

Word 2003
Visual
Basic
Programming

John Low

Copyright © 2005 by Lulu Publishing, Inc.

3131 RDU Center, Suite 210

Morrisville, NC 27560

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher or author.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

LULU PUBLISHING

3131 RDU Center, Suite 210, Morrisville, NC 27560

A Print On-Demand Solution Provider

Please visit our website at www.lulu.com

Contents

Chapter 1 Introduction	1
WHY LEARN VISUAL BASIC?	1
VISUAL BASIC EDITOR	1
<i>Start Visual Basic from Word</i>	1
<i>Visual Basic Editor</i>	2
<i>Visual Basic Menu and Toolbars</i>	2
<i>Project Explorer Window</i>	3
<i>Visual Basic Help</i>	3
<i>Properties Window</i>	4
YOUR FIRST VISUAL BASIC PROGRAM	5
<i>Macro</i> 5	
<i>Add a Visual Basic Module</i>	5
<i>Module Code Window</i>	6
<i>Name of Your Module is myModule</i>	6
<i>Add a Procedure</i>	7
<i>Add Procedure Window</i>	8
<i>Your helloWorld Macro</i>	9
<i>Completing Your Code</i>	9
<i>Visual Basic Functions</i>	10
<i>Run Your Macro</i>	11
DEBUGGING	12
<i>Compile Error</i>	12
<i>Runtime Error</i>	13
Chapter 2 Object Programming	15
WHAT IS AN OBJECT?	15
<i>Object and Class</i>	15
<i>Object Contain Other Objects</i>	15
<i>Object's Property Example</i>	16
<i>Object's Method Example</i>	17
<i>What is a Collection Object?</i>	18
OBJECT VARIABLE	19
<i>Using Objects Directly</i>	19
<i>Declaring Object Variables</i>	19
<i>Declaring Object Variables without Assigning Object References</i>	19
<i>Declaring Integer Variables without Assigning Values</i>	20
<i>Run-Time Error and Good Programming Practice</i>	21
<i>Assigning Object Reference to an Object Variable</i>	21

<i>Assign Object Variable Example</i>	22
WORKING WITH PROPERTIES AND METHODS.....	23
<i>Assign Values to Object's Properties</i>	23
<i>Assign Value to Object's Property Example</i>	24
<i>Set Statement</i>	25
<i>Set Statement Example – Property Object</i>	26
<i>Set Statement Example – Method Object</i>	28
<i>Set Statement Returns Object Reference</i>	29
<i>Set Statement Example – Comparison</i>	30
SUB PROCEDURE AND FUNCTION METHODS.....	31
<i>Sub Procedure Method Example</i>	32
<i>Function that Returns a Value</i>	33
Chapter 3 More Object Programming.....	35
WORD OBJECT MODEL.....	35
<i>Where Is My Object?</i>	35
<i>Object Browser</i>	36
<i>Icons Used in Object Browser</i>	37
<i>Default Property</i>	37
<i>Default Method</i>	38
<i>Default Property and Method Example</i>	39
<i>Event and Event Procedure</i>	41
<i>Event Procedure Example</i>	41
<i>What is an Enum Class?</i>	43
NAVIGATING THE OBJECT HIERARCHY USING OBJECT BROWSER.....	44
<i>ActiveDocument Object</i>	44
<i>Document and Paragraphs Objects</i>	45
<i>Add Method</i>	46
<i>Item Method</i>	48
<i>Range Property</i>	48
<i>Text Property</i>	49
<i>Recap</i> 50	
Chapter 4 Programming Fundamentals.....	51
SOME PROGRAMMING HOUSEKEEPING.....	51
<i>Option Explicit Statement</i>	51
<i>Continuation of Statement</i>	52
<i>Comment Line</i>	52
VARIABLES.....	53
<i>Variable Name</i>	53
VARIABLES THAT USE NUMBERS.....	54
<i>Numeric Variable Example</i>	55

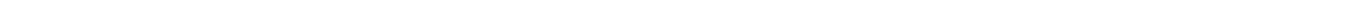
VARIABLES OF THE BOOLEAN DATA TYPE	56
<i>Comparison Operators</i>	57
<i>Comparison Operator Example</i>	57
<i>Logical Operators</i>	59
<i>Logical Operator Example</i>	60
VARIABLES THAT USE STRINGS	61
<i>Concatenate Operator &</i>	62
<i>String Data Type Example</i>	62
VARIABLES THAT USE DATES.....	64
<i>Date Data Type Example</i>	64
CONDITIONAL STATEMENT	66
<i>If...Then...Else Statement (Single-Line Form)</i>	66
<i>If...Then...Else Example 1</i>	66
<i>If...Then...Else (Block Form) Statement</i>	68
<i>If...Then...Else Example 2</i>	68
<i>If...Then...Else Example 3</i>	69
Chapter 5 Programming Loop & Other	73
WHY USE LOOP?.....	73
<i>For...Next Statement</i>	73
<i>For...Next Loop Example</i>	74
<i>For Each...Next Statement</i>	75
<i>For...Each...Next Loop Example 2</i>	78
<i>Do...Loop Statement</i>	80
<i>Use of the Do...Loop Statement</i>	81
WITH STATEMENT.....	81
FUNCTION PROCEDURE	82
WHAT IS AN ARRAY?	83
<i>Table Programming with Array Example</i>	84
DEBUGGING.....	90
<i>Debugging and Error Handling</i>	90
<i>Trace Execution of Your Program</i>	91
<i>Step Into, Step Over and Step Out</i>	94
Chapter 6 Database and Mail Merge Programming	95
CREATE DATABASE.....	95
<i>Create Table</i>	96
<i>Put Data in Your Table</i>	97
DATABASE PROGRAMMING	98
<i>DAO Object Library</i>	98
<i>Help and Documentation on DAO Objects</i>	99

DATABASE PROGRAMMING EXAMPLE	99
<i>Database Object</i>	99
<i>OpenDatabase Method</i>	100
<i>Workspace Object</i>	100
<i>Recordset Object</i>	100
<i>OpenRecordset Method</i>	101
<i>What is SQL?</i>	101
<i>Get Number of Records</i>	101
<i>Setup Table</i>	103
<i>Display Data from Database</i>	104
MAIL MERGE PROGRAMMING	109
<i>MailMerge Object</i>	110
<i>Selection Object</i>	110
<i>Execute the Mail Merge</i>	112
Chapter 7 File Programming	117
FILESYSTEMOBJECT OBJECT	117
CREATE NEW TEXT FILE	117
<i>FileSystemObject Object</i>	117
<i>TextStream Object</i>	118
LIST DIRECTORY AND FILES	119
<i>Folder Object</i>	119
PROCESSING MULTIPLE WORD DOCUMENTS	123
<i>Create New Folder for Output Files</i>	123
<i>Create Log File</i>	123
<i>Processing Multiple Word Documents</i>	124
Chapter 8 Form Programming	129
WHAT IS A FORM?	129
<i>Form and Controls</i>	130
FORM PROGRAMMING EXAMPLE – INSERT A NEW TABLE	130
<i>UserForm Object</i>	130
<i>Toolbox and Controls</i>	131
<i>Change Name of UserForm Form and Caption Property</i>	132
<i>Add Controls to UserForm Form</i>	133
<i>Change Name and Caption Property of Control</i>	135
<i>Show Method of UserForm Object</i>	137
<i>Initialize Event of UserForm Object</i>	138
<i>Click Event of CommandButton Control, cmdOK</i>	140
<i>Click Event of CommandButton Control, cmdCancel</i>	142
<i>Run Macro</i>	143
FORM PROGRAMMING EXAMPLE – BUILD TABLE OF INDEX	145

<i>Show Method of UserForm Object</i>	147
<i>Click Event of CommandButton Control, cmdFind</i>	147
<i>Click Event of CommandButton Control, cmdExit</i>	152
<i>Run Macro</i>	153
Index	157

About the Author

John Low is a software developer living in Norwalk, California. John received his first degree in mathematics, then a master's degree in computer science and a master's degree in civil engineering. He can be reached at johnlow5002@yahoo.com.



Who Should Read This Book

This book is for programmers and would-be programmers who want to learn Word 2003 Visual Basic programming as quickly as possible. I assume that you are fluent in using Word 2003 as a word processor and want to learn the Visual Basic programming part of Word 2003 to help you become more efficient with your word processing task. It is preferable that you have some programming background but the concepts are explained at the elementary level so a beginning programmer should be able to follow and learn.

I firmly believe in learning programming through examples so this book is full of programming examples to explain the concepts and technique of programming Visual Basic. I expect you to try all the examples in this book and make the code work for you. This is how you learn programming. Learning programming by doing is fun and I want you to have as much fun as I have writing this book!

Chapter 1 Introduction

Why Learn Visual Basic?

Why learn Visual Basic? Why do you have to resort to programming in a computer language that you may not be familiar with inside a word processor? In short to save time. What is inside a document is just another form of data, so it is natural that you write computer programs to manipulate the content (that is, the data) of your document. Very few people who use Word ever utilize the macro or the Visual Basic programming features. As a result they can end up spending many hours modifying documents manually that could have been done in a few minutes (or seconds) with a macro. Using Word without using Visual Basic is like using a hammer to build a house without using power tools. Being able to quickly write a program that saves hours of labor and automates a task is a great advantage.

Visual Basic Editor

Start Visual Basic from Word

To start Visual Basic from Word, follow these steps.

Steps

- 1) Start a new Word document, and call it Lesson1.
- 2) From the Word menu select Tools / Macro / Visual Basic Editor (Figure 1.1).

Figure 1.1 Start Visual Basic from Word



Visual Basic Editor

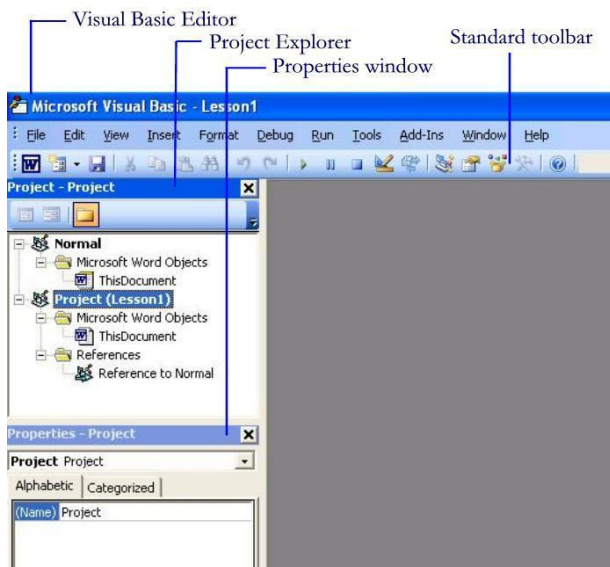
The Visual Basic Editor window appears (Figure 1.2). The Visual Basic Editor window is where you create, edit, debug, and run Visual Basic code associated with your Word documents. To the left of the Visual Basic Editor window there are two sub-windows:

- 1) Project Explorer window;
- 2) Properties window.

The Project Explorer window displays the different projects associated with the document. A project is a collection of files that contain your Visual Basic codes or programs.

The Properties window displays properties of the files or objects you've selected in the Project Explorer window.

Figure 1.2 *Visual Basic Editor*



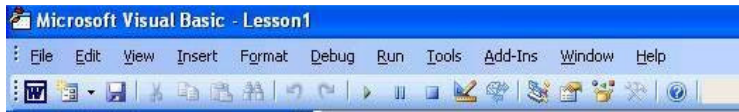
Visual Basic Menu and Toolbars

Figure 1.3 shows the Visual Basic Menu and Toolbars. The Visual Basic Menu displays the commands you use to work with Visual Basic. Besides the standard File, Edit, View, Window, and Help menus, menus are provided to access functions specific to programming such as Insert, Format, or Debug.

The Visual Basic Toolbar provides quick access to commonly used commands in the programming environment. You click a button on the toolbar once to carry out the action represented by that button. By default, the standard toolbar is displayed when you start the Visual Basic Editor. Additional toolbars such as debugging and other tasks can be toggled on or off from the Toolbars command on the View menu.

Toolbars can be docked beneath the menu bar or can float if you select the vertical bar on the left edge and drag it away from the menu bar.

Figure 1.3 *Visual Basic menu and toolbars*



Project Explorer Window

Figure 1.4 shows the Project Explorer window. You use the Project Explorer window to view, modify, and navigate the projects for every open document or template. You can resize the Project Explorer window and either dock it to or undock it from any of the sides of the Visual Basic Editor window to make it easier to use. All the Visual Basic code associated with a document or template is stored in a project that is automatically stored and saved with the document or template.

Figure 1.4 *Project Explorer window*



Note

In Word, because the Normal template is available from every Word document, there's always a project for Normal in the Project Explorer.

Visual Basic Help

Select the Project Explorer window by clicking on any part of empty space in the window. Press the F1 key and the Visual Basic Help for Project Explorer appears (Figure 1.5).

For information and help about a particular window in the Visual Basic Editor, click in the window and then press F1 to open the appropriate Help topic. To see the Help topic for any other element of the Visual Basic Editor, such as a particular toolbar button, search Help for the name of the element.

Figure 1.5 *Visual Basic Help*

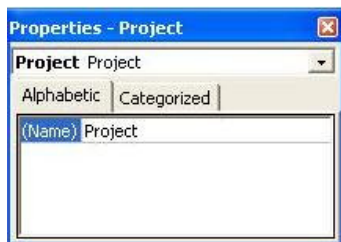


Properties Window

Figure 1.6 shows the Properties window for the selected project. You use the Properties window to set the properties of an object at design time. Objects are fundamental building blocks to all of the Visual Basic programming environments including Word, Excel, Access and Visual Basic.Net. Objects contain both code (program logic) and data, making them easier to maintain than traditional ways of writing code. Properties are data that describe an object. Methods are actions you tell the object to do. More on the topics of Objects in Chapter 2 and 3.

If you don't think you'll be using the Properties window right now, you can close it to simplify your work space a little. You can open it again at any time by clicking Properties Window on the View menu.

Figure 1.6 *Properties window*



Your First Visual Basic Program

Macro

You are ready to write your first macro. A macro is a Visual Basic program that contains a sequence of instructions (codes) that tells the computer what to do. You run a macro from the Tools / Macro menu from Word, or from the Run menu of the Visual Basic editor.

Macros are written in the Visual Basic language programming language and stored on special files called modules. When you write a macro you give it a macro name and you can have more than one macro in a module. You can have more than one module associated with your Word document and you give each module a module name.

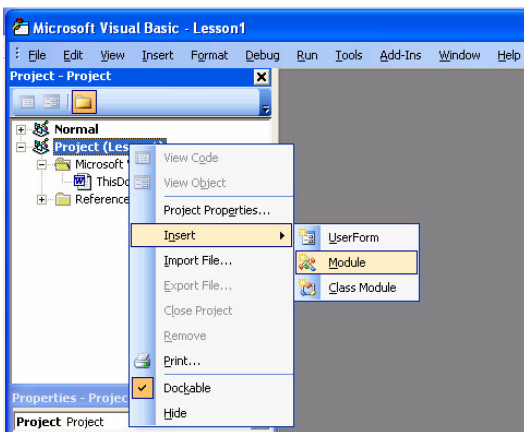
Add a Visual Basic Module

Before you write a macro you have to add (create) a module. Follow these steps to add a module:

Step

- 1) Select Project (Lesson1) in the Project Explorer window and right click on it.
- 2) Select Insert / Module (Figure 1.7).

Figure 1.7 *Add a Visual Basic module*



Another way of adding a module is to select Project (Lesson1) in the Project Explorer window and then select Module from the Insert menu.

Module Code Window

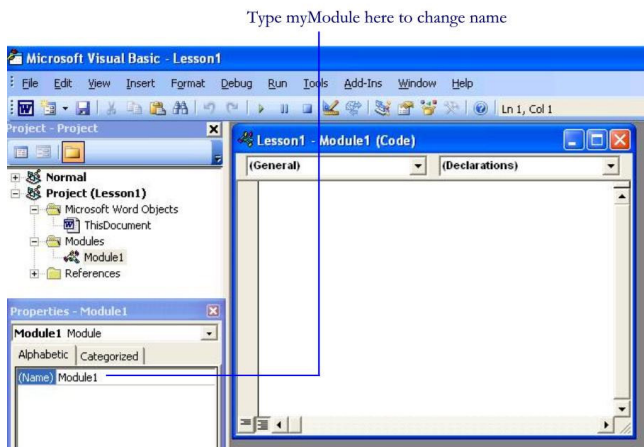
The Module Code window appears (Figure 1.8). You write Visual Basic code for your macro in the Module Code window. Visual Basic Code consists of Visual Basic language statements, constants, and declarations. Using the Code Editor window, you can quickly view and edit any of the code in your macro.

The default name of the Visual Basic module you've added is Module1. To change the default name of the module to a different name follow these steps.

Steps

- 1) Make sure the Properties window of Module1 is visible. If it is not visible, display it by selecting Module1 in the Project Explorer window and from the View menu select Properties Window.
- 2) Change the default name of the module you've added to myModule in the Properties window (Figure 1.8).

