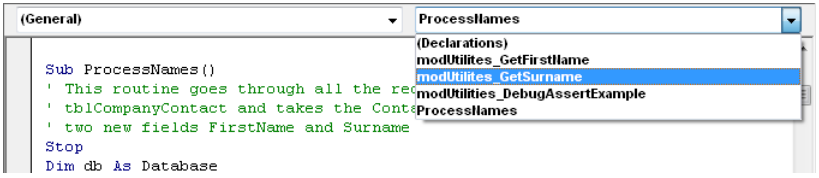

Figure 1-25 Use the drop-down menu to quickly display any function or subroutine in a module. For standard modules the drop-down on the top left only has one available choice called General; for class modules there will be other values shown in the drop-down.

Note

If you click the drop-down menu in the upper-left portion of the window, you will see only the General option. However, if you are displaying a form or report class module, as shown in Figure 1-26, you will see a list of the form sections and controls, and the drop-down menu at the upper-right will now display the events for the object selected in the lefthand list.

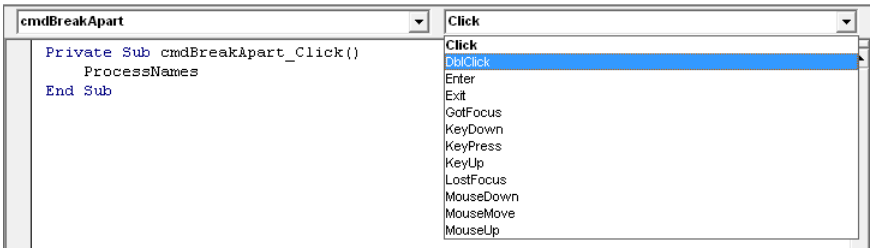

Figure 1-26 In a Form/Report class module, the drop-down menu on the left lists the controls and sections in the document. The drop-down menu on the right shows all possible events for the selected section or control. Events that have code associated with them are displayed in a bold font.

If you have multiple code windows open, you can use the Windows menu to change between the open windows. You also have the option to tile (horizontally or vertically) or cascade the open windows, as shown in Figure 1-27.

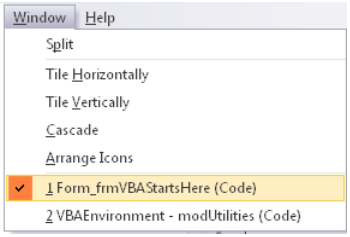

Figure 1-27 The Window menu in the Editor allows multiple, open module windows to be viewed in Tile mode or Cascade mode.

Searching Code

If you need to find a procedure or a piece of code, press Ctrl+F to open the Find dialog box and locate the code in the current procedure, module, project, or block of selected text (use the mouse to select and highlight the text before pressing Ctrl+F), as demonstrated in Figure 1-28.

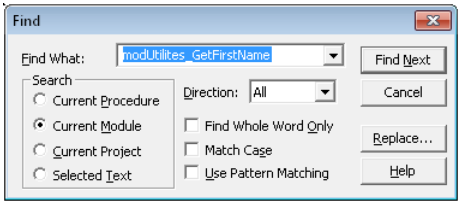

Figure 1-28 Use the Find dialog box to search and replace code fragments within a procedure, module, project, or selected text.

To view the definition of a variable or procedure (see Figure 1-29), position your cursor on it, right-click to open the shortcut menu, and then click Definition. Alternatively, again with your cursor on the procedure or variable, press Shift+F2 to go to the definition. If the code is in a different module, the appropriate module will be opened automatically.

```
Do While Not rst.EOF
    strCombinedName = rst!ContactName
    rst.Edit
    rst!FirstName = modUtilites_GetFirstName(rst!ContactName)
```

Figure 1-29 Viewing the definition of a procedure or variable.

Additionally, referring still to Figure 1-29, if you click the text *modUtilites_GetFirstName* in the subroutine *ProcessNames*, and then press Shift+F2, the body of the code for the procedure is displayed.

Debugging Code in a Module

To demonstrate how code is debugged, we will use a routine that splits a person's name from a combined field in the *frmContacts* form into separate first name and surname. Figure 1-30 shows the Contact Name in the first record split into the FirstName and Surname fields.

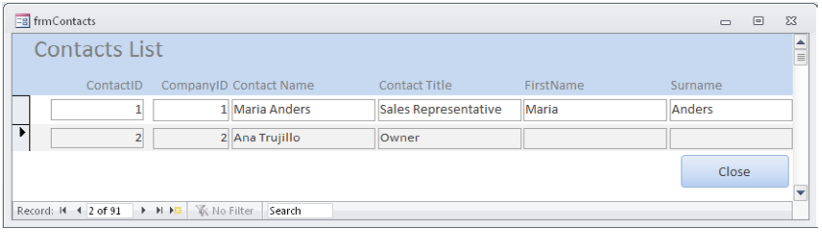

Figure 1-30 Using VBA code, the contact's full name, which is contained in the Contact Name field, is split into corresponding FirstName and Surname fields.

Return now to the opening *frmVBAStartsHere* form, and then press the button labeled Break Apart The Contact Name Into First Name And Surname, as shown in Figure 1-31.

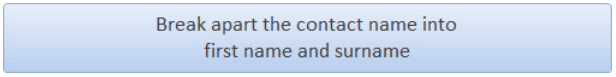

Figure 1-31 Click the Break Apart The Contact Name Into First Name And Surname button on the *frmVBAStartsHere* form to trace through and debug the application code for splitting apart the Contact Name field.

The code will pause at a *Stop* statement, as depicted in Figure 1-32.

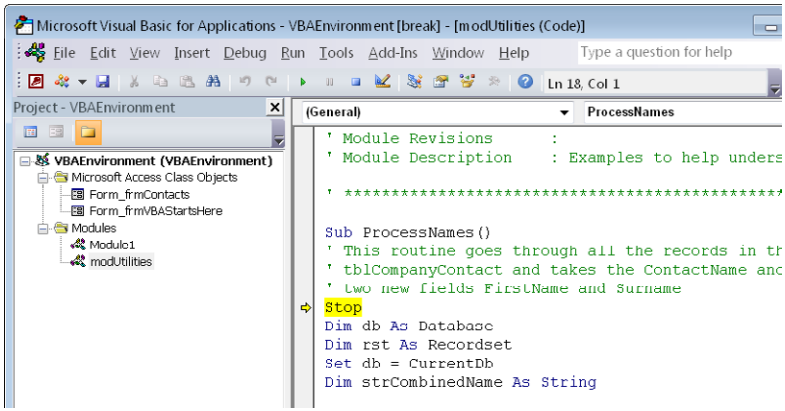

Figure 1-32 Hardcoded (permanent) breakpoints using the *Stop* keyword are a useful reminder when developing code that it is incomplete, but they should not be included in any final application.

Notice in Figure 1-32 that the code has stopped in the *modUtilities* module, and not in the form's class module.

Figure 1-33 presents the code behind the button. This code calls the procedure *ProcessNames* in the module *modUtilities*.

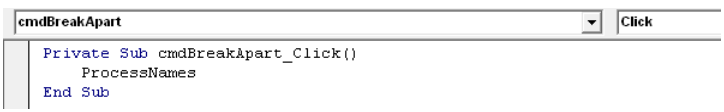

Figure 1-33 The code behind the button is written in the *Click()* event. This code calls the *ProcessNames* routine, which is has been written in a module.

In Chapter 2, you will learn about naming conventions. The convention adopted in this book is to add a prefix to procedures in modules so that we can easily see in which module a procedure is defined. In the preceding example, if you had called the *modUtilities_ProcessNames* procedure rather than *ProcessNames*, it would be easier to see how the code on the form linked to the code in the module (in this case, we have not followed the convention to illustrate the point).

There is another feature in the VBA Editor that can help display how the modules have been linked together. Selecting the Call Stack from the View menu displays the path from the forms class module to the procedure in the utilities module. Figure 1-34 illustrates that this procedure was called from a form (indicated by the "Form_" prefix) with the name *frmVBAStartsHere*, from the control called *cmdBreakApart* on the *Click* event for the control.

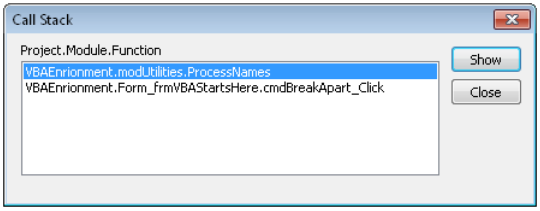

Figure 1-34 The Call Stack is a visual aid that helps to establish where you are in the code. In this example, reading from top to bottom, you are in the code unit *modUtilites_ProcessNames*, which was called from the code unit *cmdBreakApart_Click*, which is in the form *frmVBAStartsHere*.

INSIDE OUT

Creating code in a module and linking the code to the form or report

In earlier sections, you looked at how program code can be written in a form's class module, and then you saw how more general purpose code can be written in a standalone module that is not connected to a form or report. The code on the form or report can be linked to the code in a standalone module. This is shown diagrammatically in Figure 1-35.