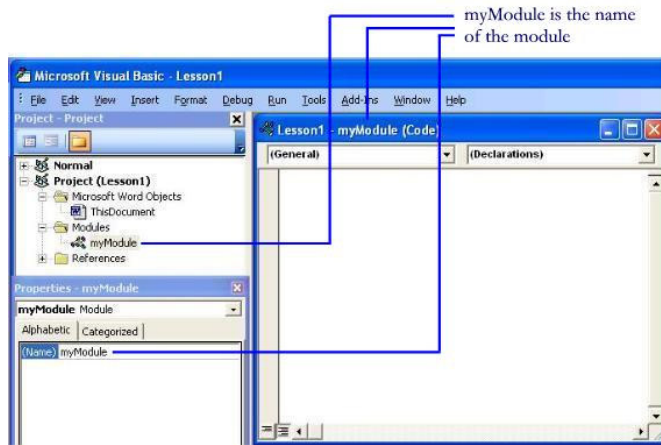
Figure 1.8 *Module Code window*



Name of Your Module is myModule

The name of the module is changed to myModule, which appears in the Project Explorer window, the Properties window and the Code window (Figure 1.9).

Figure 1.9 Name of your module is myModule



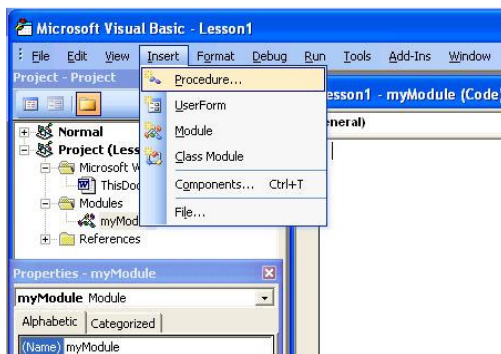
Add a Procedure

Next you will add a Procedure in the myModule module. A **procedure** is a unit of code that instructs the computer to perform a specific operation. A macro is one type of procedure. To add a procedure in the myModule module following these steps.

Steps

- 1) Make sure the myModule (Code) window is visible and in-focus (selected).
- 2) Select Insert/Procedure from the Visual Basic Editor menu (Figure 1.10).

Figure 1.10 Add a procedure



Note

If the Module Code window is not in-focus (selected) you will not be able to select Insert/Procedure from the Visual Basic menu as the Procedure field will be blanked out.

Add Procedure Window

The Add Procedure window appears (Figure 1.11).

There are three types of procedures:

- 1) Sub procedure is a unit of code that performs a task but doesn't return a value. Sub procedure is known as a macro. A Sub procedure is enclosed between the Sub and End Sub statements.
- 2) Function procedure is a unit of code that performs a specific task, like the Sub procedure. Unlike a Sub procedure, however, a function procedure returns a value. A function procedure is enclosed between the Function and End Function statements.
- 3) Property procedure is a unit of code that creates and manipulates custom properties. A property procedure is enclosed between the Property Let and End Property statements.

There are two scopes for Procedure:

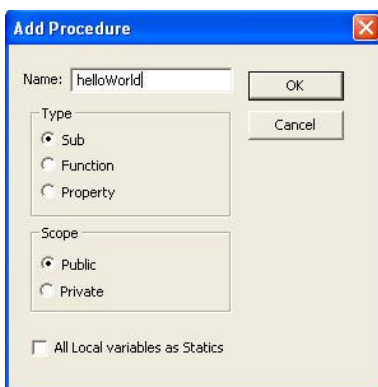
- 1) Public procedure is accessible to any procedure in any module in your application.
- 2) Private procedure is only accessible to other procedures in the same module.

Both Sub procedures and Function procedures can be either public or private.

Steps

- 1) Keep the default type of procedure unchanged as Sub and the default scope of procedure unchanged as Public.
- 2) In the Name box type helloWorld as the name of the procedure you are adding and click the OK button.
- 3) helloWorld will be the name of your macro.

Figure 1.11 *Add Procedure window*



Note

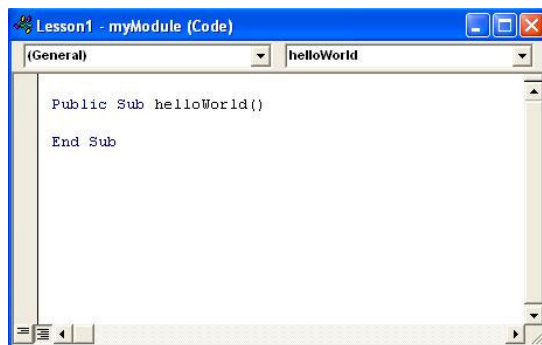
A Sub procedure is a macro.

Your helloWorld Macro

The helloWorld macro is created and the Sub and End Sub statements automatically generated in the Code window (Figure 1.12). The Visual Basic Editor positions the cursor at the beginning of the blank line between the Sub and End Sub statements. Next you are ready to write Visual Basic code for your macro.

Instead of having Visual Basic editor insert the macro (Sub procedure) and generate the Sub and End statements automatically for you, you can type in the same statements manually in the myModule (Code) window.

Figure 1.12 *helloWorld macro*



Completing Your Code

You are going to add the following code to the macro. It uses the Visual Basic function MsgBox to display a dialog box with the (Hello World!) text.

```
MsgBox "Hello World!"
```

MsgBox is one of the many function procedures that Visual Basic provides. You are using the MsgBox function that accepts one parameter, the (HelloWorld!) text. The MsgBox function accepts other optional parameters. For the general syntax and detailed information of the MsgBox function consult Visual Basic Help.

The following shows the completed code for the helloWorld macro.

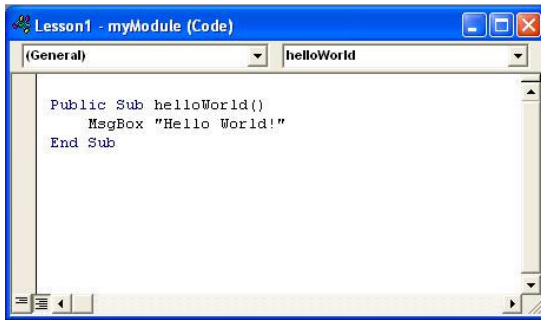
Completed Code

```
Public Sub helloWorld()  
    MsgBox "Hello World!"  
End Sub
```

Steps

- 1) Type in the above the code (Figure 1.13) between the Sub and End Sub statements of the helloWorld macro.
- 2) If you type MsgBox without capitalizing the M and the B, Visual Basic Editor automatically capitalize them for you after you've typed the statement and click anywhere in the code window. Visual Basic does that because it recognizes MsgBox as one of the Visual Basic functions.

Figure 1.13 *helloWorld macro*

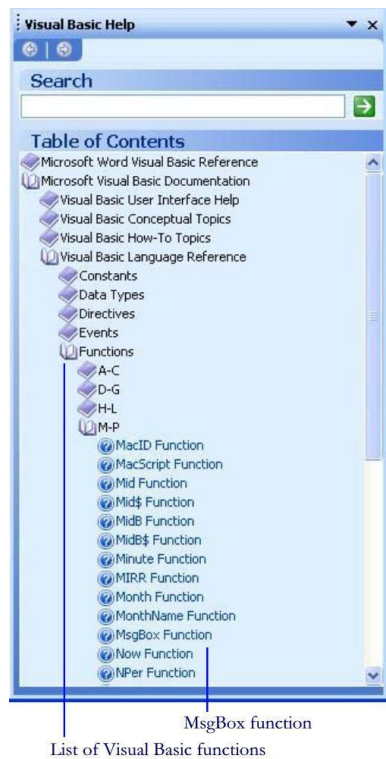


Visual Basic Functions

Visual Basic provides other functions beside MsgBox. For a list of Visual Basic functions follow these steps.

Steps

- 1) From the Visual Basic Editor's menu, select Help
- 2) Select Microsoft Visual Basic Help.
- 3) When the Visual Basic Help window appears expand the Microsoft Visual Basic Documentation folder.
- 4) Expand the Visual Basic Language Reference folder.
- 5) Expand the Functions folder.
- 6) A list of Visual Basic functions by alphabetical order appears (Figure 1.14).
- 7) Choose any of the functions, the MsgBox function being one of them, to see the general syntax, detailed information and examples.

Figure 1.14 *Visual Basic functions*

Run Your Macro

Now you are ready to run your macro. To try the macro follows these steps.

Steps

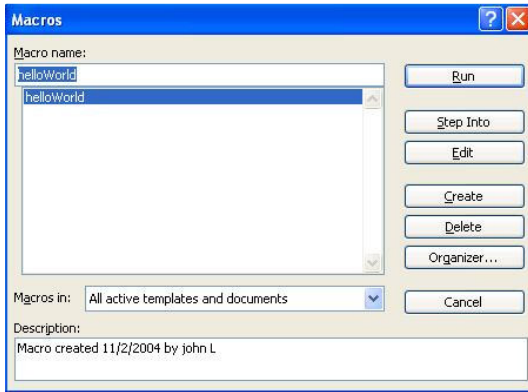
- 1) Close the Visual Basic Editor.
- 2) In the Word document window select Tools/Macro/Macro from the menu (Figure 1.15).

Figure 1.15 *Run your macro*

Steps

- 1) The Macro window appears (Figure 1.16).
- 2) helloWorld should be one of the macros in the Macro Name box.
- 3) Select helloWorld in the Macro Name box and click the Run button.

Figure 1.16 *helloWorld macro*



The Dialog box appears with the (Hello World!) greeting (Figure 1.17).

Figure 1.17 *Result of helloWorld macro*



Debugging

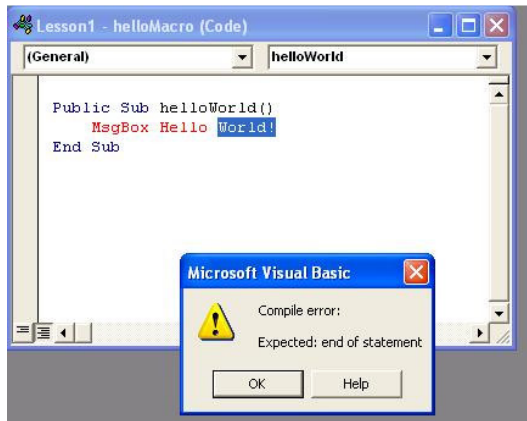
In programming or coding your codes may not work the first time you try them. Debugging refers to the process of getting the bug out of your codes so they work. Visual Basic Editor provides many tools to help you debug your program. You will be introduced to some of these tools as you progress in these chapters. For now, let's give you some simple examples.

Compile Error

Programming or coding in Visual Basic involves two steps. First you type in your code. Then you try it out by running your macro. Each time you type a line of code Visual Basic Editor invokes the Visual Basic compiler to parse (examine) the line you enter to see if there is any error. If there is an error it is called a compile error.

Steps

- 1) Start the Visual Basic Editor and open your myModule module.
- 2) In the MsgBox line of code removes the two quotation marks from “Hello World!” and click anywhere in the Code window.
- 3) A dialog box appears telling you there is a compile error (Figure 1.18).
- 4) Click the OK box, correct the error by adding back the 2 quotation marks and you will be fine.

Figure 1.18 *Compile error*

Runtime Error

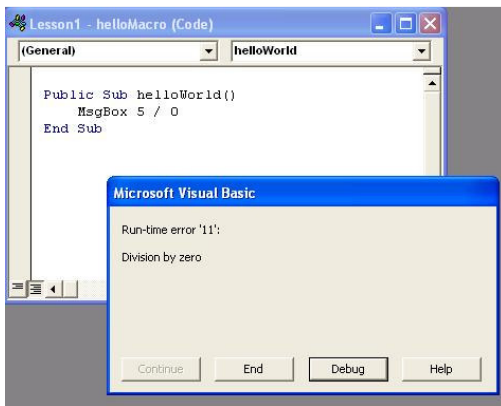
After you finish coding your macro and the compiler cannot detect any compile error you are ready to test running your codes. Any error at this stage, while you are testing or running your codes, is called a runtime error. Though the compiler cannot find any compile or syntax error in the codes, your program may not run if it has logical error.

For an example of runtime error, follow these steps.

Steps

- 1) Start the Visual Basic Editor and open your myModule module.
- 2) Replace the MsgBox line of code by MsgBox 5 / 0 (Figure 1.19).
- 3) Run the helloWorld macro.
- 4) A dialog box appears saying there is a runtime error (Figure 1.19).
- 5) This error occurs because you are dividing 5 by the number 0, causing an overflow error.
- 6) Select the End box to close the dialog box.
- 7) Correct the error by removing / 0 from the MsgBox statement and run the macro again and you should be fine.

Figure 1.19 *Runtime error*



This concludes Chapter 1.

Chapter 2 Object Programming

What is an Object?

Object and Class

An **object** is a combination of code and data that can be treated as a unit. An object contains properties, methods and events. **Properties** are data or attributes of the object. The code or **methods** of the object perform actions that can affect its own properties. **Events** are something that happen associated with an object. For example, the Document object has the Paragraphs property, the Range method and the Open event.

Objects are defined and created from **classes**. An object is an **instance** of a class. When you create an object, you are creating a copy or an instance of a class. All objects are created as identical copies of their class. After they are created they exist as individual objects, and their properties can be changed. For example assume object A and object B are objects of the Paragraphs class, so they both have the Count property but object A's Count property can have the value of 20 while object B's Count property has the value of 40.

The following code sets A as an object of the Application class. That is, you've created a copy or an instance of the Application class. As an object of the Application class, A has the same properties, methods and events of the Application class.

```
Set A = Application
```

Objects are fundamental to Office Visual Basic programming. Every unit of content and functionality in Office — each workbook, worksheet, document, range of text, slide, and so on — is an object that you can control programmatically in Visual Basic. When you understand how to work with objects, you're ready to automate tasks in Office.

Object Contain Other Objects

Properties and methods of an object can be objects by themselves, and as objects contain their own properties, methods and events. For example, the Document object contains the Paragraphs property and the Range method. The Paragraphs property is an object and contains its own properties and methods. The Range method is an object and contains its own properties and methods. The structure and relationship of objects containing other objects define the **object hierarchy**.

To use or refer to a particular object you have to navigate through the object hierarchy to get to the object you want. For example the ActiveDocument object contains the Words(1) method. The Words(1) method is an object itself and contains the Bold property. So to refer to the bold property of the first word of the active document you use the following code:

```
ActiveDocument.Words(1).Bold = True
```

Object's Property Example

The following code sets the first word of the active document bold, by assigning the True value to the ActiveDocument.Words(1).Bold property.

Completed Code

```
Public Sub objProperty()  
    ActiveDocument.Words(1).Bold = True  
End Sub
```

To try this example follows these steps.

Steps

- 1) Start a new Word document and the Visual Basic Editor.
- 2) Insert a new module and keep the default name as Module1.
- 3) Insert a new procedure (macro) and name it objProperty.
- 4) Type in the above code (Figure 2.1).
- 5) Type in some sentence like the one in Figure 2.2.
- 6) Run your macro and see the first word of the sentence set to bold as shown in Figure 2.2.

Figure 2.1 *objProperty* macro

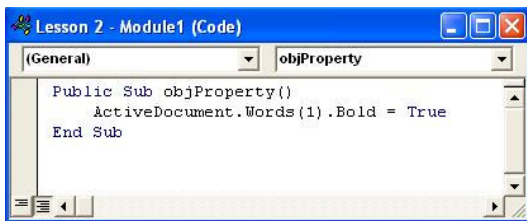
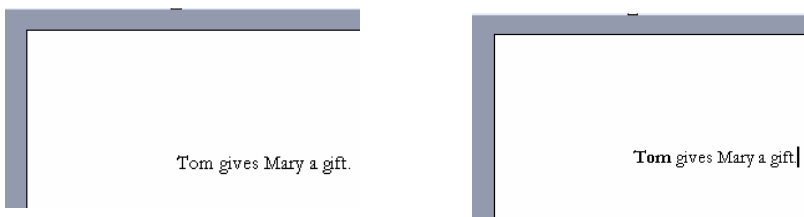


Figure 2.2

Result of objProperty macro before and after



This example shows the hierarchy structure of objects. The Application object contains the property ActiveDocument. The ActiveDocument property is an object by itself. The ActiveDocument property object contains the Words property. The Words property is an object by itself. The Words property object contains the Item(1) method. The Item(1) method is an object by itself. The Item(1) object has the Bold property. So to refer to the bold property of the first word of the active document you use the following expression:

```
Application.ActiveDocument.Words.Item(1).Bold.
```

Because the Application object is the default object and the Item(1) method is the default method of the Words object you drop the term Application and the term Item and the above expression becomes:

```
ActiveDocument.Words(1).Bold.
```

This is the expression you've used in your code in this example.

Object's Method Example

The following code inserts the text "Today " (with a trailing blank) before the first word of the active document. You use the InsertBefore method of the ActiveDocument.Words(1) object to insert the text "Today " (with a trailing blank) before the first word of the active document.

Completed Code

```
Public Sub objMethod()  
    ActiveDocument.Words(1).InsertBefore "Today "  
End Sub
```

To try this example follows these steps.

Steps

- 1) Start the Visual Basic Editor.
- 2) Use the same module as the one in the previous example.
- 3) Insert a new procedure (macro) and call it objMethod.
- 4) Type in the above code (Figure 2.3).
- 5) Type in some sentence like the one in Figure 2.4.
- 6) Run your macro and see the text "Today " (with a trailing blank) being inserted before the first word of the sentence as shown in Figure 2.4.

Figure 2.3 *objMethod* macro

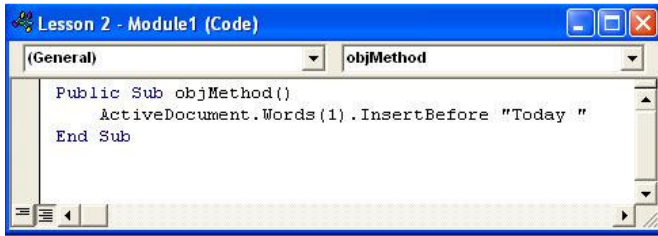
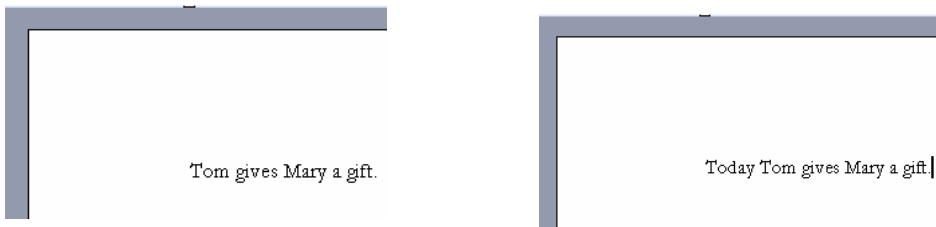


Figure 2.4

Result of the objMethod macro before and after



Similar to the previous example, this example shows the hierarchy structure of objects. The `Item(1)` object contains the `InsertBefore` method. So to refer to the `InsertBefore` method you use the expression

`Application.ActiveDocument.Words.Item(1).InsertBefore "Today "`.

Because the `Application` object is the default object and the `Item(1)` method is the default method of the `Words` object you drop the term `Application` and the term `Item` and the above expression becomes:

`ActiveDocument.Words(1).InsertBefore "Today "`.

This is the expression you've used in your code in this example.

What is a Collection Object?

A collection object is a set of related objects. Collection objects are objects themselves. Think of a collection object as an array of related objects. Each object within a collection object is called an element or an item of the collection object. Collection objects are useful in that you can perform an operation on all the objects in a given collection object as a group more efficiently.

In the previous example you've seen the example of a collection object. `Words` is a collection object and you use the `Item` method of the `Words` collection object to refer to the items of the collection object as:

`Words.Item(1), Words.Item(2), Words.Item(3),`

Because the `Item` method is the default method of the `Words` collection object you drop the term `Item` and refer to the items of the collection object as:

`Words(1), Words(2), Words(3),`

As illustrated by the previous example you use the following expression to refer to the first word of the active document:

`ActiveDocument.Words(1)`

Likewise you use the following expressions to refer to the second and the third word of the active document and so on:

```
ActiveDocument.Words(2), ActiveDocument.Words(3), ...
```

While the Words object is not one of them, many objects have similar names, one with a `-s` and one without a `-s`. For example the Paragraphs (with a `-s`) object and the Paragraph (without a `-s`) are different objects. The Paragraphs object is a collection object while the Paragraph is not. They are both objects in their own right, but as different objects they have different properties and methods. You can think of the Paragraphs collection object as one that contains all the paragraphs in a given document, whereas the Paragraph object refers to an individual paragraph in the document.

Object Variable

Using Objects Directly

You can use objects directly without using object variable to refer to it. You've seen examples of that. You use the expression

```
ActiveDocument.Words(1)
```

to refer to the `ActiveDocument.Words(1)` object directly. However if you have to refer to the same object in different parts of your codes it is too much typing, easy to make mistake and make the code difficult to read. It is preferable to declare an object variable and use the object variable to refer to the same object.

Declaring Object Variables

Declaring an object variable is easy. The following expression declares `doc` as the object variable of the Document class.

```
Dim doc as Document
```

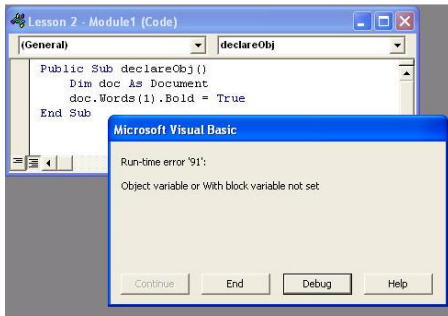
Because `doc` is now an object variable of the Document class you can use all the properties and methods of the Document object (class) after you assign to `doc` a reference to an object.

Declaring Object Variables without Assigning Object References

You must assign an object reference to the object variable you've declared before you start using it. If not, you will get a run-time error. When you assign an object reference to an object variable you tell the computer the physical memory address of the object you will be referencing in your code. The following code declares `doc` as a Document object. It then uses it to refer to the property of an object. If you run this code you will get a run-time error (Figure 2.5).

```
Public Sub declareObj()  
    Dim doc As Document  
    doc.Words(1).Bold = True  
End Sub
```

Figure 2.5 *declareObj* macro



Declaring Integer Variables without Assigning Values

It is interesting to compare the previous example with one where you declare an integer variable other than an object variable. The following code declares `intTest` as a variable of the Integer data type and then uses it without assigning values to it. If you run the code in a macro (Figure 2.6) you will get the result in Figure 2.7, where the dialog box displays the number 1, the result of `(intTest + 1)`. The computer assigns an initial value of 0 to the integer variable `intTest` for you, so you get the result of $0 + 1 = 1$.

```
Public Sub declareInteger()  
    Dim intTest As Integer  
    MsgBox (intTest + 1)  
End Sub
```

Figure 2.6 *declareInteger* macro

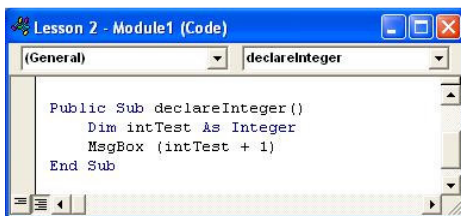


Figure 2.7 Result of *declareInteger* macro



Run-Time Error and Good Programming Practice

Why do you get different results when you declare two different types of variable, one an object variable and the other an integer variable, and when you use them before assigning object reference to the former or integer values to the latter?

When you declare an integer variable the computer assigns a default value of 0 to it when you use it. You see that in the previous example. When you declare an object variable the computer is not able to assign to it a default object when you use it so it quits and give you a run-time error.

In any event it is a good programming practice to assign integer values to your integer variables before you use them in your code. For example if you want your integer variable to be 100 and you forget to assign it. The computer will happily use the default value of 0 in the computation in your code and give you the incorrect answer. Such error can be difficult to detect.

Assigning Object Reference to an Object Variable

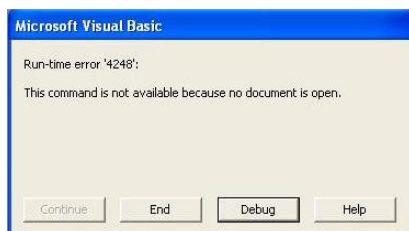
You assign an object reference to an object variable by assigning to the object variable a reference to an object. You assign an object reference to an object variable by using the Set statement. The following code assigns to the object variable doc the ActiveDocument object. That is, you assign to the object variable doc a reference to the ActiveDocument object.

```
Set doc = ActiveDocument.
```

When you assign the ActiveDocument object reference to the object variable doc you must have at least one Word document open, otherwise you get a run-time error when you use the object variable doc to perform operation in your code (Figure 2.8). If you have more than one Word documents open at the time, the Word document with the focus (the one selected) will be used.

When you assign the ActiveDocument object reference to an object variable and no Word document is open. You get this run-time error when you use the object variable.

Figure 2.8 *Assign ActiveDocument object reference when no Word document is open*



When you assign an object reference to an object variable by using the Set statement both the object variable and the object reference must be of the same class.

For example in the following code you've declared doc as an object variable of the Document class so you are fine because the ActiveDocument object is also of the Document class.

```
Dim doc as Document
Set doc = ActiveDocument
```

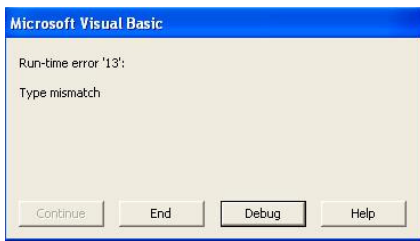
The following code will give you a run-time error (Figure 2.9) because you are assigning a reference to the ActiveDocument.Characters(1) object, which is of the Range class, to the object variable doc which is of the Document class.

```
Dim doc as Document
```

```
Set doc = ActiveDocument.Characters(1)
```

When you assign to an object variable a reference to an object of the different class you will get run-time error.

Figure 2.9 *Assign object reference of different class*



Note

When you assign an object reference to an object variable by using the Set statement both the object variable and the object reference must be of the same class.

Assign Object Variable Example

The following code declares doc as an object variable of the Document class and assign to it a reference to the ActiveDocument object. You then display doc using the MsgBox dialog box.

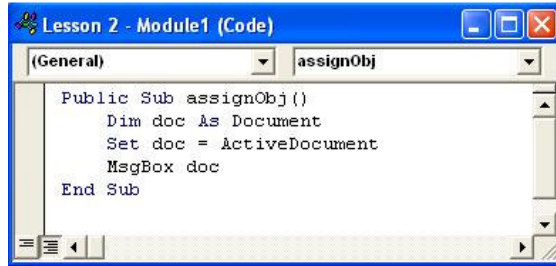
Completed Code

```
Public Sub assignObj()  
    Dim doc As Document  
    Set doc = ActiveDocument  
    MsgBox doc  
End Sub
```

To try this example follows these steps.

Steps

- 1) Make sure you have at least one Word document open.
- 2) Start the Visual Basic Editor.
- 3) Use the same module as the one in the previous example.
- 4) Insert a new procedure (macro) and call it assignObj.
- 5) Type in the above code (Figure 2.10).
- 6) Run your macro and the dialog box displays the name of the active document as shown in Figure 2.11.

Figure 2.10 *assignObj* macroFigure 2.11 Result of *assignObj* macro

Working with Properties and Methods

Assign Values to Object's Properties

When you assign value to the property of an object, you give it a new value. You've seen example of that in the following code where you assign the value of True (1 will do too) to the Bold property of the ActiveDocument.Words(1) object:

```
ActiveDocument.Words(1).Bold = True
```

You use the following syntax to assign value to a property for any type of object:

```
object.property = setting
```

When you assign a value to the property of an object both the value you are assigning and the property must not be of the conflicting data type. For example in the following code the assignment is fine because the property Bold is of the numeric data type (True = 1, False = 0). The value True (or 1) will format the text to be bold. The value False (or 0) will format the text to be non-bold.

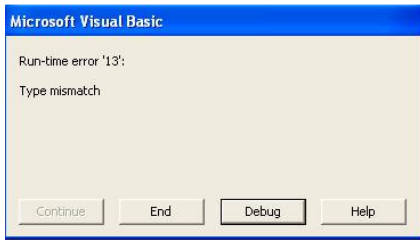
```
ActiveDocument.Words(1).Bold = True
```

When the property and the value you are assigning are of the conflicting data type you get a run-time error. The following example will give you a run-time error (Figure 2.12) because you are assigning the text (test) which is of the string data type, to the Bold property which is of the numeric data type. The string data type conflicts with the numeric data type.

```
ActiveDocument.Words(1).Bold = "test"
```

When you assign a value to the property of an object of conflicting data type you get a run-time error.

Figure 2.12 *Assign value to property of an object of conflicting data type*



Note

When you assign a value to the property of an object both the value you are assigning and the property must not be of the conflicting data type

Assign Value to Object's Property Example

This is similar to an earlier example except the following code declares `doc` as an object variable of the `Document` class, assign to it a reference to the `ActiveDocument` object, then make the first word of the active document bold, by assigning the `True` value to the `doc.Words(1).Bold` property.

Completed Code

```
Public Sub assignProperty()  
    Dim doc As Document  
    Set doc = ActiveDocument  
    doc.Words(1).Bold = True  
End Sub
```

To try this example follows these steps.

Steps

- 1) Start the Visual Basic Editor.
- 2) Use the same module as the one in the previous example.
- 3) Insert a new procedure (macro) and call it `assignProperty`.
- 4) Type in the above code (Figure 2.13).
- 5) Type in some sentence like the one in Figure 2.14.
- 6) Run your macro and see the first word of the sentence set bold (Figure 2.14).

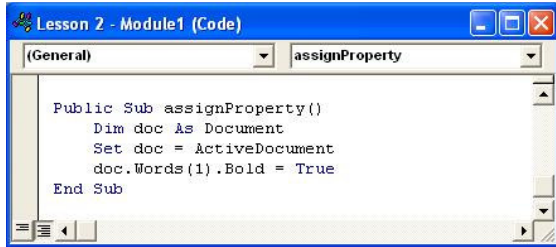
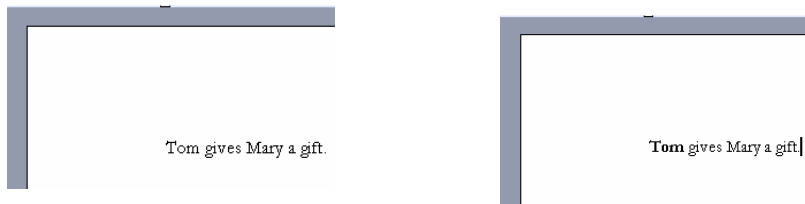
Figure 2.13 *assignProperty* macro

Figure 2.14

Result of *assignProperty* macro before and after



Set Statement

You've used the Set statement to assign an object reference to an object variable as in the following example:

```
Set doc = ActiveDocument.
```

In general you can assign any object reference to an object variable as long as they are of the same class (same object type) using the following general syntax:

```
Set objVariable = objectReference
```

The object reference can be a Property object or a Method object, as Properties and Methods can be objects themselves. When you assign a Property object reference to an object variable both the Property and the object variable must be of the same class (same object type).

When you assign a Method object reference to an object variable, the Method object must return an object reference and the object being returned and the object variable must be of the same class (same object type). Some Method objects don't return anything, for example, a Sub procedure method.

The following code shows an example of assigning a Property object reference to an object variable. You declare doc as an object variable of the Document class and assign to it a reference to the ActiveDocument property of the Application object. The assignment is permitted because both doc and the ActiveDocument property are of the same Document class.

```
Dim doc as Document
```

```
Set doc = Application.ActiveDocument
```

The following code is equivalent to the above code because the Application object is the default object so the term is dropped from the expression.

Dim doc as Document

Set doc = ActiveDocument

The following code shows an example of assigning a Method object reference to an object variable. You declare myRange as an object variable of the Range class and use the Item(1) method to return a reference to the object that represents the first character of the active document. The assignment is permitted because the object being returned and myRange are of the same Range class.

Dim myRange As Range

Set myRange = ActiveDocument.Characters.Item(1)

The following two expressions are equivalent because the Item method is the default method of the Characters class so Item is dropped from the expression.

Set myRange = ActiveDocument.Characters.Item(1)

Set myRange = ActiveDocument.Characters(1)

Note

When you assign a Property object reference to an object variable using the Set statement both the Property and the object variable must be of the same class (same object type).

When you assign a Method object reference to an object variable using the Set statement, the Method object must return an object reference and the object being returned and the object variable must be of the same class (same object type).

Set Statement Example – Property Object

The following code declares paras as an object variable of the Paragraphs class:

Dim paras As Paragraphs

The following code uses the Set statement to assign to paras a reference to the Paragraphs property of the ActiveDocument object. By this assignment you've made paras a collection object referring to all the paragraphs of the active document.

Set paras = ActiveDocument.Paragraphs

The following code sets the first word of the second paragraph of the active document bold.

paras(2).Range.Words(1).Bold = True

The following shows the completed code.

Completed Code

```
Public Sub setProperty()
    Dim paras As Paragraphs
    Set paras = ActiveDocument.Paragraphs
    paras(2).Range.Words(1).Bold = True
End Sub
```

To try this example follows these steps.

Steps

- 1) Start the Visual Basic Editor.
- 2) Use the same module as the one in the previous example.
- 3) Insert a new procedure (macro) and name it setProperty.
- 4) Type in the above code (Figure 2.15).
- 5) Type in 2 paragraphs like the one in Figure 2.16.
- 6) Run your macro and see the first word of the second paragraph changed to bold (Figure 2.16).

Figure 2.15 *setProperty* macro

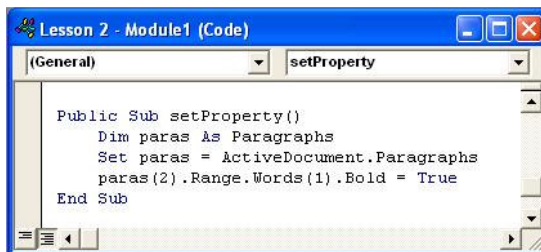
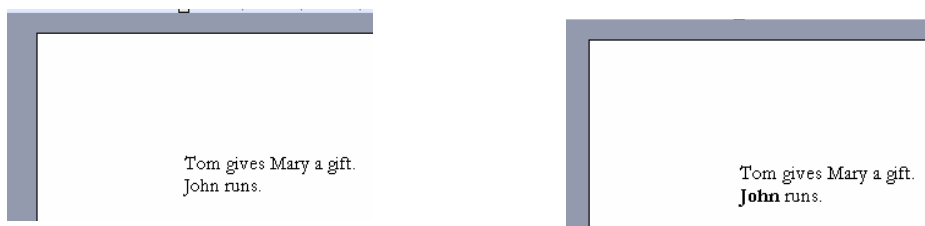


Figure 2.16

Result of setProperty macro before and after



Set Statement Example – Method Object

The following code declares para as an object variable of the Paragraph class:

```
Dim para As Paragraph
```

The following code uses the Set statement to return the reference to an object of the same Paragraph class using the Add method. The Add method adds a new paragraph mark. As a result para now is an object of the Paragraph class representing the new paragraph (an empty paragraph).

```
Set para = ActiveDocument.Paragraphs.Add
```

The following code adds the text “This sentence is new.” to the new paragraph.

```
para.Range.Text = "This sentence is new."
```

The following shows the completed code.