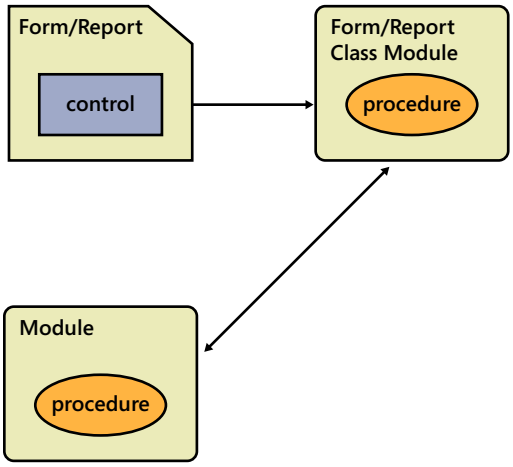

Figure 1-35 Code in a form or report class module can call code in a module. The module can contain code that is used in several parts of the application.

As an alternative to placing the code *ProcessNames* in a module, you can instead either write the code behind the *OnClick* event in the form or add the code as a subroutine to the form's class module. Which of these alternatives you choose depends on whether the code can be used in different parts of the form or in more than one form or report in the application. Because the *ProcessNames* routine can be called from a maintenance form or as part of a process for importing data, we have placed the code in a general purpose utilities module.

Debug Commands

Debugging code involves several operations. These operations are:

- Stopping or breakpointing the code so that it pauses at the correct point for investigation.
- Examining and monitoring variables.
- Modifying and repeating the code execution.

Debug.Print is a command that displays values of program variables or expressions in the Immediate window when developing code:

```
Debug.Print strCombinedName, rst!FirstName, rst!Surname
```

There is another debug command called *Debug.Assert*, which can be used to halt the execution of program code when a specific condition is *False*. For example, the following code halts execution when *lngCount* = 5 (note that the *Debug.Assert* stops when the condition is false):

```
Sub modUtilities_DebugAssertExample()
    ' Example showing Debug.Assert
    Dim lngCount As Long
    For lngCount = 1 To 10
        Debug.Print lngCount
        Debug.Assert lngCount <> 5
    Next
End Sub
```

Breakpointing Code

The *Stop* and *Debug.Assert* statements are hardcoded breakpoints, but you can also have soft breakpoints that you can use when you interact with a block of code and need to find out why the code is failing or behaving in a particular way.

There are three ways to enter a breakpoint. First, you need to locate the line in which you want to insert the breakpoint, and then do one the following:

- Press the F9 Key.
- On the Debug tab, click Toggle Breakpoint.
- Click in the margin next to the line of code (this is the easiest method).

Figure 1-36 shows the code paused at the *Stop* statement and a soft breakpoint highlighted farther down the page.

```
(General)                                    ProcessNames
Sub ProcessNames()
' This routine goes through all the records in the table
' tblCompanyContact and takes the ContactName and splits it into
' two new fields FirstName and Surname
Stop
Dim db As Database
Dim rst As Recordset
Set db = CurrentDb
Dim strCombinedName As String
Dim i As Long

Set rst = db.OpenRecordset("tblCompanyContact", dbOpenDynaset)
Do While Not rst.EOF
    strCombinedName = rst!ContactName
    rst.Edit
    rst!FirstName = modUtilites_GetFirstName(rst!ContactName)
    rst!Surname = modUtilites_GetSurname(rst!ContactName)

    Debug.Print strCombinedName, rst!FirstName, rst!Surname

    rst.Update
    rst.MoveNext
Loop
End Sub
```

Figure 1-36 The code discontinues execution at the *Stop* statement. Note the highlighted breakpoint farther down the page

Unlike *Stop* statements, which need eventually to be removed from the code, breakpoints are not remembered after you close the database. You can use the Debug menu to clear all breakpoints in the application, or you can press Ctrl+Shift+F9.

With the breakpoint set, you want the code to execute until it reaches it. Use the Continue button (see Figure 1-37) or press F5 to instruct the code to continue execution until it either completes, or reaches a breakpoint.

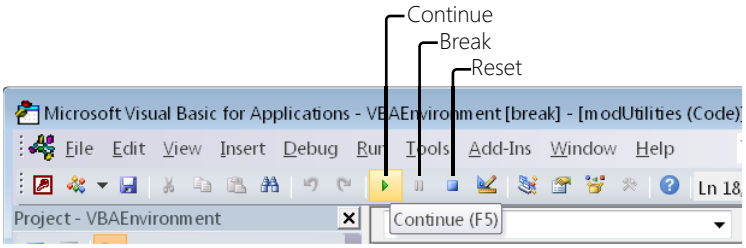

Figure 1-37 Three of the buttons on the Run menu are also displayed on the menu bar—Continue (F5), Break (Ctrl+Break), and Reset (which halts code execution).

Press F5 to continue the code execution to reach the breakpoint shown in Figure 1-38.

```
Set rst = db.OpenRecordset("tblCompanyContact", dbOpenDynaset)
Do While Not rst.EOF
    strCombinedName = rst!ContactName
    rst.Edit
    rst!FirstName = modUtilites_GetFirstName(rst!ContactName)
    rst!Surname = modUtilites_GetSurname(rst!ContactName)

    Debug.Print strCombinedName, rst!FirstName, rst!Surname

    rst.Update
    rst.MoveNext
Loop
```

Figure 1-38 Code continues to execute until it either reaches the next breakpoint or completes execution.

The *ProcessNames* routine is an example of programming with a *RecordSet* object, which is discussed in Chapter 5, "Understanding the Data Access Object Model." The program code loops through each record in the table and changes the *Firstname* and *Surname* fields.

If you switch to the Access application window and open the table *tblCompanyContact*, you can investigate whether your code has worked. And as it turns out, it has not worked as desired; Figure 1-39 shows that the entire contact name has been copied into the *First-Name* field. The name was not split apart, as intended.

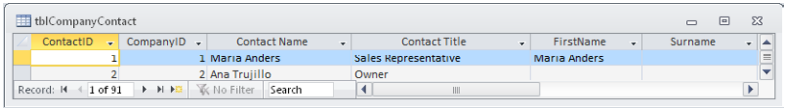

Figure 1-39 With the code paused at a breakpoint, you can switch to the application window and open other Access objects (in this case a table) to see the changes made to the data. Here, you can see that the code has not split apart the Contact Name.

Set Next Command

If you move the cursor over the first line in the loop and then right-click, you can use the *Set Next* statement to make the code go back and repeat the operation. This is typical of how code is debugged. After identifying an error, you can move back to an earlier point in the code to investigate it.

To change the current execution point to a different line of program code, place the cursor on the line that begins with `strCombinedName =`, right-click to display the shortcut menu, and then click Set Next Statement, as shown in Figure 1-40.

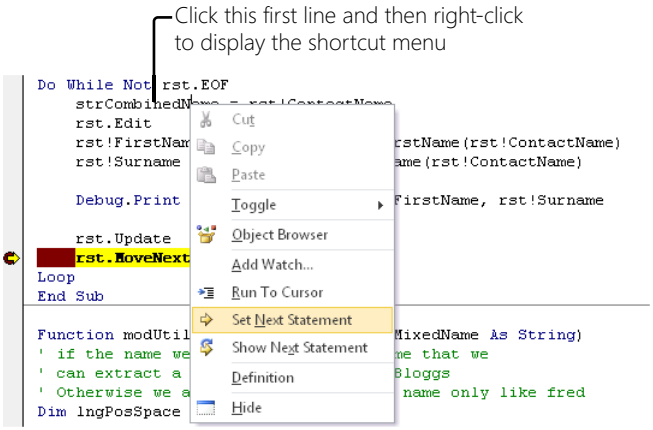

Figure 1-40 Changing the current execution point to a different line by using Set Next Statement.

After you click Set Next Statement, the yellow highlighted line changes, as shown in Figure 1-41. Notice also that you can display the values of the variable by hovering the mouse over it. (This is not restricted to variables in the highlighted line of code; you can hover the mouse over variables on other lines to view their values, too.)

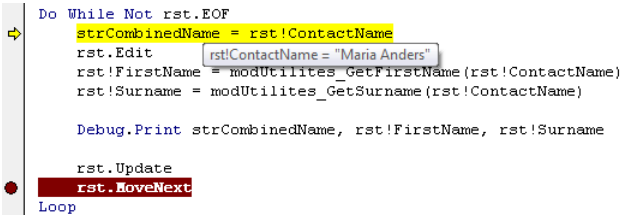

Figure 1-41 Hovering the mouse over any variables in the program code will display the variable values.

As an alternative to using Set Next Statement to change the execution point, you can also grab the yellow arrow on the side margin and drag it to a different line of code.

Breakpoint Step and Run Commands

You now know that this code has a fault, but rather than using the Continue (F5) execution method that you just saw in the previous section, you can single step through the code to locate the problem by using the Debug menu or hotkeys, as shown in Figure 1-42.

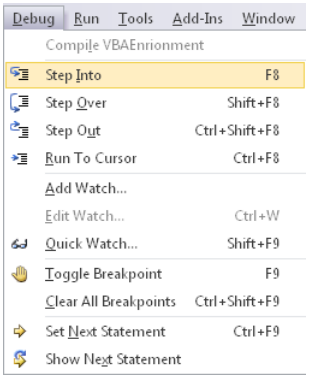

Figure 1-42 Using the Step commands on the Debug menu, you can trace through the execution of your code.