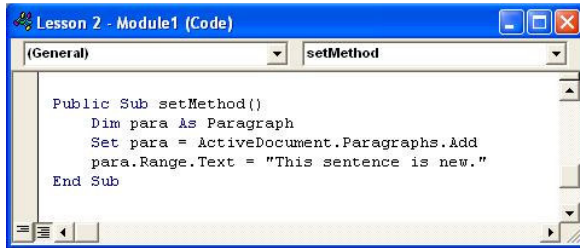
Completed Code

```
Public Sub setMethod()  
    Dim para As Paragraph  
    Set para = ActiveDocument.Paragraphs.Add  
    para.Range.Text = "This sentence is new."  
End Sub
```

To try this example follows these steps.

Steps

- 1) Start the Visual Basic Editor.
 - 2) Use the same module as the one in the previous example.
 - 3) Insert a new procedure (macro) and name it setMethod.
 - 4) Type in the above code (Figure 2.17).
 - 5) Type in a sentence like the one in Figure 2.18.
 - 6) Run your macro and see a new paragraph added (Figure 2.18).
-

Figure 2.17 *setMethod* macro


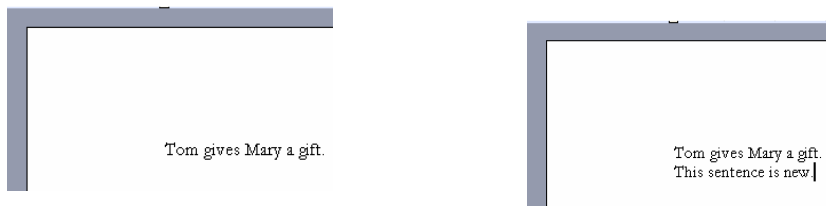
```

Lesson 2 - Module1 (Code)
setMethod
Public Sub setMethod()
    Dim para As Paragraph
    Set para = ActiveDocument.Paragraphs.Add
    para.Range.Text = "This sentence is new."
End Sub

```

Figure 2.18

Result of *setMethod* macro before and after



Set Statement Returns Object Reference

You've seen that the Set statement returns another object, or more precisely speaking, a reference to another object. The following two statements produce completely different results.

```
Set myRange = ActiveDocument.Words(1)
```

```
str = ActiveDocument.Words(1)
```

The former returns a reference to an object of the Range class whereas the latter returns the text content of the first word (of string data type) of the document. Specifically the former returns the reference (memory address) of the Words(1) method (object) whereas the latter returns the first word itself (content).

Because the Words(1) method returns an object of the Range class you have to declare myRange as an object of the Range class as follows:

```
Dim myRange as Range
```

```
Set myRange = ActiveDocument.Words(1)
```

You have to declare str as a variable of the string data type for the following code to work.

```
Dim str as String
```

```
str = ActiveDocument.Words(1)
```

Set Statement Example – Comparison

The following code illustrates the last discussion.

Completed Code

```
Public Sub setCompare()  
    Dim myRange As Range  
    Dim str As String  
    Set myRange = ActiveDocument.Words(1)  
    str = ActiveDocument.Words(1)  
    MsgBox myRange.Font.Size  
    MsgBox str  
End Sub
```

To try this example follows these steps.

Steps

- 1) Start the Visual Basic Editor.
- 2) Use the same module as the one in the previous example.
- 3) Insert a new procedure (macro) and name it setCompare.
- 4) Type in the above code (Figure 2.19).
- 5) Type in a sentence like the one in Figure 2.20.
- 6) Run your macro and the first dialog box displays the font size of the first word of the document.
- 7) The second dialog box displays the text of the first word in the document (Figure 2.20).

Figure 2.19 *setCompare* macro

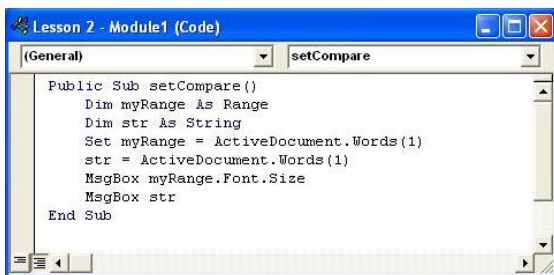
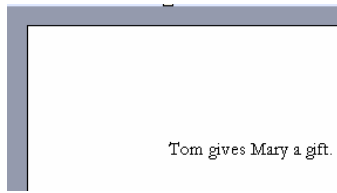


Figure 2.20

Result of setCompare macro before and after



Displays the font size of the first word



Displays the text of the first word



Sub Procedure and Function Methods

A method is an object defined by a class. There are two kinds of methods, one that returns a value or an object, as a function does, and one that don't return anything, as a Sub procedure does.

- 1) Function (method)
- 2) Sub procedure (method)

Sub procedure performs some operation but returns nothing. For example, Space2 is Sub procedure (method) of the Paragraphs object, so the following code makes all the paragraphs of the document double-spaces:

```
ActiveDocument.Paragraphs.Space2
```

You've used function (method) in earlier example such as the following code where you use the Add function (method) to return an object of the Paragraph class.

```
Set para = ActiveDocument.Paragraphs.Add
```

A function (method) can return a value that is not an object as in the following code where you use the Calculate function (method) to return a number. If the Sentences(1) object has the text 2 +3, the value returned will be 5.

```
ActiveDocument.Sentences(1).Calculate
```

Note

A function (method) returns a value or another object. A Sub procedure (method) returns nothing.

Sub Procedure Method Example

A Sub procedure (method) performs some operation. It doesn't return a value or another object as a function (method) does. The following code uses the Space2 Sub procedure (method) of the Paragraphs object to make all the paragraphs of the document to be double-spaces:

Completed Code

```
Public Sub subProc()  
    ActiveDocument.Paragraphs.Space2  
End Sub
```

To try this example follows these steps.

Steps

- 1) Start the Visual Basic Editor.
- 2) Use the same module as the one in the previous example.
- 3) Insert a new procedure (macro) and name it subProc.
- 4) Type in the above code (Figure 2.21).
- 5) Type in two sentence (single-space) like the one in Figure 2.22.
- 6) Run your macro and see the two paragraphs made double-space (Figure 2.22).

Figure 2.21 *subProc macro*

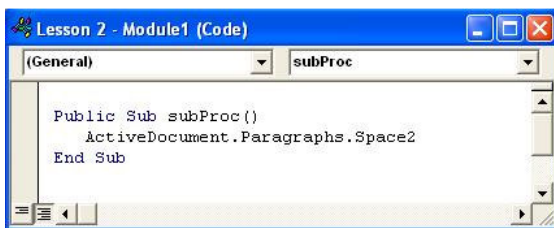
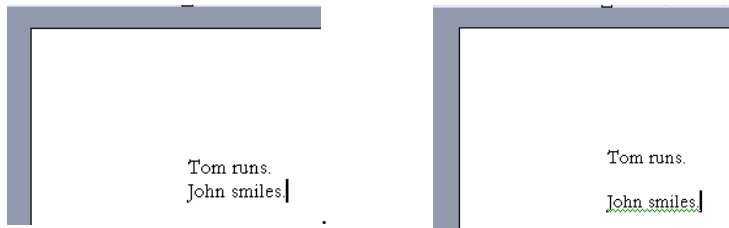


Figure 2.22

Result of *subProc* macro before and after



Function that Returns a Value

A function (method) can return a value that is not an object as in the following code where you use the Calculate function (method) to return a number. If the Sentences(1) object has the text 2 +3, the value returned will be 5.

Completed Code

```
Public Sub fnCal()
    MsgBox ActiveDocument.Sentences(1).Calculate
End Sub
```

To try this example follows these steps.

Steps

- 1) Start the Visual Basic Editor.
- 2) Use the same module as the one in the previous example.
- 3) Insert a new procedure (macro) and name it fnCal.
- 4) Type in the above code (Figure 2.23).
- 5) Type a sentence of 2 + 3 as shown in Figure 2.24.
- 6) Run your macro and the dialog box displays the value of 5, the calculated result of 2 + 3 (Figure 2.24).

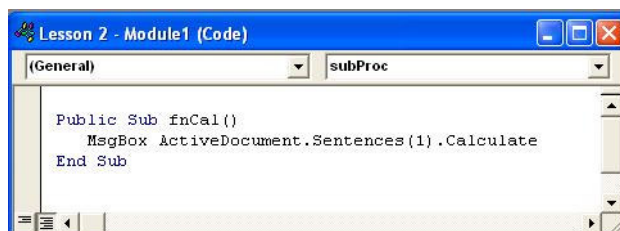
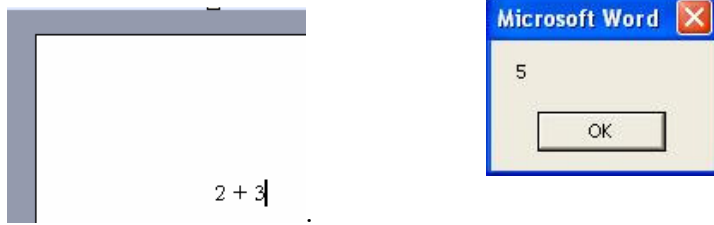
Figure 2.23 *fnCal* macro

Figure 2.24

Result of the fnCal macro before and after



This concludes Chapter 2.

Chapter 3 More Object Programming

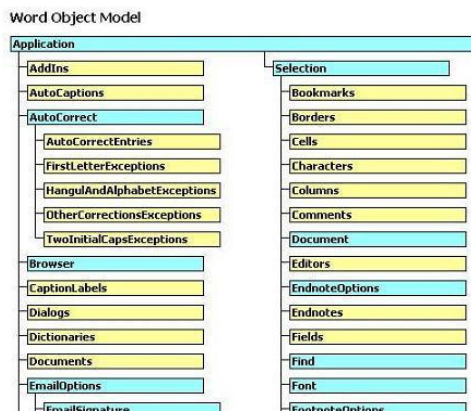
Word Object Model

Where Is My Object?

To automate or program Word effectively using Visual Basic requires an understanding of the commonly used Word objects and their properties and methods. It's important to know an object's place in the object model, because before you can work with an object, you have to navigate through the object model to get to it. This usually means that you have to step down through all the objects above it in the object hierarchy to get to it.

Figure 3.1 shows a partial list of Word objects from Visual Basic Help. To navigate through pages and pages of charts like this you are sure to get lost in search for the object you are looking for. For this purpose the Object Browser is an indispensable tool, giving you a quick way to navigate through the object models and showing you what objects are available including the properties and methods of those objects.

Figure 3.1 *Partial list of Word object model*



Object Browser

To display the Object Browser follow these steps.

Steps

- 1) Start Visual Basic Editor.
- 2) From the View menu, choose Object Browser, or
- 3) Press F2, or
- 4) Click the Object Browser button on the toolbar.
- 5) The Object Browser appears (Figure 3.2).

Figure 3.2 *Object Browser*

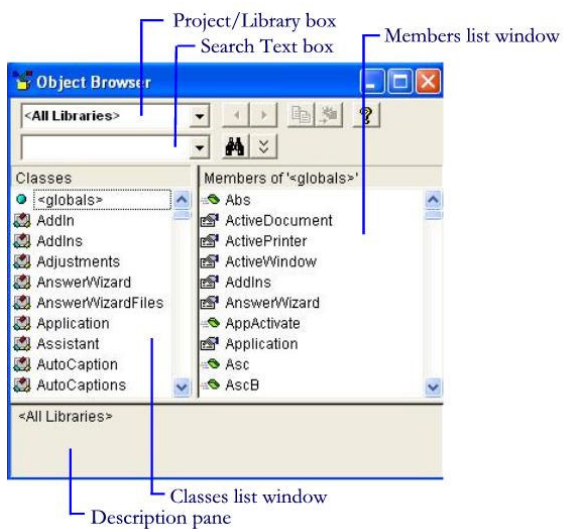









Figure 3.2 shows the Object Browser.

- 1) Project/Library box allows you to select a single library or project, or to view all libraries and projects.
- 2) Search Text box lets you find objects and members.
- 3) Classes list window display all the classes belonging to the selected project or library in the Project/Library box.
- 4) Members list window shows the properties, methods, and events that belong to the class selected in the Classes list window.
- 5) Description pane shows the types and arguments of properties, methods, events and their descriptions.

Icons Used in Object Browser

Figure 3.3 shows icons used in the Object Browser. The following sections describe these in more details.

Figure 3.3 *Icons used in Object Browser*

Icon	Element Type
	Class
	Property
	Default Property
	Method
	Default Method
	Event
	Enumeration

Default Property

The default property of an object has the icon of a hand holding a note with a tiny blue dot in the upper left hand corner (Figure 3.3, Figure 3.4). Many objects have default properties. You can drop the name of a default property of an object when referencing it. For example, Figure 3.4 shows that the Text property is a default property of the Range class. So the following two expressions are equivalent when you use them in your code (myRange has been declared as an object variable of the Range class).

```
myRange.Text = "Tom smiles."
```

or

```
myRange = "Tom smiles."
```

Knowing default properties of objects helps you understand abbreviated code such as the above example, when you have to read code written by other programmers.

Figure 3.4 *Default property*



Note

You can drop the name of a default property of an object when referencing it.

Default Method

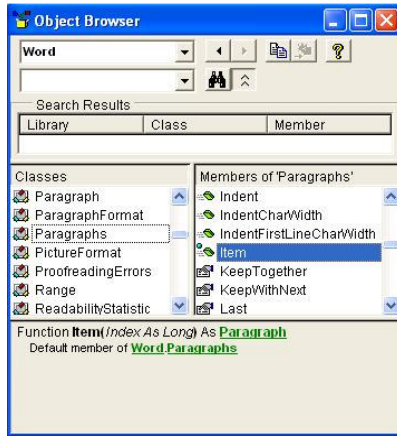
The default method of an object has the icon of a 3-dimensional bluish rectangle with a tiny blue dot in the upper left hand corner (Figure 3.3, Figure 3.5). Many objects have default methods. You can drop the name of a default method of an object when referencing it. For example, Figure 3.5 shows that the Item method (function) is a default method of the Paragraphs class. So the following two expressions are equivalent when you use them in your code (paras has been declared as an object variable of the Paragraphs class).

```
paras.Item(2).Range.Text = "Tom smiles."
```

Or

```
paras(2).Range.Text = "Tom smiles."
```

Knowing default methods of objects helps you understand abbreviated code such as the above example, when you have to read code written by other programmers.

Figure 3.5 *Default method*

Note

You can drop the name of a default method of an object when referencing it.

Default Property and Method Example

This example adds a new paragraph to the end of the document. It illustrates the use of default properties and default methods.

The following code declares `paras` as an object variable of the `Paragraphs` class. You then use the `Set` statement to assign to `paras` a reference to the `Paragraphs` property of the `ActiveDocument` object.

```
Dim paras As Paragraphs
```

```
Set paras = ActiveDocument.Paragraphs
```

Assuming the document has only one paragraph, the following code uses the `Add` method (a function method) of `paras` (a `Paragraphs` object) to add a new paragraph mark to the end of the document.

```
paras.Add
```

The expression `Paras.Item(2)` uses the `Item(2)` method (a function method) of `paras` (a `Paragraphs` object) to return a reference to an object of the `Paragraph` class that represents the second paragraph of the document. That is, now `Paras.Item(2)` is a `Paragraph` object that represents the second paragraph of the document. The expression `Paras.Item(2).Range` uses the `Range` property of `Paras.Item(2)` (a `Paragraph` object) so now `Paras.Item(2).Range` becomes an object of the `Range` class. The following code assigns the text "That was yesterday." to the `Text` property of `Paras.Item(2).Range` (a `Range` object).

```
Paras.Item(2).Range.Text = "That was yesterday."
```

Because the `Item` method is a default method of the `Paragraphs` class and the `Text` property is a default property of the `Range` class the two terms are dropped and you use the following statement instead of the above statement.

```
Paras(2).Range = "That was yesterday."
```

The following shows the completed code.

Completed Code

```
Public Sub defaultPropertyMethod()  
    Dim paras As Paragraphs  
    Set paras = ActiveDocument.Paragraphs  
    paras.Add  
    paras(2).Range = "That was yesterday."  
End Sub
```

To try this example follows these steps.

Steps

- 1) Start a new Word document and the Visual Basic Editor.
- 2) Insert a new module and keep the default name as Module1.
- 3) Insert a new procedure (macro) and name it defaultPropertyMethod.
- 4) Type in the above code (Figure 3.6).
- 5) Type in one (only one) paragraph like the one in Figure 3.7.
- 6) Run your macro and see a second paragraph of text added (Figure 3.7).

Figure 3.6 *defaultPropertyMethod* macro

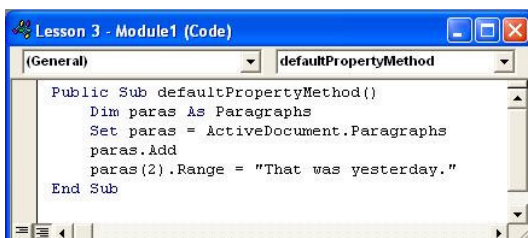
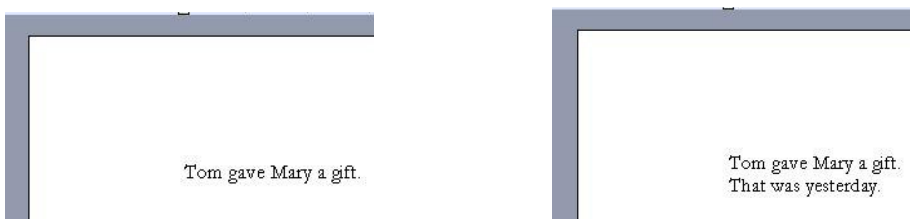


Figure 3.7 *Result of odefaultPropertyMethod* macro before and after



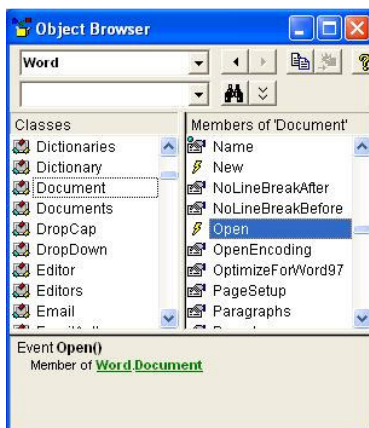
Event and Event Procedure

Recall that an object contains properties, methods and events. An event in the member List window in the Object Browser has the icon of a little yellowish lightning strike. Figure 3.8 shows the Open event of the Document class. When you open a document the Open event (of the Document object representing the document you've opened) is said to occur. This causes a message to be sent to the Windows operating system which processes the message and broadcasts it to all the windows (objects) active in the computer memory. Each window (object) can then take appropriate action based on its own instructions for dealing with this message (the Open event triggered by your opening of this particular document). The instruction (code) you write for dealing with this event (the Open event) is an event procedure or an event handler.

You've seen in Chapter 1 the three types of procedure, Sub procedure (macro), function procedure and property procedure. The event procedure is yet just another type of procedure.

When you write the event procedure for an object the naming convention for the name of the event procedure is you combine the object's name, an underscore `_`, and the event name. For example, when you write an event procedure for the Open event of a Document object you use the procedure name of `Document_Open`.

Figure 3.8 *Event procedure in Object Browser*



Event Procedure Example

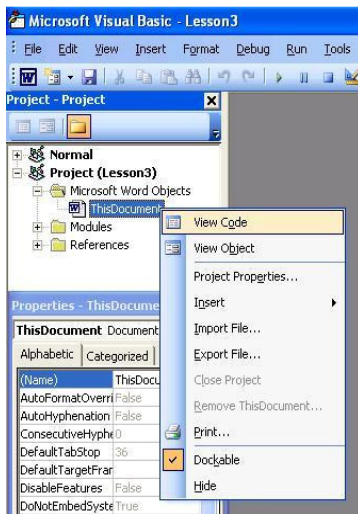
In this example you write an event procedure that is invoked to display a dialog box to greet you whenever you open the Word document.

Follow these steps to create an event procedure:

Steps

- 1) Open the Visual Basic Editor from your document.
- 2) From the Project window right click ThisDocument and select View Code (Figure 3.9).

Figure 3.9 *Create an event procedure*



The Code window for ThisDocument appears (Figure 3.10).

Follow these steps to add your code.

Steps

- 1) The ThisDocument (Code) window has two dropdown boxes (Figure 3.10). Scroll down to the Document object and select it.
- 2) With the left dropdown box of the code window displaying Document, the right dropdown box contains a list of the events associated with the Document object. Select Open, the Open event of the Document object (Figure 3.10).
- 3) Visual Basic editor automatically generates the heading Private Sub Document_Open() and the ending End Sub of the event procedure for you. Type in the above code (Figure 3.10) between the heading and the ending of the event procedure.
- 4) Choose File/Save from the Visual Basic menu.
- 5) Close your Word document.
- 6) Open your Word document.
- 7) You will see the Welcome greeting box appearing (Figure 3.11).

Figure 3.10 *Document_Open event procedure*

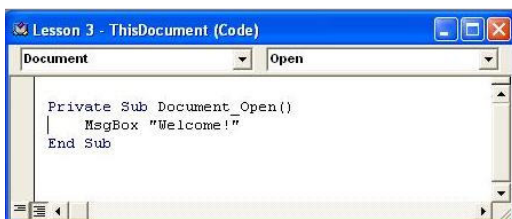
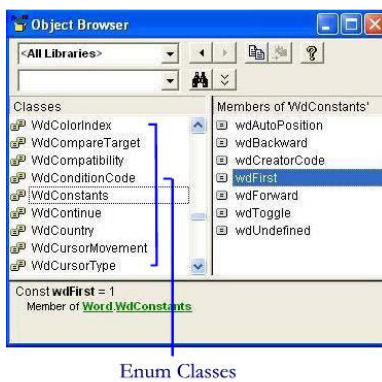


Figure 3.11 Result of *Document_Open* event procedure

What is an Enum Class?

An enum class is a special class that contains just data but no methods. An enum class associates constant values with names. For example Visual Basic has built-in enum class `WdConstants` that associates the name `wdFirst` with the value of 1 (Figure 3.12). You use the name in an enum class instead of a value to improve clarity of your code.

Figure 3.12 *Enum class*

For example, create a macro with the following code and run your macro you will get a dialog box displaying the value of 1 (Figure 3.13).

```
Public Sub wd_First()

    MsgBox wdFirst

End Sub
```

Figure 3.13



Navigating the Object Hierarchy Using Object Browser

In an earlier example you use the following code to add a new paragraph of text to the active document.

```
Public Sub defaultPropertyMethod()  
  
    Dim paras As Paragraphs  
  
    Set paras = ActiveDocument.Paragraphs  
  
    paras.Add  
  
    paras(2).Range = "That was yesterday."  
  
End Sub
```

In the following section you will go through the process of using the Object Browser to help you navigate the object hierarchy, and find information for coding the above example.

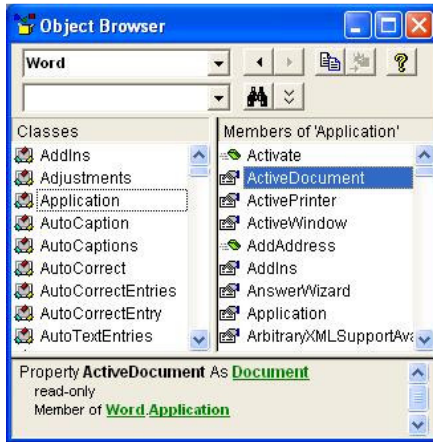
ActiveDocument Object

The ActiveDcoument object is a property of the Application class. You've seen the ActiveDocument object quite a bit and you will see it quite a bit in your future code. You can start using the ActiveDocument object in your code, if you already have an active document open (a document with the focus). Recall that before you can use an object in code it must be an object already in the computer memory. Before you can use an object variable in code it must refer to an object in memory. When you have an active document that is open (with the focus) the ActiveDocument object automatically represents the active document which occupies the computer memory as an object, so you can use the ActiveDocument object in code. If you use the ActiveDocument object in code without having an active document open you'll get a run-time error.

To find the ActiveDocument object in the Object Browser follow these steps.

Steps

- 1) Start Visual Basic Editor and the Object Browser.
- 2) In the Class List window scroll down to the Application class and select it.
- 3) The Member List window now displays all the properties, method and events of the Application class. Select the ActiveDocument property.
- 4) The Description pane shows the description of the ActiveDocument property.

Figure 3.14 *ActiveDocument object*

The description pane has the following description of the ActiveDocument property.

Property ActiveDocument As Document

The first word Property tells you that the ActiveDocument object is a property of the Application class. The (As Document) phrase tells you that the ActiveDocument property is an object of the Document class. Because it is a Document object it has all the properties, methods and events of the Document class. If you need more information of the ActiveDocument property press F1 while the ActiveDocument object is highlighted in the Member List window and Visual Basic Help will give you the information.

Document and Paragraphs Objects

Next you want to look up the properties and methods of the Document object to see which properties and methods can help you add a new paragraph of text to the active document. To look up the properties and methods of the Document class you click the hyperlinked Document in the Description pane (Figure 3.15).

Figure 3.15 *Go to Document class*

Click here to go to the Document class

This selects the Document class in the Class List window and displays the properties, methods and events of the Document class in the Members List window (Figure 3.16). Scroll down the list of properties, methods and events of the Document class in the Members List window to the Paragraphs property (Figure 3.16).

The Description pane shows that the Paragraphs property is of the Paragraphs class. In the following code you declare paras as an object variable of the Paragraphs class and use the Set statement to assign to paras a reference to the ActiveDocument.Paragraphs object which represents all the paragraphs of the active documents.

Dim paras As Paragraphs

Set paras = ActiveDocument.Paragraphs

This assignment using the Set statement is permitted because both paras and the ActiveDocuemt.Paragraphs object are of the same Paragraphs class.

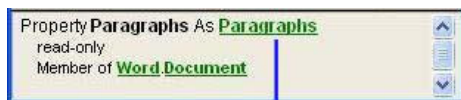
Figure 3.16 Paragraphs object



Add Method

Now that you've paras as a Paragraphs object you want to look up the properties and methods of the Paragraphs class by clicking the hyperlinked Paragraphs in the Description pane (Figure 3.17).

Figure 3.17 Go to Paragraphs class



Click here to go to the Paragraphs class

This selects the Paragraphs class in the Class List window and displays the properties and methods of the Paragraphs class in the Members List window (Figure 3.18). You want to use the Add method (function) to add a new paragraph mark to the document (Figure 3.18).

The Description pane shows that the Add method (a function method) returns a Paragraph object. Bear in mind the Paragraph object and the Paragraphs object are different objects, where the latter is a collection object. You use the Add method of the Paragraphs object to return a different object, a Paragraph object.

The Add method (function) has the following syntax according to the Description pane:

```
Function Add([Range]) As Paragraph
```

The Add method (function) takes on an optional parameter [Range], not to be confused with the Range class. The [Range] parameter specifies the position where you want your new paragraph mark added. If you do not

specify where to add the paragraph mark as in the following code, the paragraph mark is added to the end of the document after the last paragraph.

```
paras.Add
```

Figure 3.18 *Add method*



If you run the following code in a macro, assuming you have only one paragraph in the document and you've turned on the (show) paragraph mark option, you will see a new paragraph mark added at the end of the document (Figure 3.19).

```
Dim paras As Paragraphs
```

```
Set paras = ActiveDocument.Paragraphs
```

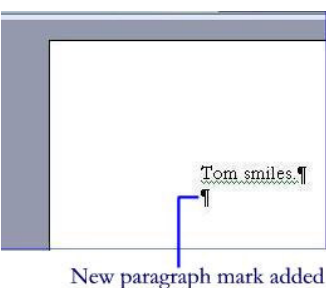
```
paras.Add
```

After you run the above code you have 2 paragraphs in your document, the second paragraph with no text (an empty paragraph) in it (Figure 3.19). You can now use the Item method of the Paragraphs object as in the following code to refer to the second paragraph (second item) of your document.

```
Paras.Item(2)
```

Using the above expression with only one paragraph in the document will give you a run-time error.

Figure 3.19 *Add new paragraph mark*



Item Method

While you still have the Paragraphs class selected in the Class List window (Figure 3.20), scroll down to the Item method in the members List window and select it. The Description pane shows that the Item method (a function method) returns a Paragraph object. Bear in mind the Paragraph object and the Paragraphs object are two different objects, where the latter is a collection object. You use the Item method of the Paragraphs object to return a different object, a Paragraph object.

The Item method (function) has the following syntax according to the Description pane (Figure 3.20).

Function Item(Index As Long) As Paragraph

The Item method (function) requires a parameter (Index As Long), where you use Index (1, 2, 3, ...) to refer to the 1st, 2nd, item (paragraph) of the document.

You use the following expression to refer to the second paragraph.

```
paras.Item(2)
```

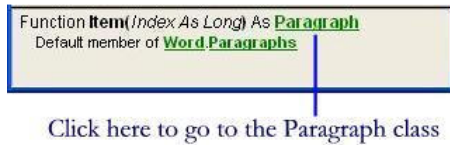
Because the Item method returns a Paragraph object `paras.Item(2)` refers to a Paragraph object that represents the second paragraph of the document.

Figure 3.20 *Item method*



Range Property

Now that you're working with `paras.Item(2)`, a Paragraph object, you want to look up the properties and methods of the Paragraph class by clicking the hyperlinked Paragraph in the Description pane (Figure 3.21).

Figure 3.21 *Go to Paragraph Class*

This selects the Paragraph class in the Class List window and displays the properties and methods of the Paragraph class in the Members List window (Figure 3.22).

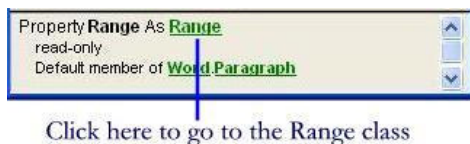
Scroll down to the Range property in the members List window and select it (Figure 3.22). The Description pane shows that the Range property is of the Range class. So the following expression is a Range object:

```
paras.Item(2).Range
```

Figure 3.22 *Range property*

Text Property

Now that you're working with `paras.Item(2).Range`, a Range object, you want to look up the properties and methods of the Range class by clicking the hyperlinked Range in the Description pane (Figure 3.23).

Figure 3.23 *Go to Range class*

This selects the Range class in the Class List window and displays the properties and methods of the Range class in the Members List (Figure 3.24).

Scroll down to the Text property in the members List window and select it (Figure 3.24). The Description pane shows that the Text property is of the String data type. Because the Text property is of the String data type you can assign to it a string expression as in the following expression:

```
paras.Item(2).Range.Text = "That was yesterday."
```

Because the Item method is the default method of the Paragraphs class and the Text property is the default property of the Range class you can drop those two terms and use the following equivalent expression:

```
paras(2).Range = "That was yesterday."
```

Figure 3.24 *Text property*



Recap

You've gone through the process of using the Object Browser to navigate the object hierarchy to help you with your coding of the following:

```
Public Sub defaultPropertyMethod()
```

```
    Dim paras As Paragraphs
```

```
    Set paras = ActiveDocument.Paragraphs
```

```
    paras.Add
```

```
    paras(2).Range = "That was yesterday."
```

```
End Sub
```

This concludes Chapter 3.

Chapter 4 Programming Fundamentals

Some Programming Housekeeping

Option Explicit Statement

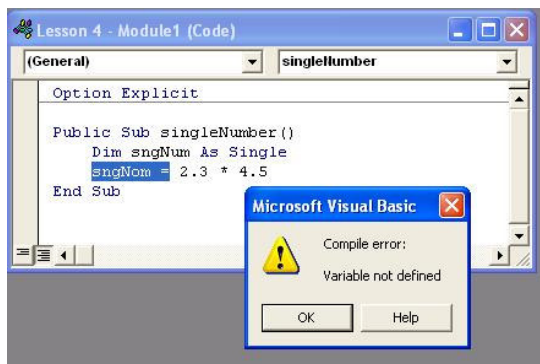
It is a sound programming practice to put in the Option Explicit statement at the top of your module (Figure 4.1). In the presence of the Option Explicit statement you must declare every variable you use in the code or you will get a compile error when you run your code (macro).

The Option Explicit statement is for the debugging and maintainability of your code. Suppose you declare a variable and misspell the name of the variable in one of the many places in the code where you use it. Without the Option Explicit statement the computer will not detect the error in your code and you may get an unintended result which sometimes is difficult to detect and debug.

In the following example you declare sngNum as a variable of the Single data type (more on that later) and then misspell it as sngNom when using it. When you run the macro you will get a Compile Error (figure 4.1).

```
Public Sub singleNumber()  
  
    Dim sngNum As Single  
  
    sngNom = 2.3 * 4.5  
  
End Sub
```

Figure 4.1 *Option Explicit statement*



Continuation of Statement

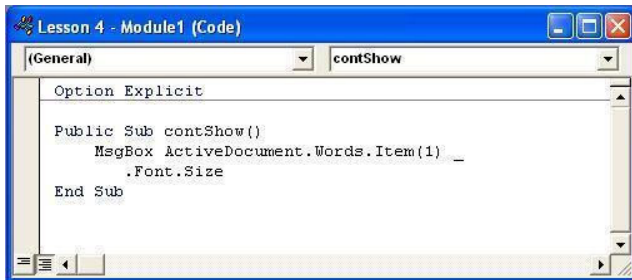
If you need to type long Visual Basic statements you can break the line into multiple lines by terminating the line with a blank space followed by the underscore character `_` before you continue to the next line, such as the following (see also Figure 4.2).

```
MsgBox ActiveDocument.Words.Item(1) _  
    .Font.Size
```

The above statement is equivalent to the following:

```
MsgBox ActiveDocument.Words.Item(1).Font.Size
```

Figure 4.2 *Continuation of statement*

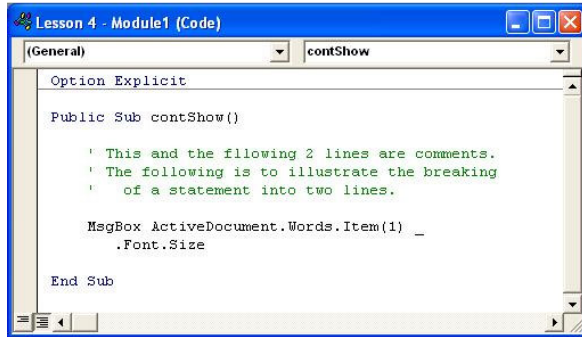


Note

To continue your statement into the next line you must terminate the line with a blank space followed by `_`. The blank space before `_` is necessary for this to work.

Comment Line

A comment line begins with the apostrophe `'` (Figure 4.3). You add comments to your code to explain the working of your code, to add clarity and maintainability of your code. Other programmers can understand your code a lot better if you have comments and documentation of your code.

Figure 4.3 *Comment line*

Variables

You use variable to represent, store and manipulate different types of data. The types of data can be objects, numbers, dates, Boolean expression, strings and other data types. In the following sections you will learn about these different types of data and use them in your code.