You can do this in several ways. One way is to keep clicking the Debug menu options, but it is much faster to use the following function key combinations to step through the code:

- **F8** Follows the code execution to the next step.
- **Shift+F8** Moves over a procedure call and executes everything in the procedure, but does not show you the detailed execution steps.
- **Ctrl+Shift+F8** Indicates that you have examined the procedure in enough detail and want to complete the execution of this current procedure, but stops once you have returned to the calling code.
- **Ctrl+F8 or right-clicking and selecting Run To Cursor** Process all the lines until you reach the current position of the cursor.
- **Locate a line, right click, and then select Set Next Statement.**

It is important to remember that when you press either Shift+F8 or Ctrl+Shift+F8, both operations cause any code to execute. If you do not want the code to execute, then locate the next line that you do want to execute, and then use Set Next Statement to change the execution point.

For the purposes of this example, keep pressing the F8 key until you arrive at the point shown in Figure 1-43.

Figure 1-43 shows the unmodified code and the mouse hovering over the variable. The displayed value for the variable leads you to spot the logical error.

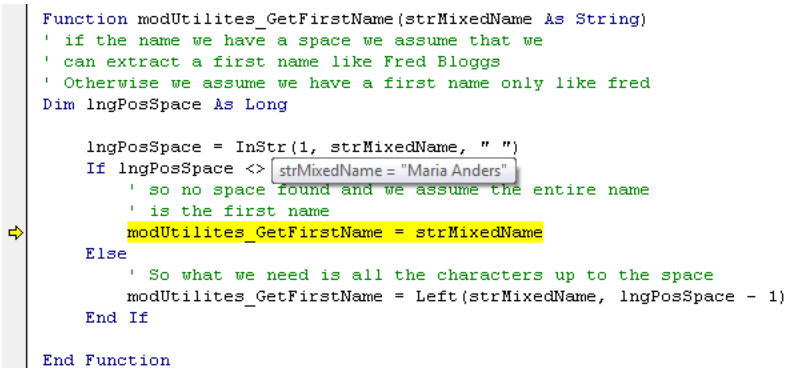

Figure 1-43 Pressing F8 repeatedly brings you to this point in the code. Notice the displayed value for the variable.

The bug in this code occurs because of a space in a name. The position of the space could be represented by a value of `lngPosSpace` 6, yet the code states that when `lngPosSpace` <> 0, we have found the entire name. So the logical test is the wrong way around. The following line needs to be changed from:

```
If lngPosSpace <> 0 Then
```

to:

```
If lngPosSpace = 0 Then
```

The problem with the code in Figure 1-43 is that it has branched into the wrong part of the processing. You would have expected the code to branch into the statements after the *Else* keyword. The mistake here is in testing for <> when you should be testing for =. You need to now fix the code.

To fix the code, edit the <> to an = sign, as shown in Figure 1-44. Then right-click the line containing the *IF* statement and select Set Next Statement (this means that we can repeat the last action). Figure 1-44 shows the modified code and the result of selecting Set Next Statement to change the execution point back to the line containing the coding error.

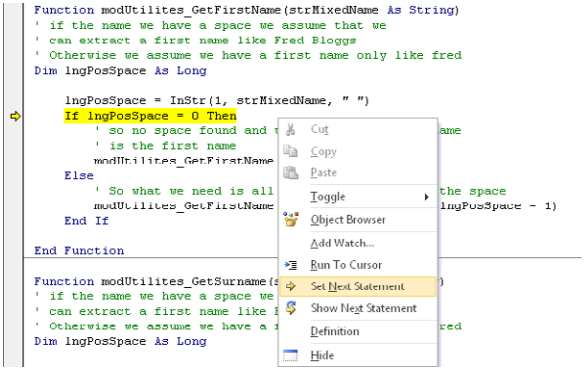

Figure 1-44 After changing the <> operator to =, right-click the mouse over the line where you changed the code and select Set Next Statement to go back and repeat executing the step from the code line that has now been corrected.

As before, press F8 to follow the code execution (you will also need to fix a similar coding error in the procedure *modUtilites_GetSurname*). Figure 1-45 shows how the code execution point has branched to the correct point to extract the first name.

```
Function modUtilites_GetFirstName(strMixedName As String)
' if the name we have a space we assume that we
' can extract a first name like Fred Bloggs
' Otherwise we assume we have a first name only like fred
Dim lngPosSpace As Long

    lngPosSpace = InStr(1, strMixedName, " ")
    If lngPosSpace = 0 Then
        ' so no space found and we assume the entire name
        ' is the first name
        modUtilites_GetFirstName = strMixedName
    Else
        ' So what we need is all the characters up to the space
        modUtilites_GetFirstName = Left(strMixedName, lngPosSpace - 1)
    End If

End Function
```

Figure 1-45 This time, pressing F8 to step through the code takes the program to the correct processing statements.

There are a number of ways to see the result of evaluating an expression. The easiest method is to hover the mouse pointer over the expression, but you can also paste a code fragment into the Immediate window and see the result before executing the line of code (this is useful when you want to see the values for different parts of a complex expression).

Displaying Variables in the Locals Window

The Locals window gives you an instant view of the values in your program variables. This is particularly useful for complex variables that have many components, such as a *Recordset*. Figure 1-46 displays the local variables in your procedure.

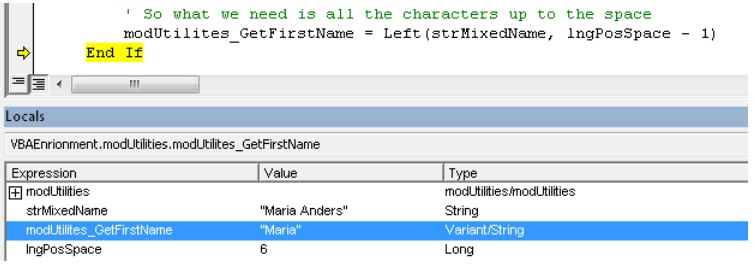

Figure 1-46 You can use the Locals window to both display and change values in variables.

In either the Locals window or the Immediate window, you can directly edit the values in variables, as shown by the highlighted value in Figure 1-47.

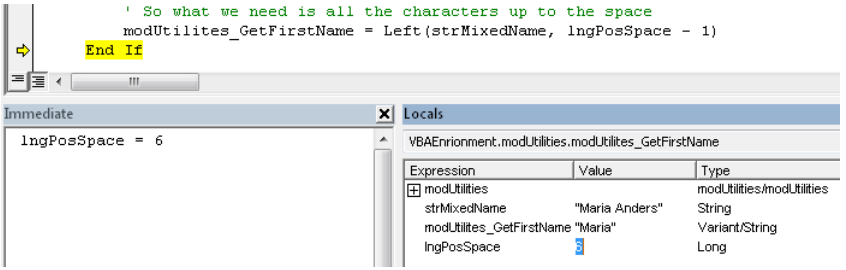

Figure 1-47 Variables can also be assigned values in the Immediate Window.

Tracing Procedures with the Call Stack

The Call Stack shows you where you are once your code has moved through several layers of execution (see Figure 1-48). You can also use it to move to any of the procedures shown by just clicking on the procedure itself in the Call Stack dialog box and then pressing the Show button.

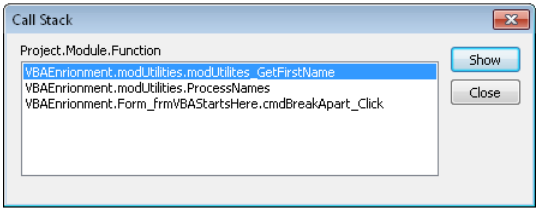

Figure 1-48 You can use the Call Stack to help find where you are in your code, or you can use it to move directly to a procedure.

In Figure 1-48, the top line in the Call Stack dialog box shows the current routine that is executing. Below that is the succession of routines that were called to take the execution to its current point. Double-click any routine in the call stack to display that routine's code (note that the execution point remains unchanged if you do this).

Watching Variables and Expressions

The Watches window is particularly useful for monitoring values as you iterate in a loop. With the Watches window displayed, you can right-click and add an expression or variable to be monitored. Figure 1-49 shows the shortcut menu to add a *Watch* variable.

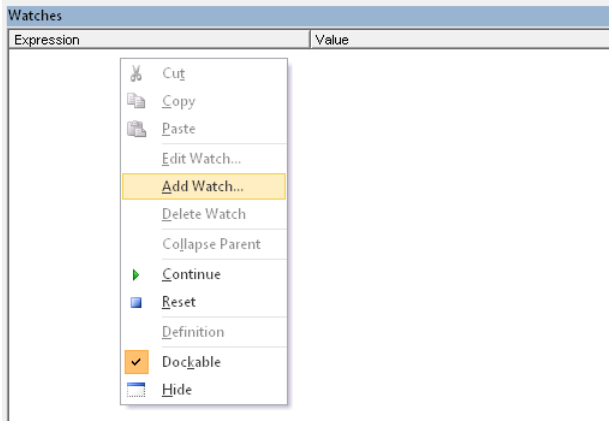

Figure 1-49 The Watches window is particularly useful when debugging repeating loops in code.

INSIDE OUT

Investigating values in variables with complex structures

Normally, *Watch* variables are simple values, but if you add a more complex type of object (in this case a field from a *Recordset*), you get a lot more information. Figure 1-50 shows the result of adding a *Recordset*'s field value to the Watches window. This kind of variable is discussed in Chapter 5, and at this point, we only want to illustrate how more complex objects can be examined by using the Watches window.