Variable Name

Your variable name must begin with an alphabet letter and may not contain spaces. For full detail of rules of naming variable consult Visual Basic Naming Rules of Visual Basic Help.

Though not mandatory, you should name your variables beginning with a non-capitalized alphabet and a prefix as presented in Figure 4.4. You don't capitalize the name of your variables to distinguish them from the Visual Basic keywords which are all capitalized. You use the prefix (Figure 4.4) with the name of your variables to tell what data types the variables are associated with. These naming conventions are meant to increase clarity and maintainability of your code. They are also invaluable in debugging of your code.

Variable names are not case sensitive. For example the variable names `intCash` and `intcash` are the same to Visual Basic. You won't have much opportunity to practice though, because if you type your variable name as `intCash` and later type it as `intcash`, Visual Basic changes the previous `intCash` to `intcash`. If you declare your variable in a `Dim` statement such as `(Dim intCash)` then when you later type the variable as `intcash` Visual Basic changes it to `intCash`.

The naming conventions for object variables are less uniform. In general they should be descriptive. For examples you may use `doc`, `para` and `paras` as object variable names for the Document, Paragraph, and Paragraphs classes respectively.

Figure 4.4 *Variables naming convention*

Data type	Prefix	Example
Boolean	bln	blnFlag
Byte	byt	bytType
Date	dtm	dtmBirth
Double	dbl	dblLarge
Integer	int	intCounter
Long	lng	lngIndex
Single	sng	sngBig
String	str	strName

Variables that Use Numbers

You use variables of the numeric types to represent, store and manipulate different types of numbers. Figure 4.5 shows four types of numbers. Integer and Long types are integers. The Long data type has larger integer values than the Integer data type, and requires more computer memory storage. Single and Double types are decimal numbers for floating point arithmetic computation. The Double data type produces more accuracy (therefore the term double) than the Single data type, and requires more computer memory storage.

The following code declares sngNum as a variable of the Single data type:

```
Dim sngNum as Single
```

Figure 4.5 *Number data types*

Data type	Description
Integer	Integers.
Long	Integers.
Single	Single-precision floating-point numbers.
Double	Double-precision floating-point numbers.

You use arithmetic operator to perform computation on numbers or number variables. Figure 4.6 shows the commonly used arithmetic operators.

The following code assigns to sngNum, a variable of the Single data type, the result of the multiplication of two numbers, 2.3 and 4.5.

```
Dim sngNum As Single
```

```
sngNum = 2.3 * 4.5
```


Figure 4.6 *Arithmetic operator*

Arithmetic Operator	Description
*	Multiply
+	Add
-	Subtract
/	Divide
^	Raise to the power of
Mod	Divide two numbers and return only the remainder

Numeric Variable Example

The following code declares `sngNum` as a variable of the Single data type, and assigns to it the result of multiplying two numbers, 2.3 and 4.5.

```
Dim sngNum As Single
```

```
sngNum = 2.3 * 4.5
```

The following code assigns `sngNum`, which has the result of the computation, to the `Text` property of the `Words(1)` object.

```
ActiveDocument.Words(1).Text = sngNum
```

The following shows the completed code.

Completed Code

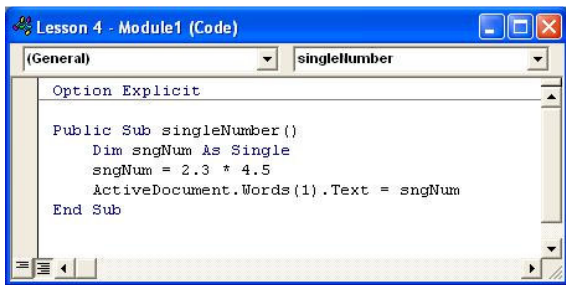
```
Public Sub singleNumber()  
    Dim sngNum As Single  
    sngNum = 2.3 * 4.5  
    ActiveDocument.Words(1).Text = sngNum  
End Sub
```

To try this example follows these steps.

Steps

- 1) Start a blank document and start Visual Basic Editor.
- 2) Insert a new module and keep the default name as Module1.
- 3) Insert a new procedure (macro) and name it singleNumber.
- 4) Type in the above completed code (Figure 4.7).
- 5) Run your macro and see the number 10.35, result of $2.3 * 4.5$, displayed in your document (Figure 4.8).

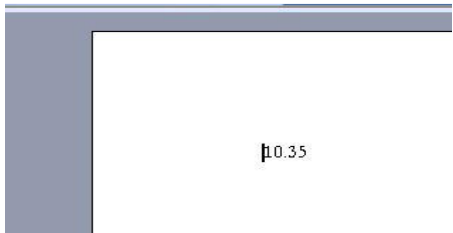
Figure 4.7 *singleNumber macro*



```
Lesson 4 - Module1 (Code)
(General) | singleNumber
Option Explicit

Public Sub singleNumber()
    Dim sngNum As Single
    sngNum = 2.3 * 4.5
    ActiveDocument.Words(1).Text = sngNum
End Sub
```

Figure 4.8 *Result of singleNumber macro*



Variables of the Boolean Data Type

Boolean data type can only be True or False (Figure 4.9). The following code declares blnFlag as a variable of the Boolean data type.

```
Dim blnFlag as Boolean
```

Figure 4.9 Boolean data type

Data type	Description
Boolean	Can only be True or False.

Comparison Operators

A Comparison operator (Figure 4.10) is used to compare two expressions and return a value that is True or False. That is, it returns a result of the Boolean data type. The following code declares `blnFlag` as a variable of the Boolean data type. The comparison `3 > 1` produces the value of True which is assigned to `blnFlag`.

```
Dim blnFlag as Boolean
```

```
blnFlag = (3 > 1)
```

Figure 4.10 Comparison operator

Comparison Operator	Description
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
<>	Not equal to

Comparison Operator Example

The following code declares `blnFlag` as a variable of the Boolean data type.

```
Dim blnFlag as Boolean
```

The following tests to see if the font size of the first word is less than 14. If it is, `blnFlag = True`. If it is not, `blnFlag = False`.

```
blnFlag = (ActiveDocument.Words(1).Font.Size < 14)
```

The following code sets the first word to bold if the font size of the first word is less than 14.

```
ActiveDocument.Words(1).Bold = blnFlag
```

The following shows the completed code.

```
Completed Code
```

```
Public Sub blnCompare()
```

```
Dim blnFlag As Boolean
blnFlag = (ActiveDocument.Words(1).Font.Size < 14)
ActiveDocument.Words(1).Bold = blnFlag
End Sub
```

To try this example follows these steps.

Steps

- 1) Start the Visual Basic Editor.
- 2) Use the same module as the one in the previous example.
- 3) Insert a new procedure (macro) and name it `blnCompare`.
- 4) Type in the above completed code (Figure 4.11).
- 5) If the document is blank, type in a sentence as that in Figure 4.12. If your document is not blank, make sure the font size of the first word is less than 14.
- 6) Run your macro and see the first word set to bold (Figure 4.12).

Figure 4.11 *blnCompare* macro

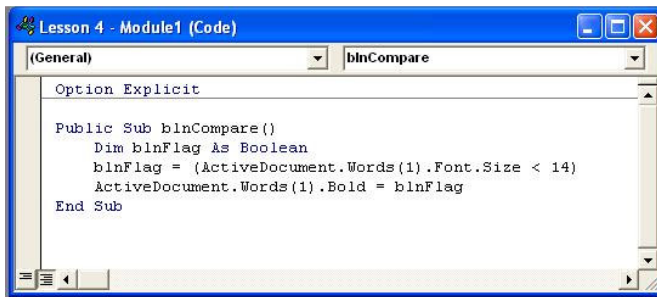


Figure 4.12 Result of *blnCompare* macro before and after



Logical Operators

You use logical operators (Figure 4.13) to perform logical operations on data or variables that are of the Boolean data type. Figure 4.14 shows the results of logical operators And and Or. The logical operator Not inverts whatever value it is used with.

The two following expressions both return a True value.

$(3 > 1)$

$(4 > 2)$

The following expression returns a True value.

$(3 > 1) \text{ And } (4 > 2)$

The following expression returns a False value.

$(3 > 1) \text{ And } (2 > 4)$

The following expression returns a True value.

$(3 > 1) \text{ Or } (2 > 4)$

The following expression returns a True value.

Not $(2 > 4)$

Figure 4.13 *Logical operator*

Logical Operator	Description
And	Logical And
Or	Logical Or
Not	Logical Not

Figure 4.14 *Result of And and Or*

And	True	False
True	True	False
False	False	False

Or	True	False
True	True	True
False	True	False

Logical Operator Example

The following code declares `blnFlag` as a variable of the Boolean data type.

```
Dim blnFlag as Boolean
```

The following tests to see if the font size of the first word is less than 14 and if there are less than 10 paragraphs in the document. If both of the conditions are True, then `blnFlag = True`. If not then `blnFlag = False`.

```
blnFlag blnFlag = (ActiveDocument.Words(1).Font.Size < 14) _  
    And (ActiveDocument.Paragraphs.Count < 10)
```

The following code format the first word to bold if `blnFlag = True`.

```
ActiveDocument.Words(1).Bold = blnFlag
```

The following shows the completed code.

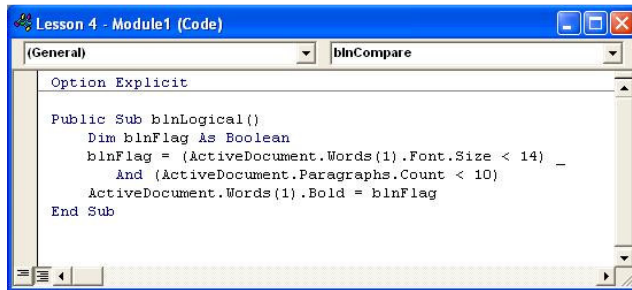
Completed Code

```
Public Sub blnLogical()  
    Dim blnFlag As Boolean  
    blnFlag = (ActiveDocument.Words(1).Font.Size < 14) _  
        And (ActiveDocument.Paragraphs.Count < 10)  
    ActiveDocument.Words(1).Bold = blnFlag  
End Sub
```

To try this example follows these steps.

Steps

- 1) Start the Visual Basic Editor.
- 2) Use the same module as the one in the previous example.
- 3) Insert a new procedure (macro) and name it `blnLogical`.
- 4) Type in the above code (Figure 4.14).
- 5) If the document is blank, type in a sentence as that in Figure 4.16. If your document is not blank, make sure the font size of the first word is less than 14. Also make sure there are less than 10 paragraphs in the document.
- 6) Run your macro and see the first word set to bold (Figure 4.16).

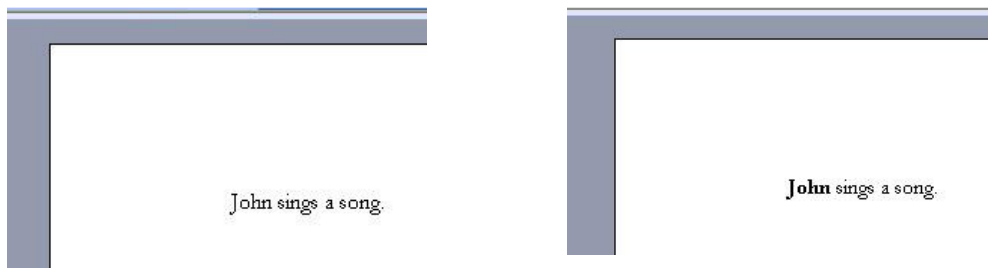
Figure 4.15 *blnLogical* macro


```

Option Explicit

Public Sub blnLogical()
    Dim blnFlag As Boolean
    blnFlag = (ActiveDocument.Words(1).Font.Size < 14) _
        And (ActiveDocument.Paragraphs.Count < 10)
    ActiveDocument.Words(1).Bold = blnFlag
End Sub

```

Figure 4.16 Result of *blnLogical* macro before and after

Variables that Use Strings

The string data type consists of a sequence of contiguous characters that represent the characters themselves rather than their numeric values. A string can include letters, numbers, spaces, and punctuation. You use variable of the string data type to represent, store and manipulate characters or texts.

The following code declares strName as a variable of the string data type and assigns to it the text John.

```
Dim strName as String
```

```
strName = "John"
```

Figure 4.17 *String data type*

Data type	Description
String	Contains characters or text.

Concatenate Operator &

You use the concatenate operator & to concatenate or join two or more strings or variables of the string data type. The following code declares strWord as a variable of the string data type, joins two strings, Hello and

(World) to form the string “Hello World” and store it in strWord. As a result strWord contains the string “Hello World”.

```
Dim strWord As String
```

```
strWord = "Hello" & " World"
```

Figure 4.18 *Concatenate operator*

String Operator	Description
&	Concatenate

String Data Type Example

The following code declares strWord as a variable of the string data type.

```
Dim strWord As String
```

The following code stores the second word of the document in strWord.

```
strWord = ActiveDocument.Words(2)
```

The following code insert <text> before the text in strWord and append </text> after the text in strWord, and stores the resulting string in strWord.

```
strWord = "<text>" & strWord & "</text>"
```

The following code replaces the second word of the document by the content of strWord.

```
ActiveDocument.Words(2) = strWord
```

The following shows the completed code.

Completed Code

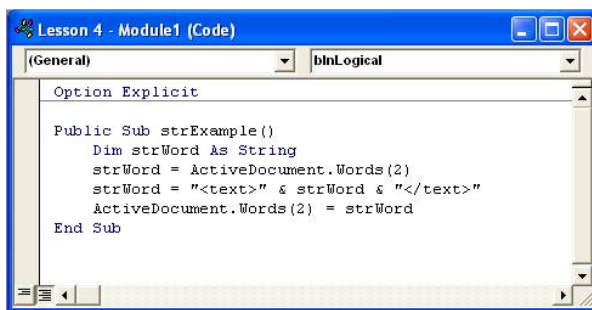
```
Public Sub strExample()  
    Dim strWord As String  
    strWord = ActiveDocument.Words(2)  
    strWord = "<text>" & strWord & "</text>"  
    ActiveDocument.Words(2) = strWord  
End Sub
```


To try this example follows these steps.

Steps

- 1) Start the Visual Basic Editor.
- 2) Use the same module as the one in the previous example.
- 3) Insert a new procedure (macro) and name it `strExample`.
- 4) Type in the above code (Figure 4.19).
- 5) If the document is blank, type in a sentence as that in Figure 4.20.
- 6) Run your macro and see the second word of the document, `sings`, replaced by `<text>sings </text>`.

Figure 4.19 *strExample* macro



```
Lesson 4 - Module1 (Code)
(General) | btnLogical
Option Explicit

Public Sub strExample()
    Dim strWord As String
    strWord = ActiveDocument.Words(2)
    strWord = "<text>" & strWord & "</text>"
    ActiveDocument.Words(2) = strWord
End Sub
```

Figure 4.20 *Result of strExample macro before and after*



Variables that Use Dates

The `date` data type or variable of the date data type represents dates. The following code declares `dtmBirth` as a variable of the Date data type and assigns to it the date `#10/6/2004#`.

```
Dim dtmBirth as Date
```

```
dtmBirth = #10/1/2004#
```

Any recognizable date values can be assigned to a variable of the Date data type. The date vales must be enclosed within number signs `#`. The following statements are all equivalent.

```
dtmBirth = #10/1/2004#
```

```
dtmBirth = #October 1, 2004#
```

```
dtmBirth = #1 Oct 2004#
```

Figure 4.21 *Date data type*

Data type	Description
Date	Floating-point numbers that represent dates.

Date Data Type Example

The following code declares `dtmBirth` as a variable of the Date data type and `strText` as a variable of the String data type.

```
Dim dtmBirth As Date
```

```
Dim strText As String
```

The following code assigns a date to `dtmBirth`.

```
dtmBirth = #10/1/1970#
```

The following code joins three string expressions and stores the result in `strText`.

```
strText = "Your birth date is " & dtmBirth & "!"
```

The following code writes the content of `strText` as the first sentence of a blank document.

```
ActiveDocument.Sentences(1) = strText
```

The following shows the completed code.

Completed Code

```
Public Sub dateExample()  
    Dim dtmBirth As Date  
    Dim strText As String  
    dtmBirth = #10/1/1970#  
    strText = "Your birth date is " & dtmBirth & "!"  
    ActiveDocument.Sentences(1) = strText  
End Sub
```

To try this example follows these steps.

Steps

- 1) Start the Visual Basic Editor.
- 2) Use the same module as the one in the previous example.
- 3) Insert a new procedure (macro) and name it dateExample.
- 4) Type in the above code (Figure 4.22).
- 5) Make sure the document is blank.
- 6) Run your macro and see the display of the text (Your birth date is 10/1/1970!) (Figure 4.23).