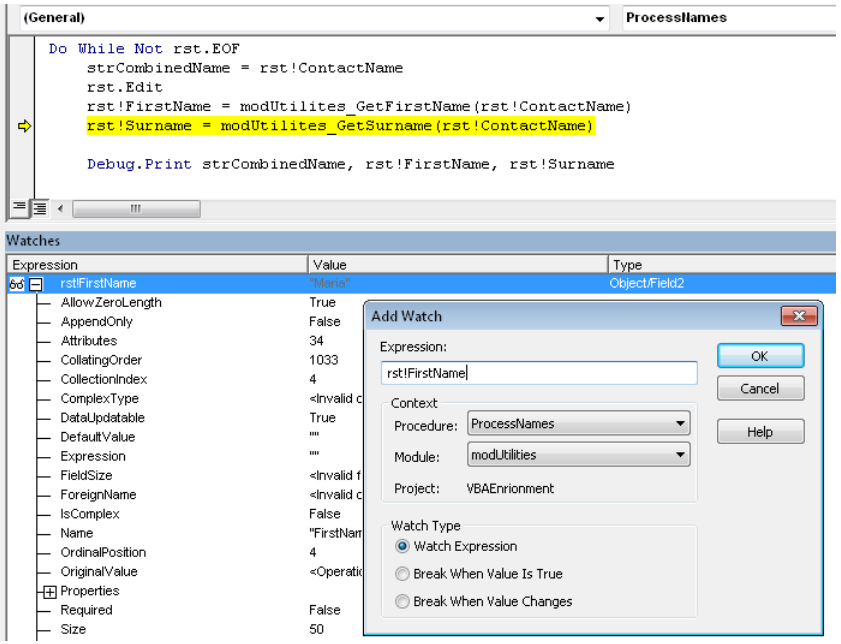

Figure 1-50 A *Recordset* variable is an object variable; rather than holding a single value, it has a more complex structure, shown here being added to the Watches window.

Figure 1-51 demonstrates how more complex variables can be directly edited in the Watches window. You might find this easier than changing values in the Immediate window.

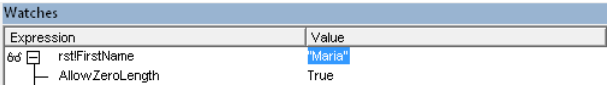

Figure 1-51 The values for watched variables can be directly edited.

The ability to drill up and down into more complex structures is also a feature shared by the Locals window.

Adding Conditional Watch Expressions

Rather than use *Debug.Assert* or modify your code with a *Stop* statement, you can add expressions to conditionally pause the execution of your code when an expression is *True* or when a value changes. Figure 1-52 shows the inclusion of a *Watch* variable that will cause the code to break execution when a specific condition holds.

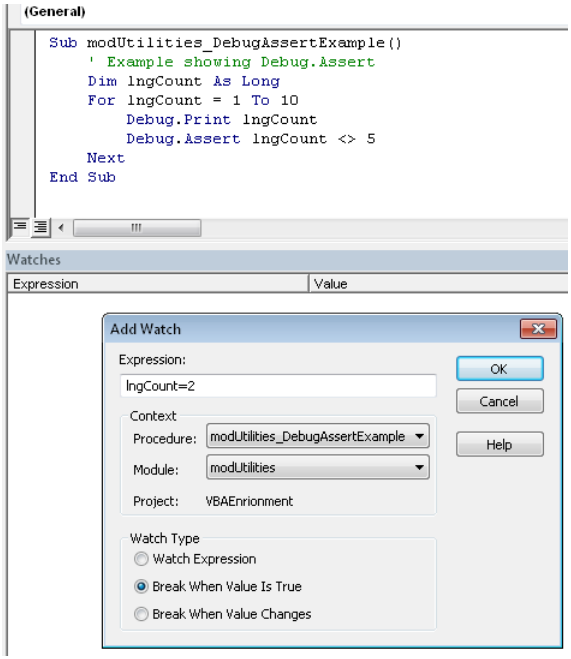

Figure 1-52 Adding a *Watch* expression to break the execution of the code.

One last word regarding the Watches window: be aware that the settings are not permanent. They are cleared once you exit the application.

Working with the Immediate Window

The Immediate window is a scratch pad for performing calculations as well as a powerful tool to display and modify properties of tables, queries, and forms as they are executing. Figure 1-53 presents some examples that you should try typing into the Immediate window. Type a question mark beside the item that you want to calculate, and then press Enter.

The Immediate window will continuously scroll as more information is displayed and there is no option to clear the window (to clear the window, you highlight all text in the window and press the Delete key).

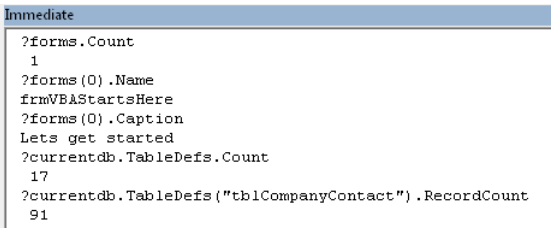

Figure 1-53 The Immediate window is a combination scratch pad and a tool to display and modify properties of tables, queries, and forms.

Changing Code On-the-Fly

Throughout this chapter, you have seen how to change your program code while it is executing, and you might wonder if there are limitations on doing this? The answer is yes, but it doesn't often get in the way of your development.

In the example shown in Figure 1-54, we have defined a new variable while the code is executing.

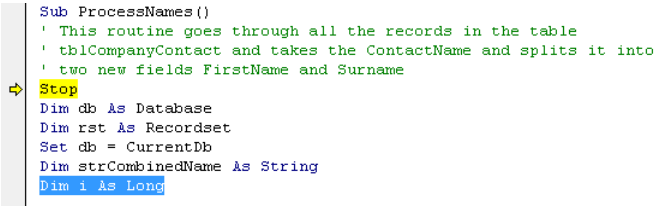

Figure 1-54 The new variable 'i' has been added while code is executing.

If you try deleting (or changing) variables while the code is executing, you will be presented with a warning that this will cause the code to stop executing (you might decide to add a comment after the variable to remind yourself to delete it later when the code is no longer executing).

For example, if we now decide that we have made a mistake and want to change the name of our new variable in Figure 1-54 from 'i' to something different, then you will see the warning shown in Figure 1-55. This means that you either must ignore your change (select Cancel and fix it later) or stop code execution.

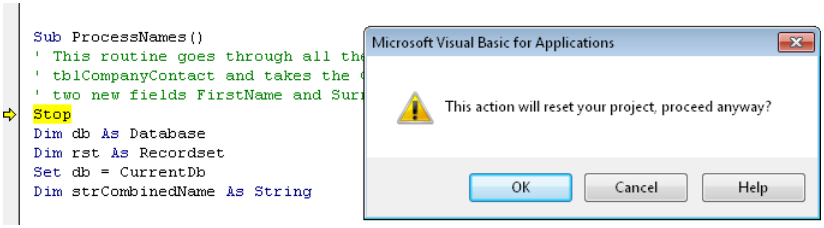

Figure 1-55 A warning appears if you attempt to delete variables while the code is executing.

Using the Object Browser and Help System

In this section, you will look at how you can configure the behavior of the Help system and the use of the Object Browser as an alternative method for locating help on objects.

Configuring the Help System

VBA has an excellent Help system. To use it, simply click a word that you do not understand, and then press F1. However, it's best to have the Help system set to work with Show Content Only From This Computer; otherwise, many of the help topics might not easily locate help for a particular keyword, function, or method. Figure 1-56 shows this setting being changed at the bottom of the figure.

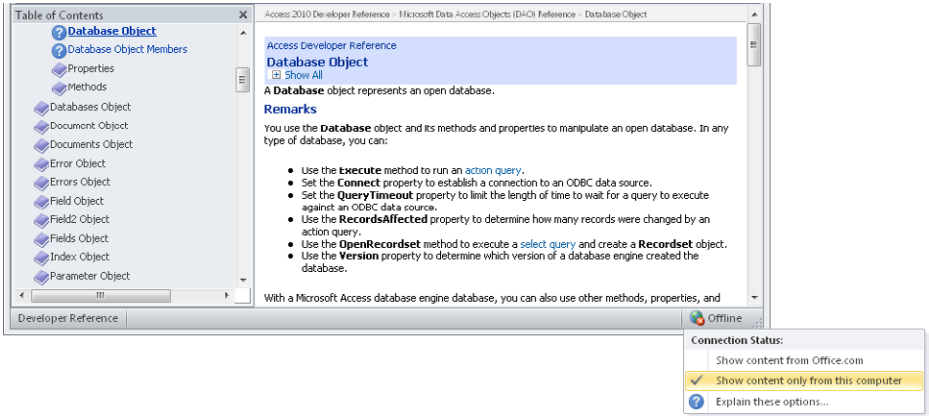

Figure 1-56 With the Help screen open, setting the Help system to Show Content Only From This Computer can offer better identification of keywords.

Access comes with an extensive Help system, and by highlighting a keyword in code (for example *CurrentDb*) and pressing F1, you can display help on the statement or object, as shown in Figure 1-57.

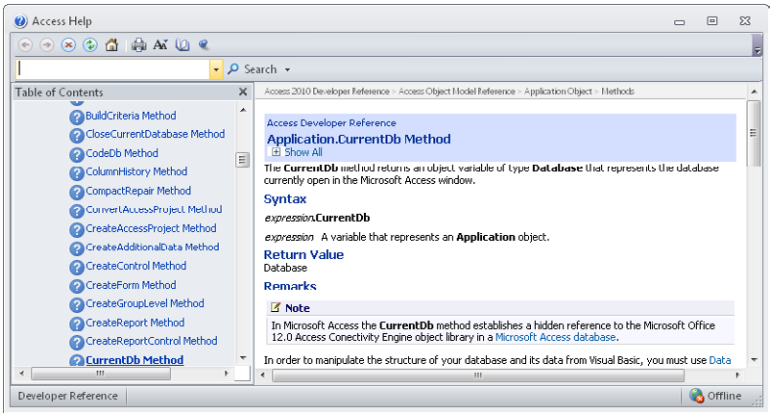

Figure 1-57 Press F1 when you have the cursor on a VBA keyword to locate the keyword in the VBA Help system.

Working with the Object Browser

As you move into more advanced programming (as well as progress through this book), you will see that when you work with objects outside of the Office suite, getting help by pressing F1 will not always display the help information for the object. In this case, you can use the Object Browser (Figure 1-58) either by using the toolbar or pressing the F2 key. Later in this book, we add references to other libraries (for example Microsoft Excel). Help is then also available on these external libraries through the object browser.

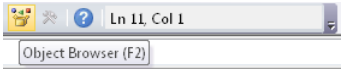

Figure 1-58 You can use the object browser to locate objects and help on your project code and the built-in features in Access.

The object browser can be used for code units designed in your application, external referenced programming units, and Office components including Access (Figure 1-59) where we have searched for the text *Currentdb*.

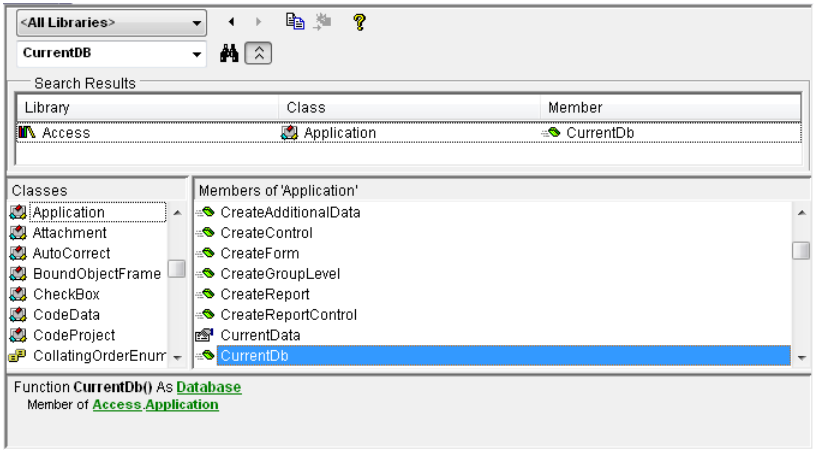

Figure 1-59 When you have located an item in the object browser, press F1 to display the help file that describes the object.

Summary

The VBA Editor and debugging environment in Access offers many useful features to assist you and enhance your productivity in developing applications. Among these features are:

- VBA allows you to significantly modify program code while it is executing. The features for stepping backwards and forwards through the code while it is paused permits you to very quickly isolate logic errors in your code and rectify them. There are minor restrictions on the changes that you can make to the code without the need to restart the code.
- The ability to have both code executing and being able to redesign associated objects such as queries and forms (other than the form that is currently executing the code) is another useful productivity feature.
- The Immediate window is one of the most productive features of the VBA environment. With it, you can test and modify properties of the executing form while the form is executing code.
- The search features that allow you to locate code either by pressing Shift+F2 on an executing procedure or Ctrl+F for general searching. Again, these tools offer unique productivity.

We end this chapter with some general comments on developing within the VBA environment.

Mixed Versions of Access

Since Access 2007, you might experience problems if you are developing with multiple versions of the Office products on a single computer. This is because different versions of the product require different core libraries to be loaded when switching between the versions. Although it is possible to develop with multiple versions on a single computer, it is not recommended, and we would suggest that for all versions prior to Access 2007, you can use a single computer, but for versions including and after Access 2007, you should consider having separate virtual or physical computers. There is a switch over feature to support different versions on a single computer, but you might find that either it takes an unacceptable amount of time to switch or you easily become vulnerable to any issues if library references are not correctly switched over.

Expression Builder

The Expression Builder is an indispensible tool when building applications to find the correct syntax when referring to controls on a form. Unfortunately, the VBA environment does not have an Expression Builder option. The easiest way to get around this problem is to go into the Query design tool, create a dummy query, and then go to the Criteria and right-click, selecting Build, which will bring up the Expression Builder (Chapter 4, "Applying the Access Object Model," discusses this in more detail).

Object Browser

When using 32-bit Microsoft ActiveX controls in a 64-bit operating system, the controls might appear to work well, but there appear to be problems that cause Access to crash when using the Object Browser to display the associated help information.

Debugging Modal Forms

When a user is interacting with a modal form, he or she cannot interact with other objects on the desktop. Debugging code on modal forms is more challenging because you cannot easily interact with other Access objects, such as checking data values in a table or query. The best advice here is to remove the modal property when debugging the form and then set it back to modal once you have resolved any problems in your code.

CHAPTER 9

Adding Functionality with Classes

You have seen in earlier chapters how Microsoft VBA program code is either contained in a module or held in a form's class module. In this chapter, you look at how VBA also allows you to construct your own class modules.

It is often overlooked that VBA supports Object-Oriented Programming (OOP), so in this chapter, we introduce you to OOP concepts by having you construct your own classes. Many Microsoft Access developers take a look at classes and then give up because they have difficulty seeing the benefit and justification for using classes. It's true that much of what can be achieved with a simple class can also be achieved by using libraries of code, and that to build classes you often need to put in more effort during the initial development, but there are benefits in using classes that will be explored in this chapter as well as in Chapter 10, "Using Classes and Events," and Chapter 11, "Using Classes and Forms."

This chapter focuses on two examples of classes, and uses each example to introduce the techniques for creating your own classes.

The first example involves applying classes to solve a problem of designing a dynamic *Tab* control that saw in Chapter 7, "Using Form Controls and Events." This example will demonstrate how classes can be used to improve the design of a general purpose tool that can be re-used in your applications.

The second example looks at how to build classes to handle data for a specific business problem.

After reading this chapter, you will:

- Understand how to create class modules.
- Know how to use *Let*, *Get*, *Set*, and *New* with classes.
- Be able to create collection classes.
- Be able to create base and derived classes.
- Be able to create a hierarchy of classes.

> **Note**
>
> **As you read through this chapter, we encourage you to also use the companion content sample databases, BuildingClasses.accdb and BuildingClassesAfterExportImport.accdb, which can be downloaded from the book's catalog page.**

The object-oriented view to developing software became popular in the 1980s, and in addition to OOP, many terms such as Object-Oriented Design (OOD) and Object-Oriented Analysis (OOA) became increasingly popular.

You have already seen many examples of working with objects in Access. These objects have properties that describe the object, and methods that cause an object to perform an operation. Access maintains collections of like objects; for example, the *Forms* collection, which contains *Form* objects that open on the desktop, and the *TableDefs* collection in the Data Access Object (DAO) model, which contains all the *TableDef* objects. These are examples of working with objects, but not examples of OOP.

OOP Programming (which is supported in VBA) means taking these ideas of working with objects and extending this concept to guide how program code is written.

Classes can be applied in several different ways in Access to:

- Improve the quality of code (OOP can help you develop more maintainable code).
- Extend form/report behavior (OOP allows you to take control of the underlying behavior of Access objects and wrap or extend the behavior).
- Integrate External Components (some external components do not expose all their functionality and OOP features can help with this).

Improving the Dynamic *Tab* Control

In Chapter 7, you saw how to design a dynamic *Tab* control form that can load and unload pages by using an array of Types, where each item in the array corresponds to a form that is loaded into a subform control. The type structure for that is as follows:

```
Private Type PageInfo
    strPageName As String
    strPageSubForm As String
    strRelatedPage As String
    blCanBeLoaded As Boolean
End Type

Dim AvailablePages() As PageInfo
```

As an alternative to using a Type, you will define these pages as objects with properties that correspond to each part of the Type structure, and then you will build a collection to hold these objects, which replaces the array that held the types.

We need the following properties for our object:

- *PageName*
- *SubFormPageName*
- *RelatedPageName*
- *CanBeUnloaded*

You might have noticed that we have renamed the *CanBeLoaded* property in the preceding list to *CanBeUnloaded*. This is because an object-oriented perspective helps you to think in terms of how an object's state can be changed, so this is a more appropriate term to use. With the object's basic properties determined, you can now proceed to create the object class.

Creating a Class Module

To begin, in the Project pane, you create a new class module, as shown in Figure 9-1.