Figure 4.22 *dateExample* macro

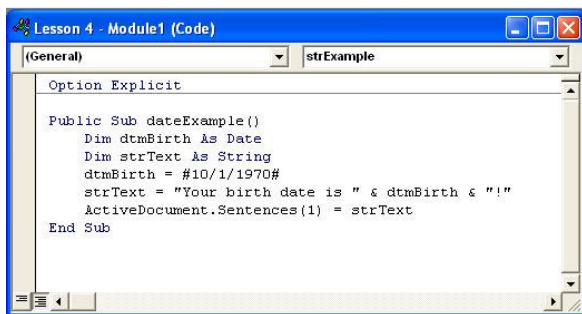
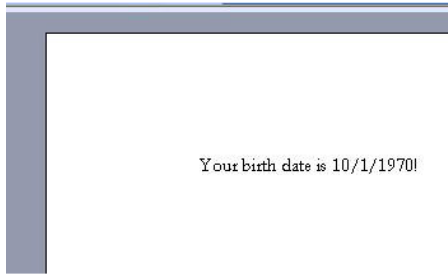


Figure 4.23 *Result of dateExample macro*



Conditional Statement

You use conditional statements to control the flow of your program's execution. Visual Basic provides the If...Then...Else conditional statement to do that. There are two kinds of If...Then...Else statements:

If...Then...Else statement (sing-line form)

If...Then...Else statement (block form)

If...Then...Else Statement (Single-Line Form)

The If...Then...Else single-line form of statement is used to execute one or more lines of code if certain condition is true. Figure 4.24 shows the syntax of the If...Then...Else statement. In syntax, items inside square brackets [] are optional, so in this syntax [Else statements] are optional. Statements can be one or more lines of statements separated by colons.

Figure 4.24

```
If condition Then statements [ Else statements ]
```

If...Then...Else Example 1

This example is similar to an earlier one in this chapter except you implement the same logic using the If...Then statement in one line of code. You test to see if the font size of the first word is less than 14. If it is then you set the first word to bold.

The following shows the completed code.

Completed Code

```
Public Sub ifThen()  
    If (ActiveDocument.Words(1).Font.Size < 14) Then _  
        ActiveDocument.Words(1).Bold = True  
End Sub
```

To try this example follows these steps.

Steps

- 1) Start the Visual Basic Editor.
- 2) Use the same module as the one in the previous example.
- 3) Insert a new procedure (macro) and name it ifThen.
- 4) Type in the above code (Figure 4.25).
- 5) If the document is blank, type in a sentence as that in Figure 4.26. If your document is not blank, make sure the font size of the first word is less than 14.
- 6) Run your macro and see the first word set to bold (Figure 4.26).

Figure 4.25 *ifThen macro*

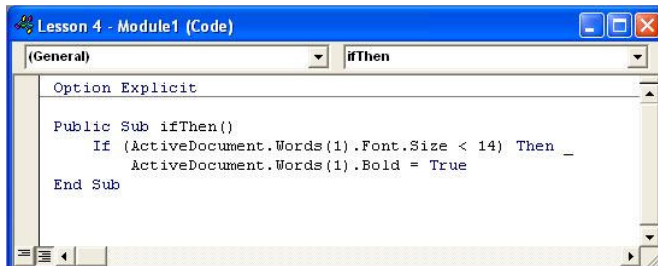
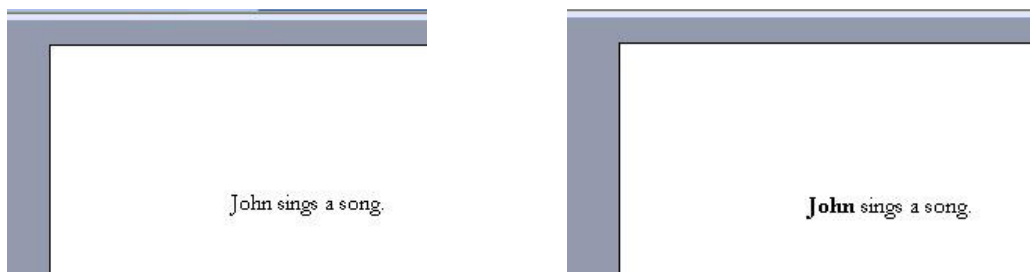


Figure 4.26 *Result of ifThen macro before and after*



If...Then...Else (Block Form) Statement

The If...Then...Else block form of statement executes one or more lines of code if certain condition is true. If the condition is false it executes another line or lines of code. Figure 4.27 shows the syntax of the If...Then...Else block form of statement. Statements can be one or more lines of statements and they don't have to be separated by colons.

Figure 4.27

```
If condition Then  
    [ statements ]  
[ Else ]  
    [ statements ]  
End If
```

If...Then...Else Example 2

The following code tests if the font size of the first word is less than 14

```
If (ActiveDocument.Words(1).Font.Size < 14) Then
```

If the above is true the following code inserts [before the first word and append] after the first word.

```
ActiveDocument.Words(1) = "[ " & ActiveDocument.Words(1) & "]" "
```

The following code makes the second word (was the first word before the last statement was executed) bold.

```
ActiveDocument.Words(2).Bold = True
```

The following is the completed code.

Completed Code

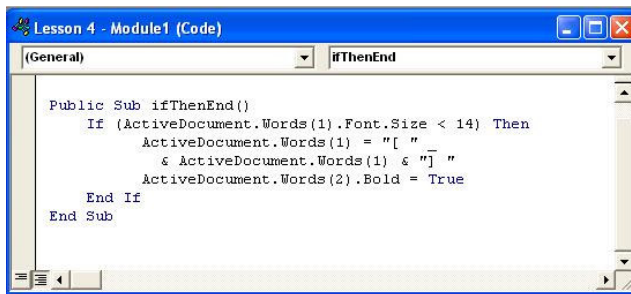
```
Public Sub ifThenEnd()  
    If (ActiveDocument.Words(1).Font.Size < 14) Then  
        ActiveDocument.Words(1) = "[ " _  
            & ActiveDocument.Words(1) & "]" "  
        ActiveDocument.Words(2).Bold = True  
    End If  
End Sub
```

To try this example follows these steps.

Steps

- 1) Start the Visual Basic Editor.
- 2) Use the same module as the one in the previous example.
- 3) Insert a new procedure (macro) and name it `ifThenEnd`.
- 4) Type in the above code (Figure 4.28).
- 5) If the document is blank, type in a sentence as that in Figure 4.29. If your document is not blank, make sure the font size of the first word is less than 14.
- 6) Run your macro and see the first word set to bold with the insertion of `[` and `]` before and after the word (Figure 4.29).

Figure 4.28 *ifThenEnd macro*



```

Lesson 4 - Module1 (Code)
(General) | ifThenEnd
Public Sub ifThenEnd()
    If (ActiveDocument.Words(1).Font.Size < 14) Then
        ActiveDocument.Words(1) = "[ " &
            & ActiveDocument.Words(1) & "]"
        ActiveDocument.Words(2).Bold = True
    End If
End Sub
  
```

Figure 4.29 *Result of ifThenEnd macro before and after*



If...Then...Else Example 3

The following code tests if the font size of the first word is less than 16.

```
If (ActiveDocument.Words(1).Font.Size < 16) Then
```

If it is the following code sets the font size of the first word to 16.

```
ActiveDocument.Words(1).Font.Size = 16
```

If not the following code increases the original font size of the first word by 4.

```
ActiveDocument.Words(1).Font.Size = ActiveDocument.Words(1).Font.Size + 4
```

The following is the completed code.

Completed Code

```
Public Sub ifThenElse()  
    If (ActiveDocument.Words(1).Font.Size < 16) Then  
        ActiveDocument.Words(1).Font.Size = 16  
    Else  
        ActiveDocument.Words(1).Font.Size = _  
            ActiveDocument.Words(1).Font.Size + 4  
    End If  
End Sub
```

To try this example follows these steps.

Steps

- 1) Start the Visual Basic Editor.
- 2) Use the same module as the one in the previous example.
- 3) Insert a new procedure (macro) and name it `ifThenElse`.
- 4) Type in the above code (Figure 4.30).
- 5) If the document is blank, type in a sentence as that in Figure 4.31. If your document is not blank, make sure the font size of the first word is less than 14.
- 6) Run your macro and you will see the font size of the first word changed to 16 (Figure 4.31)
- 7) Run the macro again and you will see the font size changed to 20 (16 + 4) (Figure 4.32).

Figure 4.30 *ifThenElse* macro

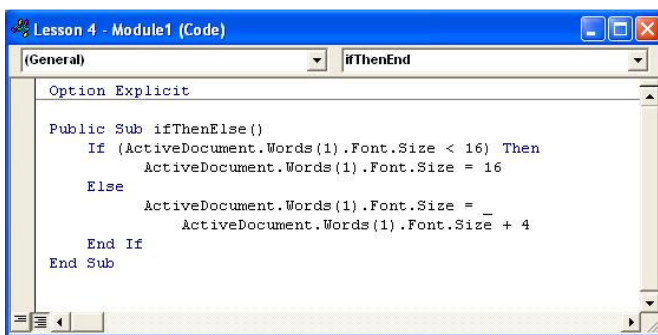


Figure 4.31 *Result of ifThenElse macro before and after*

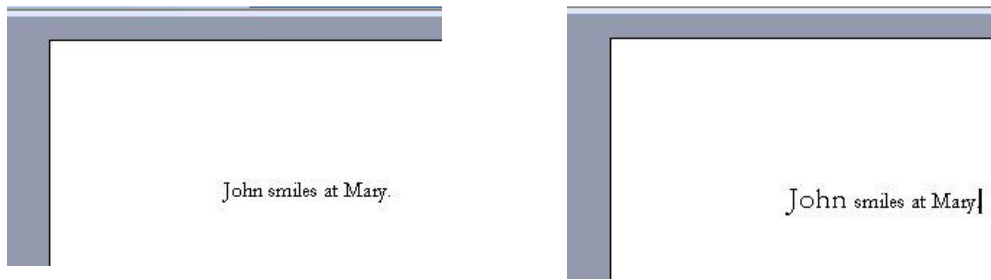
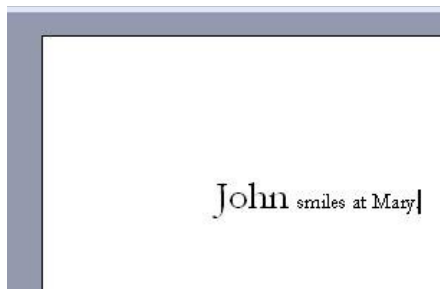


Figure 4.32 *Result of ifThenElse macro run again*



This concludes Chapter 4.

Chapter 5 Programming Loop & Other

Why Use Loop?

Loop in Visual Basic provides a mean to perform the same task or operation repetitively. You will learn to use loop in your code in this chapter.

For...Next Statement

Figure 5.3 shows the syntax of the For...Next statement. The For...Next loop repeats one or more statements a specified number of times until an exit condition occurs. The exit condition occurs when (counter > end) or when the (Exit For) statement is encountered, whichever comes first. Recall that in syntax, items inside the square brackets [] are optional. Statements are optional and can be one or more lines of statements.

Figure 5.1 *Syntax of For...Next statement*

```

For counter = start To end [ Step increment ]
    [ statements ]
    [ Exit For ]
    [ statements ]
Next [ counter ]

```

The following code repeat a statement 5 times with intIndex = 1, 2, 3, 4, 5. It sets the font size of the 1st word, 2nd word, 3rd word, 4th word and 5th word to size 12 (= 10 + 1 * 2), 14 (= 10 + 2 * 2), 16 (= 10 + 3 * 2), 18 (= 10 + 4 * 2) and 20 (= 10 + 5 * 2) respectively. In this loop the exit condition is (intIndex > 5).

```

For intIndex = 1 To 5
    doc.Words(intIndex).Font.Size = 10 + intIndex * 2
Next

```

In the following example an additional exit condition is imposed by the (Exit For) statement.

```

For intIndex = 1 To ActiveDocument.Words.Count
    ActiveDocument.Words(intIndex).Font.Size = 10 + intIndex * 2
    If (ActiveDocument.Words(intIndex).Font.Size > 20) Then Exit For

```

Next

In the above loop the exit condition is

```
intIndex > ActiveDocument.Words.Count
```

or

```
ActiveDocument.Words(intIndex).Font.Size > 20
```

whichever condition comes first.

For...Next Loop Example

The following example uses the For...Next loop to set the font size of the first 5 words of the document incrementally to 12, 14, 16, 18 and 20 respectively.

The following code declares blnFlag as a variable of the Boolean data type.

```
Dim blnFlag as Boolean
```

The following code tests to see if the font size of the first word is less than 14. If it is, blnFlag = True. If it is not, blnFlag = False.

```
blnFlag = (ActiveDocument.Words(1).Font.Size < 14)
```

The following code sets the first word to bold if the font size of the first word is less than 14.

```
ActiveDocument.Words(1).Bold = blnFlag
```

The following shows the completed code.

Completed Code

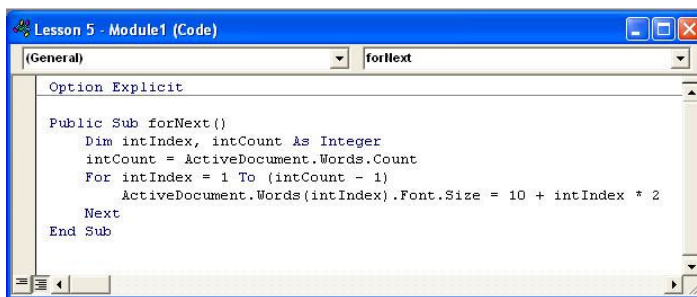
```
Public Sub forNext()  
    Dim intIndex, intCount As Integer  
    intCount = ActiveDocument.Words.Count  
    For intIndex = 1 To (intCount - 1)  
        ActiveDocument.Words(intIndex).Font.Size = 10 + intIndex * 2  
    Next  
End Sub
```

To try this example follows these steps.

Steps

- 1) Start a blank document and start Visual Basic Editor.
- 2) Insert a new module and keep the default name as Module1.
- 3) Insert a new procedure (macro) and name it forNextMacro.
- 4) Type in the above code (Figure 5.2).
- 5) Type a sentence as that in Figure 5.3. Run the macro and see the font size of the first 5 words of the document set incrementally to 12, 14, 16, 18 and 20 respectively.

Figure 5.2 *forNext macro*

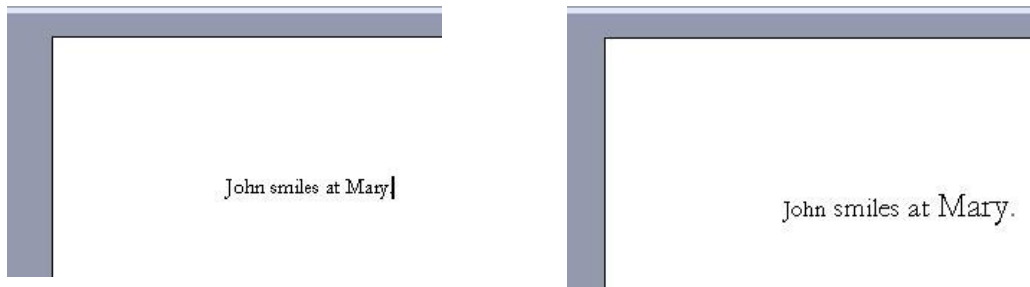


```

Lesson 5 - Module1 (Code)
(forNext)
Option Explicit

Public Sub forNext()
    Dim intIndex, intCount As Integer
    intCount = ActiveDocument.Words.Count
    For intIndex = 1 To (intCount - 1)
        ActiveDocument.Words(intIndex).Font.Size = 10 + intIndex * 2
    Next
End Sub
  
```

Figure 5.3 *Result of forNext macro before and after*



For Each...Next Statement

Figure 5.4 shows the syntax of the For Each...Next statement. The For Each...Next loop repeats one or more statements for each element in a group (collection) of objects. The loop exits when all the members of the collection object are exhausted or when the (Exit For) statement is encountered, whichever comes first. Statements are optional and can be one or more lines of statements.

Figure 5.4

```
For Each element In group
    [ statements ]
    [ Exit For ]
    [ statements ]
Next [ element ]
```

The following example iterates through every paragraph in the active document, and underlines the second character of each paragraph if the first character is the letter T. The loop exits when there are no more paragraphs to process.

```
Dim para As Paragraph
For Each para In ActiveDocument.Paragraphs
    If para.Range.Characters(1) = "T" Then
        para.Range.Characters(2).Font.Underline = True
    End If
Next
```

In the following example an additional exit condition is imposed by the (Exit For) statement. The loop iterates through every paragraph in the active document to search for the text Tom as the first word of the paragraph. If the search is successful, that is, if the text Tom is found as the first word of any paragraph the loop exits. If the search is unsuccessful, the loop exits after searching for every paragraph of the document. The Visual Basic function (Trim) is used to trim off any leading or trailing spaces of the words being searched for.

```
Dim para As Paragraph
Dim blnFound
blnFound = False
For Each para In ActiveDocument.Paragraphs
    If (Trim(para.Range.Words(1)) = "Tom") Then
        blnFound = True
        Exit For
    End If
Next
```

For...Each...Next Loop Example 1

This example underlines the 2nd character of every paragraph of the document if the 1st character of the paragraph is T. It does this by looping through each paragraph in the document. If the 1st character of the paragraph is T the 2nd character of the same paragraph is underlined.

The following shows the completed code.

Completed Code

```
Public Sub forEachLoop()
    Dim para As Paragraph
    For Each para In ActiveDocument.Paragraphs
        If para.Range.Characters(1) = "T" Then
            para.Range.Characters(2).Font.Underline = True
        End If
    Next
End Sub
```

To try this example follows these steps.

Steps

- 1) Start the Visual Basic Editor.
- 2) Use the same module as the one in the previous example.
- 3) Insert a new procedure (macro) and name it forEachLoop.
- 4) Type in the above completed code (Figure 5.5).
- 5) Type some sentence in your document like that in Figure 5.6. Run the macro and you will see the 2nd character underlined for every paragraph of the document if the 1st character of the paragraph is T (Figure 5.6).