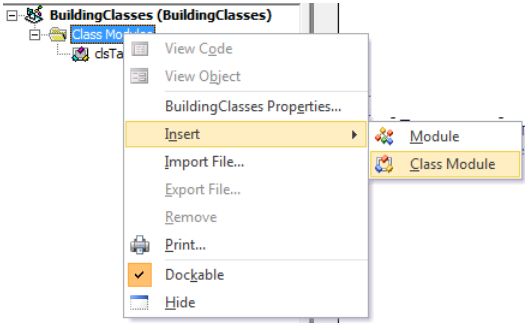

Figure 9-1 Use the Project pane to create a new class module.

With this file created, you then save it using an appropriate class name; for this example, use *clsTabPage*. Because you are now working in a class module, you do not need to explicitly define that you are creating a class (as you would need to do in Microsoft .NET). Next, you define the object's internal variables at the top of the class module code, as illustrated in the following:

```
Option Compare Database
Option Explicit

' These could be declared as either Dim or Private
' as within a class their scope is restricted
Dim p_PageName As String
Dim p_SubFormPageName As String
Dim p_RelatedPageName As String
Dim p_CanBeUnloaded As Boolean
```

Note that these variables include the prefix "p_" to indicate that they are private variables to each class object (other popular prefixes include "m" or "m_"). The next step is to provide the user with a way of reading and writing these variable values.

The *Let* and *Get* Object Properties

After you define the object's internal variables or attributes for your class, you need to create a mechanism to read or write these values. To do this, you define properties. On the Insert menu, click Procedure to open the Add Procedure dialog box, as shown in Figure 9-2.

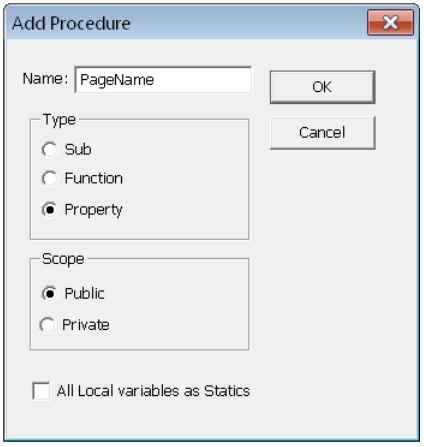

Figure 9-2 Use the Add Procedure dialog box to create a new private or public property.

Ensure that you are not clicked inside any other property when you insert a new property; otherwise, it will fail to add the property correctly to the class. The code that is created needs appropriate data types to be specified for the return type of the property and the parameter type passed to the property.

As shown in the code that follows, you use the *Get* statement to read an object property from the internal private variable, and the *Let* statement to assign a value to the internal private variable. An object can have a number of internal variables, but you might only need to make a few of these available to the user. The idea is to keep the object's external interface very simple, exposing only the minimum number of essential features that a user will need. It is up to you to decide for which properties you want both a *Let* and *Get*, depending on whether the property is to be read-only (*Get* but no *Let*) or write-only (*Let* but no *Get*):

```
Public Property Get PageName() As String
    PageName = p_PageName
End Property
Public Property Let PageName(ByVal PageName As String)
    p_PageName = PageName
End Property
Public Property Get RelatedPageName() As String
    RelatedPageName = p_RelatedPageName
End Property
Public Property Let RelatedPageName(ByVal RelatedPageName As String)
    p_RelatedPageName = RelatedPageName
End Property
Public Property Get CanBeUnloaded() As Boolean
    CanBeUnloaded = p_CanBeUnloaded
End Property
Public Property Let CanBeUnloaded(ByVal CanBeUnloaded As Boolean)
    p_CanBeUnloaded = CanBeUnloaded
End Property
Public Property Get SubFormPageName() As String
    SubFormPageName = p_SubFormPageName
End Property

Public Property Let SubFormPageName(ByVal SubFormPageName As String)
    p_SubFormPageName = SubFormPageName
End Property
```

Creating an Object with *New* and *Set*

To test your new class, you create a module (not a class module) to verify that you can create an object. If you insert a breakpoint and trace through the code execution, you will learn a great deal, as you can trace through the codes execution into the class module code.

You can define the object variable and then later create an object with the *New* keyword, or as is also shown demonstrated in the following code, with the *aTab2* object, you can both define and create the object at the same time. It is largely a matter of personal preference as to which method you choose to use.

Once you have finished with the object, set the object variable to *Nothing*; this destroys the object. The object would be destroyed anyhow when the code stops execution, but explicitly tidying up your objects is good practice and becomes more important when you work with more complex objects:

```
Sub modTabs_TestObject()
    ' test creating an object
    Dim aTab As clsTabPage
    Set aTab = New clsTabPage
    aTab.PageName = "ProductList"
    aTab.RelatedPageName = "Product Details"
    aTab.SubFormPageName = "frmTabsDynamicProductList"
    aTab.CanBeUnloaded = False

    Debug.Print aTab.PageName
    Set aTab = Nothing

    Dim aTab2 As New clsTabPage
    aTab2.PageName = "Product Details"
    Debug.Print aTab2.PageName
    Set aTab2 = Nothing
End Sub
```


INSIDE OUT *Initialization* and *Termination* Events

When you are in a class module, you can select Class from the upper-left drop-down menu, which normally shows (General). Select Initialize or Terminate from the drop-down list that appears, and then generate the following procedures (in this example the type name *ObjectType* is not a real type but could for example be replaced with a real object type such as a *DAO.RecordSet* object):

```
Private Sub Class_Initialize()
    Set p_Object = New ObjectType
End Sub
Private Sub Class_Terminate()
    Set p_Object = Nothing
End Sub
```

Because class objects can contain other class objects or built-in class objects such as a *Recordset*, you might need to use the *New* keyword in *Initialize* to create an object that is assigned to a private variable, and then set the objects to *Nothing* to close the objects in the *Terminate* procedure. Externally, when your class object is created, the *Initialize* procedure is executed, and when it is set to *Nothing* or the variable goes out of scope, the *Terminate* procedure is executed.

Collection of Objects

A VBA collection is a set of objects that you can use in a similar manner as the built-in collections, such as the *Forms* collection that you worked with in earlier chapters.

The example that follows defines a collection that is used to hold our Tab page objects:

```
Sub modTabs_Collection()
    ' test creating an object
    Dim TabPages As New Collection
    Dim aTab As clsTabPage
    Set aTab = New clsTabPage
    aTab.PageName = "ProductList"
    aTab.RelatedPageName = "Product Details"
    aTab.SubFormPageName = "frmTabsDynamicProductList"
    aTab.CanBeUnloaded = False
    TabPages.Add aTab, aTab.PageName
    Set aTab = Nothing

    Set aTab = New clsTabPage
    aTab.PageName = "Product Details"
    aTab.RelatedPageName = ""
    aTab.SubFormPageName = "frmTabsDynamicProductDetails"
    aTab.CanBeUnloaded = True
    TabPages.Add aTab, aTab.PageName
    Set aTab = Nothing

    For Each aTab In TabPages
        Debug.Print aTab.PageName, aTab.SubFormPageName, _
aTab.RelatedPageName, aTab.CanBeUnloaded
    Next
    Debug.Print TabPages.Count

    Stop
    Set aTab = TabPages("ProductList")
    Debug.Print aTab.PageName
    Debug.Print TabPages("Product Details").PageName
    ' note 1 based collection unlike built in collections
    Debug.Print TabPages(1).PageName
    Set TabPages = Nothing
    Set aTab = Nothing
End Sub
```

Notice how the *aTab* variable is used several times to create objects, and how setting it to *Nothing* does *not* destroy the object. This is because once you have created an object, you add it to the collection, which is then responsible for managing the object (when the collection is set to *Nothing*, it will destroy the objects it contains.

When you add an object to a collection, you must also specify a collection key value (which must be unique). Doing this means that rather than referring to a collection object as

TabPages(1), you can use the key and refer to this as TabPages("Product List"). The *Collection* object's *Add* method also allows you to specify an optional *Before* or *After* argument for positioning an object relative to other objects in the collection. The collections first element is 1 and not 0 (which is what the built-in Access collections use).

Be aware that when you refer to an object by using TabPages(1).PageName, you cannot take advantage of IntelliSense assistance. This is because this type of collection can hold different types of objects, so the environment cannot know exactly which properties would apply to an object.

INSIDE OUT

VBA collection classes

The built-in VBA collection classes that you have been working with are different from an Access collection. The first difference is that the Access collections, such as *TableDefs*, can only hold one type of object; a VBA collection can hold different types of objects (this explains why the IntelliSense is limited). The second difference is that VBA collection classes are 1-based, whereas the Access collections are 0-based.

In the next section, you will be creating your own collection classes that wrap around the VBA collection class. These collections will start to look more like an Access collection.

Once you have added an object to a collection and specified the key value, you will find that you cannot subsequently display the key value—it is hidden. If your procedures need to be able to refer to the key, you might find it useful to add your own property to the object class, which saves and holds the key value in each object. Looking in the class *clsTabPage*, you see the following (it is not essential to do this in the class):

```
Dim p_Key As String
Public Property Get Key() As String
    Key = p_Key
End Property
Public Property Let PageName(ByVal PageName As String)
    p_PageName = PageName
    p_Key = PageName
End Property
```

Creating Collection Classes

A VBA *Collection* object supports a limited number of operations—*Add*, *Count*, and *Remove*. You will likely want to be able to add more operations to your collection. To do that, you need to define your own collection class, called *clsTabPageCollection*.

Defining a collection class follows the same steps as defining a normal class to create the class module. Your collection class will contain a VBA collection, so you define an internal variable called *p_TabPages*. As we previously described, classes can have two specially named methods for initializing and terminating the class. The simple *clsTabPage* didn't need any special operations, but the new class needs to create a VBA collection, and then remove all the objects from the collection when it is terminated, as illustrated in the following code:

```
Private p_TabPages As Collection

Private Sub Class_Initialize()
    Set p_TabPages = New Collection
End Sub

Private Sub Class_Terminate()
    Dim aClassPage As clsTabPage
    For Each aClassPage In p_TabPages
        p_TabPages.Remove CStr(aClassPage.PageName)
    Next
    Set p_TabPages = Nothing
End Sub
```

You also want to have the standard operations for counting, adding, and removing items from the class, so you need to add these methods to our collection (you also add an *Item* method, which is another standard feature of a class):

```
Public Property Get Count() As Long
    Count = p_TabPages.Count
End Property

Public Sub Add(aClassPage As clsTabPage)
    p_TabPages.Add aClassPage, aClassPage.PageName
End Sub

Public Sub Remove(PageName As Variant)
    p_TabPages.Remove CStr(PageName)
End Sub

Public Function Item(PageName As Variant) As clsTabPage
    Set Item = p_TabPages(PageName)
End Function
```

Once you start defining your own collection class, you will find that a number of the expected built-in collection class features no longer work. For example, you cannot use a *For Each* loop, or index the collection by using the friendly key name (you will see how to

get around this). The following procedure can be used to test the class; the program lines that are commented out have been included to show what will not work in our collection class:

```
Sub modTabs_clsTabPageCollection()
    ' test creating an object
    Dim TabPages As New clsTabPageCollection
    Dim aTab As clsTabPage
    Dim lngCount As Long
    Set aTab = New clsTabPage
    aTab.PageName = "ProductList"
    aTab.RelatedPageName = "Product Details"
    aTab.SubFormPageName = "frmTabsDynamicProductList"
    aTab.CanBeUnloaded = False
    TabPages.Add aTab
    Set aTab = Nothing

    Set aTab = New clsTabPage
    aTab.PageName = "Product Details"
    aTab.RelatedPageName = ""
    aTab.SubFormPageName = "frmTabsDynamicProductDetails"
    aTab.CanBeUnloaded = True
    TabPages.Add aTab
    Set aTab = Nothing

'    For Each aTab In TabPages
'        Debug.Print aTab.PageName, aTab.SubFormPageName, _
'                aTab.RelatedPageName, aTab.CanBeUnloaded
'    Next
    For lngCount = 1 To TabPages.Count
        Set aTab = TabPages.Item(lngCount)
        Debug.Print aTab.PageName, aTab.SubFormPageName, _
                aTab.RelatedPageName, aTab.CanBeUnloaded
    Next
    Set aTab = Nothing
'    Set aTab = TabPages("ProductList")

    ' following will work
    Set aTab = TabPages.Item(1)
    Debug.Print TabPages.Item(1).PageName
    Debug.Print aTab.PageName
    Set aTab = Nothing
    Set TabPages = Nothing
End Sub
```

There are two techniques available to get around the problem of not being able to refer to the collection class by using the key names. The first technique involves adding an *AllItems* function to the collection class, and the second method involves exporting, editing, and re-importing the class.

INSIDE OUT

Adding *AllItems* to a collection class

When you use the *AllItems* method, you need to add the following property to the class (you can give this property an alternative name):

```
Public Function AllItems() As Collection
    Set AllItems = p_TabPages
End Function
```

In the sample testing file, modTabs_clsTabPageCollection2, you can see how to use this feature. The important code is as follows:

```
' works with allitems
For Each aTab In TabPages.AllItems
    Debug.Print aTab.PageName, aTab.SubFormPageName, _
        aTab.RelatedPageName, aTab.CanBeUnloaded
Next
Set aTab = TabPages.AllItems("ProductList")
Debug.Print aTab.PageName
Debug.Print TabPages.AllItems("ProductList").PageName
```

This is a satisfactory solution as long as you are prepared to insert the *.AllItems* reference when using the collection with the object's key.

Exporting and Re-importing the Class

The reason that you cannot refer to collections by using standard syntax is because VBA classes do not allow special attributes to be set on a class, and these are required to support standard syntax.

If you right-click the collection class module in the project window, export it to a text file, and then open the text file in notepad, you will see the following header information in the class:

```
VERSION 1.0 CLASS
BEGIN
  MultiUse = -1  'True
END
Attribute VB_Name = "clsTabPageCollection"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = False
Attribute VB_Exposed = False
Option Compare Database
Option Explicit
' class clsTabPagesCollection
Private p_TabPages As Collection
```

These attributes are not exposed in the VBA environment. There is a special attribute value, which when set to 0, sets the member as the default member for the object. You want the *Item* method to be the default member and you need to change the method adding the following attribute definition (this will enable references such as *TabPages("ProductList")* to work). Also, to support enumeration in a *For … Each* loop, you need to add the *NewEnum* method, as shown in the following:

```
Public Function Item(ByVal Index As Variant) As clsTabPage
Attribute Item.VB_UserMemId = 0
         Set Item = p_TabPages(Index)
End Function

Public Function NewEnum() As IUnknown
Attribute NewEnum.VB_UserMemId = -4
         Set NewEnum = p_TabPages.[_NewEnum]
End Function
```

After saving these changes, import the class back into your project, as shown in Figure 9-3.

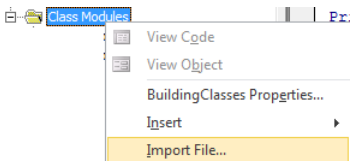

Figure 9-3 Re-importing a class back into Access.

If you look in the VBA Editor, you will not be able to see the new attribute you just added in the *Item* method because it remains hidden.

This then means that the following references will work (note that in the sample database BuildingClassesAfterExportImport.accdb, the following code will work, because we have performed this rather complex operation; in the sample database BuildingClasses.accdb, this code has been commented out because it will not work):

```
For Each aTab In TabPages
    Debug.Print aTab.PageName, aTab.SubFormPageName, _
        aTab.RelatedPageName, aTab.CanBeUnloaded
Next
Set aTab = TabPages("ProductList")
Debug.Print TabPages("ProductList").PageName
Debug.Print aTab.PageName
```

This process needs to be repeated for each collection class in your project.

Using Classes with the Dynamic Tab

You are now able to modify the code in the *frmTabsDynamic* form to make use of your new classes.

At the top of the module, where you had defined an array of types, declare your collection class as shown here:

```
Option Compare Database
Option Explicit
Dim TabPages As clsTabPageCollection
Dim lngTabPages As Long
```

The form's *Open* and *Close* events then create and dispose of the collection, as shown in the following:

```
Private Sub Form_Close()
    Set TabPages = Nothing
End Sub

Private Sub Form_Open(Cancel As Integer)
    Set TabPages = New clsTabPageCollection
    LoadTabs
End Sub
```

In the following code, in the *LoadTabs* procedure, you create and load your class objects into the collection:

```
    Do While Not rst.EOF
        Set aTabPage = New clsTabPage
        aTabPage.PageName = rst!PageName
        aTabPage.SubFormPageName = rst!SubFormName
        aTabPage.CanBeUnloaded = rst!CanUnloadPage
        aTabPage.RelatedPageName = Nz(rst!RelatedPage)
        TabPages.Add aTabPage
        Set aTabPage = Nothing
        If rst!DefaultVisible And lngPageVisibleCount + 1 < lngTabPages Then
            LoadThePage aTabPage, lngPageVisibleCount
            lngPageVisibleCount = lngPageVisibleCount + 1
        End If

        lngArray = lngArray + 1
        rst.MoveNext
    Loop
```

There are some other minor references in the code that used the array of types that now need to be changed to use the new collection and objects.

Simplifying the Application with Classes

In the preceding sections, you have been able to change your dynamic tab to use classes, but it has not as yet resulted in any simplification of the applications code. In fact, you now have more code to maintain than when you started. But you now have a framework in which you can start to work that will lead to simplification and improved maintenance of your code.

In examining the *frmTabsDynamic* form, you can see that it has a general routine *LoadTabs* that involves reading information and placing the information into your collection. This operation could be placed inside the collection. So we can start to enhance our collection (*clsTabPageCollection2*) by adding the data loading function. But the process of loading the information also involves setting values in controls on the form. This means you also want to allow the collection to reference the controls on the form.

To begin, add new private members to the class:

```
' class clsTabPagesCollection
Private p_TabPages As Collection
Private p_TabControl As TabControl
Private p_Controls As Controls
```

You must change the termination routine to clear the new variables and provide properties for setting the new variables, as follows:

```
Private Sub Class_Terminate()
    Dim aClassPage As clsTabPage
    For Each aClassPage In p_TabPages
        p_TabPages.Remove CStr(aClassPage.PageName)
    Next
    Set p_TabPages = Nothing
    Set p_TabControl = Nothing
End Sub
Public Property Let TabControl(ByRef TabCtl As TabControl)
    Set p_TabControl = TabCtl
End Property
Public Property Let Controls(ByRef Ctrls As Controls)
    Set p_Controls = Ctrls
End Property
```

You can then move the appropriate routines programmed into the form into the collection class.

> **Note**
>
> **The full code for this can be seen in the sample file.**

The result of this is an impressive reduction in the code on the form, which now shrinks to the following (see *frmTabsDynamic2*):

```
Option Compare Database
Option Explicit
Dim TabPages As clsTabPageCollection2
Private Sub Form_Close()
    Set TabPages = Nothing
End Sub
Private Sub Form_Open(Cancel As Integer)
    Set TabPages = New clsTabPageCollection2
    TabPages.TabControl = Me.TabCtl0
    TabPages.Controls = Me.Controls
    TabPages.LoadFromTable Me.Name, "tblTabPages"
End Sub
Private Sub TabCtl0_DblClick(Cancel As Integer)
    TabPages.TabPageDoubleClick CLng(Me.TabCtl0)
End Sub
```

Although the total amount of code remains unchanged, much of the code has moved out of the form and into the classes. There are a couple of advantages to creating classes to perform these operations:

- The code on the form is significantly simplified; it will be easy to add it to other forms or in other applications.
- The new classes are easy and intuitive to work with, so using them in the future should improve your applications, and you can add more features to these classes.

Some might argue that rather than using classes, which involves constructing a framework, you could more simply have built a re-useable library. This line of argument nearly always holds; thus, the decision to use classes becomes a question of whether it seems more intuitive and natural than using a traditional code module.

INSIDE OUT

Classes and associated terminology

Another term for creating an object is *instantiating* the class object. This means using the *New* keyword to create the class object.

The term *Encapsulation* is often used to convey the idea of tucking away all the functionality inside the class, such that the class only exposes as small a public interface as required to fulfill its purpose. With a class, you are wrapping up all the messy code and placing that inside a box so that you don't need to deal with it on a regular basis.

Creating a Hierarchy of Classes

In this example, you look at creating a hierarchy of classes, which demonstrates the ability of classes to be used as building blocks for improving the design in managing data objects. The example involves a business problem for which the classes need to perform complex calculations (although you will stick to simple calculations in the example).

Suppose that you have analyzed an insurance company's business, the result of which revealed that the company sells a large number of different insurance products, but you noticed that there are common features in the products. Often, one type of policy only differs from another in a small number of ways. The task is to build an Access application that assists with generating the policy documents and performing appropriate calculations for the different policies.

Creating a Base Class

The first task is to identify common features to all policies as well as the most standard calculations that a policy would require to perform. This involves creating a class, which will serve as the base class. In the following code, this is called *clsPolicy*.

From the project window in the VBA Editor, create a class module, and then save the module with the name *clsPolicy*, as demonstrated in the following code:

```
Option Compare Database
Option Explicit

' clsPolicy is the base class which has common features
' required in other classes

Dim p_MonthlyPremium As Currency

Public Property Get MonthlyPremium() As Currency
    MonthlyPremium = p_MonthlyPremium
End Property

Public Property Let MonthlyPremium(ByVal MonthlyPremium As Currency)
    p_MonthlyPremium = MonthlyPremium
End Property

Public Function CalculateAnnualPolicyValue() As Currency
    CalculateAnnualPolicyValue = p_MonthlyPremium * 12
End Function
```

This class can then be tested by using the following code:

```
Sub modInsurance_Policy()
    ' create a Policy from clsPolicy
    Dim Policy As New clsPolicy
    Policy.MonthlyPremium = 10
    ' Expect 120
    Debug.Print Policy.CalculateAnnualPolicyValue()
    Set Policy = Nothing
End Sub
```

Derived Classes

With the basic insurance policy class created, you can now create several other classes that will all use some of the base class features. This involves creating a class, which will serve as the derived class, and in the following code is called *clsHomePolicy*, being derived from the base class *clsPolicy*. The term derived is used because the class is in some way related or derived from the base class:

```
Option Compare Database
Option Explicit

' clsHomePolicy uses clsPolicy
Dim p_Policy As clsPolicy

Private Sub Class_Initialize()
    Set p_Policy = New clsPolicy
End Sub
Private Sub Class_Terminate()
    Set p_Policy = Nothing
End Sub

Public Property Get MonthlyPremium() As Currency
    MonthlyPremium = p_Policy.MonthlyPremium
End Property

Public Property Let MonthlyPremium(ByVal MonthlyPremium As Currency)
    p_Policy.MonthlyPremium = MonthlyPremium
End Property

Public Function CalculateAnnualPolicyValue() As Currency
    CalculateAnnualPolicyValue = p_Policy.CalculateAnnualPolicyValue() + 50
End Function
```

The first derived class, *clsHomePolicy*, contains a base class object, *clsPolicy*, so you need to have initialization and termination events to create and dispose of the base class object.

The *clsHomePolicy* is only loosely tied to *clsPolicy*, which means that you need to add all the required properties and methods into the new class. But if you look at the *CalculateAnnual PolicyValue* method, you will see how it can take advantage of the calculation in the base class.

INSIDE OUT

Inheritance and polymorphism in classes

Note that we are using the term *derived* here in a very loose manner. Many OOP languages incorporate the concept of *inheritance*, which means truly deriving classes, and they use the term *polymorphism* for how derived classes can implement variations on methods available through base classes.

VBA does *not* support direct inheritance or explicit polymorphism, but you can use the approach described here to create structures that offer some of these characteristics.

Another OOP term is *multiple inheritance*, which means inheriting from more than one base class; by embedding other classes using this technique, we can also form structures that behave in some respects like those having multiple inheritance. The techniques used here to produce a hierarchy can also be described by the term *wrapper*, where we wrap around one class for the purpose of extending or changing its functionality.

As is illustrated in the code that follows, you can now define two additional classes, one called *clsSpecialHomePolicy*, which is derived from *clsHomePolicy*, and the other, called *clsCarPolicy*, is derived from *clsPolicy* (you can view the code in the sample database):

```
Option Compare Database
Option Explicit

' clsSpecialHomePolicy
Dim p_Policy As clsHomePolicy

Private Sub Class_Initialize()
    Set p_Policy = New clsHomePolicy
End Sub
Private Sub Class_Terminate()
    Set p_Policy = Nothing
End Sub

Public Property Get MonthlyPremium() As Currency
    MonthlyPremium = p_Policy.MonthlyPremium
End Property

Public Property Let MonthlyPremium(ByVal MonthlyPremium As Currency)
    p_Policy.MonthlyPremium = MonthlyPremium
End Property

Public Function CalculateAnnualPolicyValue() As Currency
    CalculateAnnualPolicyValue = p_Policy.CalculateAnnualPolicyValue() + 100
End Function
```

These classes can be tested with the following code:

```
Sub modInsurance_Policy()
    ' create a Policy from clsPolicy
    Dim Policy As New clsPolicy
    Policy.MonthlyPremium = 10
    ' Expect 120
    Debug.Print Policy.CalculateAnnualPolicyValue()
    Set Policy = Nothing

    ' create a HomePolicy
    Dim HomePolicy As New clsHomePolicy
    HomePolicy.MonthlyPremium = 10
    ' Expect 120+50 = 170
    Debug.Print HomePolicy.CalculateAnnualPolicyValue()
    Set HomePolicy = Nothing

    ' create a SpecialHomePolicy
    Dim SpecialHomePolicy As New clsSpecialHomePolicy
    SpecialHomePolicy.MonthlyPremium = 10
    ' Expect 120+50+100 = 270
    Debug.Print SpecialHomePolicy.CalculateAnnualPolicyValue()
    Set SpecialHomePolicy = Nothing

    ' create a CarPolicy
    Dim CarPolicy As New clsCarPolicy
    CarPolicy.MonthlyPremium = 10
    ' Expect 120+80 = 200
    Debug.Print CarPolicy.CalculateAnnualPolicyValue()
    Set CarPolicy = Nothing
End Sub
```


Summary

In this chapter, you learned about classes via two examples. In the first example, you saw how a general purpose framework for working with form *Tab* controls can dynamically load subforms and be re-written using classes. The final result was simplified application code with the complexity hidden within the class.

The second example introduced techniques for building a hierarchy of classes by using a base class and several derived classes. This provides a more structured and maintainable solution when using classes.

Index

Symbols

A

B

C

D

E

F

M

N

O

P

Q

R

T

U

V

W

About the Author

Andrew Couch has been working with Microsoft Access since 1992, developing, training, and consulting on Client-Server design projects. With his wealth of experience in Access and SQL products, he has been able to mentor software houses, blue chip companies, and independent developers. Alongside running his own consultancy, Andrew has been heavily involved in the developer community and jointly founded the UK Access User Group more than 13 years ago. He has also earned Access MVP status for the last 5 years.

Andrew's passion lies with VBA programming and extending the reach of VBA programmers into cloud computing and the .NET environment. He hopes that this book serves as an example of his dedication to this exceptional piece of technology and its application.

In addition to consulting and regularly speaking at community events, Andrew has developed the Migration Upsizing SQL Tool (MUST), which is a tool that allows users to easily convert Access Databases to SQL Server by using an Access-based application. Due to the success of MUST, which is used by over 150 companies, SQL Translation capabilities and WebForm code generators for .NET were added to the product range. More recently the MUST technologies have been extended further to deliver automated services for converting Access database to a web legal format for publishing to SharePoint.