Figure 5.5 *forEachLoop* macro

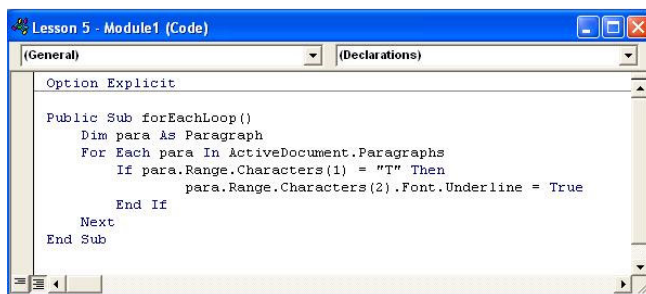
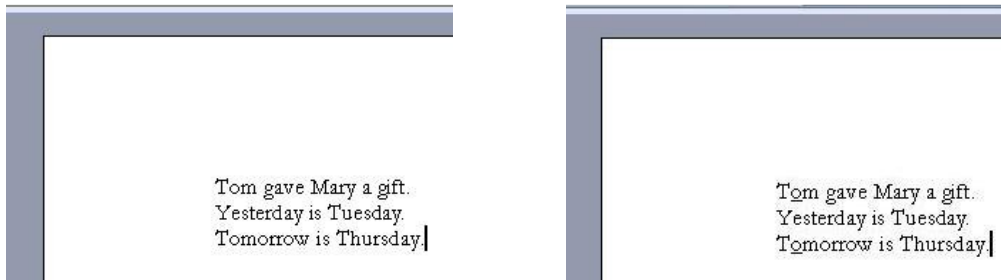


Figure 5.6 *Result of forEachLoop macro before and after*



For...Each...Next Loop Example 2

The following example sets the font size of each character in the document to 30 if the character is either o or a. You do this by using two loops. The first loop loops through each paragraph in the document. Within the first loop (that is, within each paragraph) it loops through every character and set its font size to 30 if the character is either a or o.

The following shows the completed code.

Completed Code

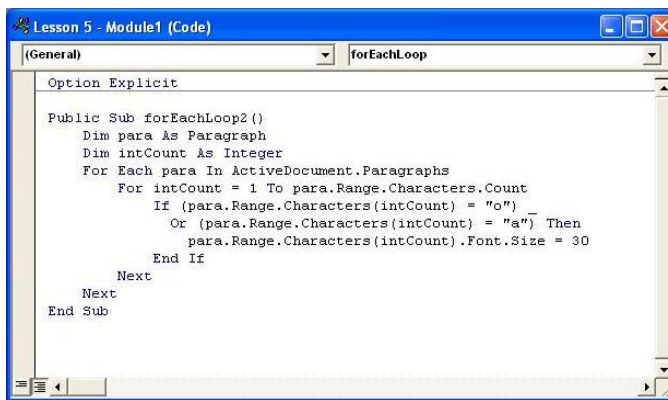
```
Public Sub forEachLoop2()  
    Dim para As Paragraph  
    Dim intCount As Integer  
    For Each para In ActiveDocument.Paragraphs  
        For intCount = 1 To para.Range.Characters.Count  
            If (para.Range.Characters(intCount) = "o") _  
                Or (para.Range.Characters(intCount) = "a") Then  
                para.Range.Characters(intCount).Font.Size = 30  
            End If  
        Next  
    Next  
End Sub
```


To try this example follows these steps.

Steps

- 1) Start the Visual Basic Editor.
- 2) Use the same module as the one in the previous example.
- 3) Insert a new procedure (macro) and name it forEachLoop2.
- 4) Type in the above completed code (Figure 5.7).
- 5) If the document is blank, type in a sentence as that in Figure 5.8.
- 6) Run the macro and you will see the font size of every character which is either a or o, set to 30 (Figure 5.8).

Figure 5.7 *forEachLoop2* macro

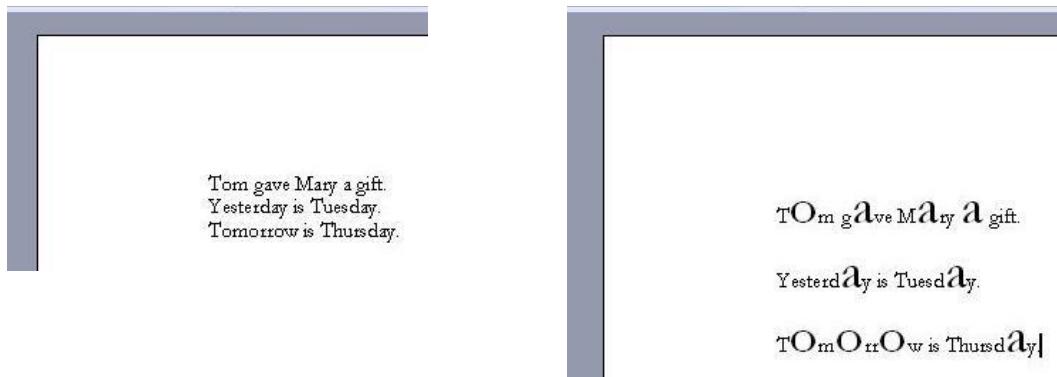


```

Lesson 5 - Module1 (Code)
|General| forEachLoop
Option Explicit

Public Sub forEachLoop2()
    Dim para As Paragraph
    Dim intCount As Integer
    For Each para In ActiveDocument.Paragraphs
        For intCount = 1 To para.Range.Characters.Count
            If (para.Range.Characters(intCount) = "o") _
                Or (para.Range.Characters(intCount) = "a") Then
                para.Range.Characters(intCount).Font.Size = 30
            End If
        Next
    Next
End Sub
  
```

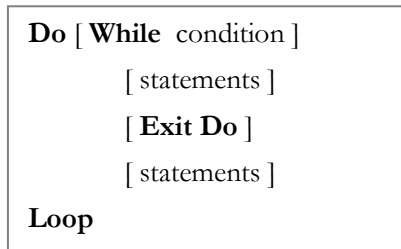
Figure 5.8 Result of *forEachLoop2* macro before and after



Do...Loop Statement

Figure 5.9 shows the syntax of the Do...Loop statement. The Do...Loop statement repeats a block of statements while a condition (While condition) is true or when the (Exit Do) statement is encountered. If both the (While condition) clause and the (Exit Do) statement are omitted in the Do...Loop statement the iteration continues indefinitely. Statements are optional and can be one or more lines of statements.

Figure 5.9



The following example uses the Do...Loop to calculate the sum of $1 + 2 + \dots + 100 = 5050$. The loop exits when `intCount > 100`, which occurs when the condition (`While intCount <= 100`) is violated.

```
Dim intCount, intSum As Integer  
  
intCount = 1  
  
intSum = 0  
  
Do While intCount <= 100  
    intSum = intSum + intCount  
    intCount = intCount + 1  
  
Loop
```

The following example uses another variation of the Do...Loop to calculate the same sum of $1 + 2 + \dots + 100 = 5050$. The loop exits when `intCount > 100`, which occurs when the (`Exit Do`) statement is executed.

```
Dim intCount, intSum As Integer  
  
intCount = 1  
  
intSum = 0  
  
Do  
    intSum = intSum + intCount  
    intCount = intCount + 1  
    If (intCount > 100) Then Exit Do  
  
Loop
```

Use of the Do...Loop Statement

Why we have so many loops structure?

The Do...Loop statement is extremely versatile of which the For...Next statement is a special case. That is, every For...Next loop can be implemented using the Do...Loop loop but not vice versa. The For...Next loop takes less code and is simple to implement. You use the For...Next loop to solve most of the iteration problem you encounter in programming and save the situation to use the Do...Loop loop when the logic is more complicated in the iteration, or when it is next to impossible to do the iteration using the other method.

An example is when you write an event handler to perform some operation when an event occurs. In this situation you use the Do...Loop to repetitively check to see if certain event has occurred. If the event has occurred then certain operations get performed.

With Statement

The With statement allows you to write a series of statements on an object without specifying the name of the object in each of the statement. Figure 5.10 shows the syntax of the With statement. Statements are optional and can be one or more lines of statements that use the properties and methods of the object.

Figure 5.10

<pre> With object [statements] End With </pre>
--

The following code sets the InsideLineStyle, InsideLineWidth, OutsideLineStyle and OutsideLineWidth properties of the Borders object, table2.Borders.

```

Dim table2 as Table

table2.Borders.InsideLineStyle = wdLineStyleSingle

table2.Borders.InsideLineWidth = wdLineWidth100pt

table2.Borders.OutsideLineStyle = wdLineStyleSingle

table2.Borders.OutsideLineWidth = wdLineWidth100pt

```

The following is equivalent to the above code using the With statement.

```

Dim table2 as Table

With table2.Borders

    .InsideLineStyle = wdLineStyleSingle

    .InsideLineWidth = wdLineWidth100pt

```

```
.OutsideLineStyle = wdLineStyleSingle  
.OutsideLineWidth = wdLineWidth100pt
```

End With

Function Procedure

There are four types of procedure in Visual Basic, Sub procedure, function procedure, property procedure and event procedure and you've seen examples of Sub procedure and event procedure. A function procedure is similar to a Sub procedure, but unlike a Sub procedure which returns nothing, a function procedure returns a value or an object. A function procedure may take arguments that are passed to it by a calling procedure. A calling procedure can be any of the four types of procedure, Sub, function, property or event procedures. A function returns a value or an object by assigning a value or an object to its name in one or more statements of the function procedure.

The following example shows max, a function procedure that takes two arguments (numbers) num1 and num2 and returns the greater of the two numbers to main, the calling procedure which is a Sub procedure.

Completed Code

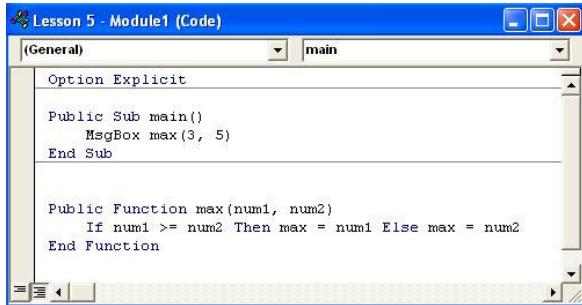
```
Public Sub main()  
    MsgBox max(3, 5)  
End Sub
```

```
Public Function max(num1, num2)  
    If num1 >= num2 Then max = num1 Else max = num2  
End Function
```

To try this example follows these steps.

Steps

- 1) Start the Visual Basic Editor.
- 2) Use the same module as the one in the previous example.
- 3) Insert a new procedure (macro) and name it main
- 4) Insert a new function and name it max.
- 5) Type in the above completed code (Figure 5.11).
- 6) Run the macro and you will see the dialog box displaying the number 5 (greater of 3 and 5) (Figure 5.12).

Figure 5.11 *Function procedure example*Figure 5.12 *Result of function procedure*

What is an Array?

An Array is a collection of data using the same variable name and of the same data type. You refer to each element of the array by the subscripts 0, 1, 2, and so on.

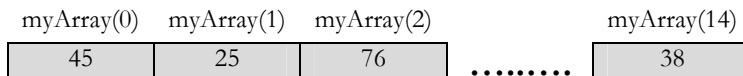
The following code declares `intArray` as an array of the Integer data type with 15 elements.

‘ Declare an array of integers with 15 items

```
Dim intArray(15) As Integer
```

Visual Basic uses subscripts starting from 0, so `intArray(0)` refers to the 1st item of the array, `intArray(1)` refers to the 2nd item, and `intArray(14)` refers to the 15th item, the last item of the array. Figure 5.13 shows the layout of the array.

Figure 5.13



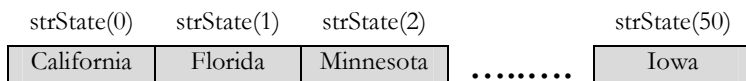
The following code declares `strState` as an array of the `String` data type with 51 items to store the names of the 51 states.

```
' Declare an array of strings with 51 items
```

```
Dim strState(51) As String
```

Figure 5.14 shows the layout of the array.

Figure 5.14



Note

All items in an array must be of the same data type.

Table Programming with Array Example

This example parses (examines) a paragraph of words separated by the delimiters “;” or “.”, and converts the words in the paragraph into a table with one word per row without the delimiters “;” or “.”.

The following code declares `objTable` as an object variable of the `Table` class.

```
Dim objTable As Table
```

The following code declares `intIndex` and `intIndex2` as variables of the `Integer` data type. You will use them in a loop later in the program.

```
Dim intIndex, intIndex2 As Integer
```

The following code adds a new paragraph mark in the document.

```
ActiveDocument.Paragraphs.Add
```

The following code selects the new paragraph mark added by the above code.

```
ActiveDocument.Paragraphs.Last.Range.Select
```

The above code is to position your selection in the last paragraph mark at which position to create a new table. You've added this last paragraph mark which contains nothing yet (an empty paragraph). So when you are adding a new table later you are adding the new table at the end of the document. All these positioning and maneuvering are to make sure the new table you are going to create does not overwrite or replace any existing paragraph in your document.

Note

The Select method is a method of the Range class. It is a sub procedure method; therefore it does not return any value or another object. What does it do? It selects the parent object which invokes the Select method. From here on any Selection object or object variable of the Selection class refers to this particular object. That is, any subsequent use of a Selection object or object variable of the Selection class is the same as using this particular object.

The following code creates a new table of 12 rows and 1 column and assigns to objTable (an object variable of the Table class which you've declared earlier) the reference to the new table just created. You create the table by using the Add method (a function method) of the Tables class. From hereon objTable represents the table you've just created.

```
Set objTable = ActiveDocument.Tables.Add _
    (Range:=Selection.Range, NumRows:=12, NumColumns:=1)
```

The Add method is a function method that takes some required parameters and some optional parameters. The Range parameter is required and must be a reference to an object of the Range class. You use the Range parameter to tell the Add method where to put the new table. Recall you've used the Select method to select the last paragraph mark of the document, as a result Selection.Range refers to the last paragraph mark (which contains a empty paragraph) of the document and the new table will be created there.

The NumRows and NumColumns parameters tell the Add method the numbers of row and column of the table to be created. In this case you are creating a table of 12 rows and 1 column. The Range, NumRows and NumColumns parameters are required parameters. That means you must supply them when you use the Add method of the Tables class.

The following code sets the ColumnWidth, RulerStyle, InsideLineStyle, .InsideLineWidth, OutsideLineStyle, OutsideLineWidth properties of the table. You use the With statement to save some typing of the name of the objects of which you are using the methods or properties. Again you use the combination of the Select method and the Selection object to do this. See the earlier discussion on using the Select method and Selection object on doing similar task.

```
With objTable
    .Select
    With Selection
        'Set column width to 1 inch
        .Columns.SetWidth _
            ColumnWidth:=InchesToPoints(1#), _
            RulerStyle:=wdAdjustNone
    With .Borders
        .InsideLineStyle = wdLineStyleSingle
        .InsideLineWidth = wdLineWidth100pt
        .OutsideLineStyle = wdLineStyleSingle
        .OutsideLineWidth = wdLineWidth100pt
    End With
    End With
End With
```

End With

End With

End With

The following code uses the For...Next loop to parse (examine) the first paragraph of the document and put the words of the paragraph into a table with one word per row, discarding the delimiters “;” or “.”.

```
intIndex2 = 1

For intIndex = 1 To (ActiveDocument.Paragraphs(1).Range.Words.Count - 1)

    If (Trim(ActiveDocument.Paragraphs(1).Range.Words(intIndex)) <> ";") And _
        (Trim(ActiveDocument.Paragraphs(1).Range.Words(intIndex)) <> ".") Then

        objTable.Cell(intIndex2, 1).Range.Text = _

            ActiveDocument.Paragraphs(1).Range.Words(intIndex)

        intIndex2 = intIndex2 + 1

    End If

Next
```

You use two counters, intIndex and intIndex2 in this loop. The counter intIndex is used to loop through every word in the first paragraph. You use counter intIndex2 to keep track of the words you put into the table. You have to use a different counter for this purpose because there are delimiters “;” or “.” (each considered as one word) that you don’t want to put into the table. The following expression gives the total number of words in the first paragraph.

```
ActiveDocument.Paragraphs(1).Range.Words.Count - 1
```

You subtract 1 from the Count property to get the number of words in the paragraph because paragraph mark is counted as one word in a Words collection object.

You use the following expression to refer to the intIndexth (intIndex = 1, 2, ...) word in the 1st paragraph.

```
ActiveDocument.Paragraphs(1).Range.Words(intIndex)
```

That is, the following expressions refer to the 1st word, 2nd word, ... of the 1st paragraph.

```
ActiveDocument.Paragraphs(1).Range.Words(1)
```

```
ActiveDocument.Paragraphs(1).Range.Words(2)
```

.....

You use the 1st of the following two expressions as they are equivalent:

```
ActiveDocument.Paragraphs(1).Range.Words(intIndex)
```

```
ActiveDocument.Paragraphs.Item(1).Range.Words.Item(intIndex).Text
```


The expression

```
ActiveDocument.Paragraphs(1).Range.Words(intIndex)
```

represents the $\text{intIndex}^{\text{th}}$ ($\text{intIndex} = 1, 2, \dots$) word of the 1st paragraph. Such a word is either a string of alphabetized or numeric characters with no blank in between, or delimiters such as “;”, “:”, “&”, “(“), “)”, “+”, and so on. For example the paragraph “He runs.” has 4 words, He, runs, “.” and the paragraph mark. The paragraph (3+4 gives 7.) has 7 words, 3, +, 4, gives, 7, “.” and the paragraph mark.

Note

Each item or member in the Words collection object is a Range object that represents one word. Each word (item) can be a string of alphabetized or numeric characters with no space within the string, or punctuations or other delimiters such as “&”, “+”, Paragraph mark is considered a word (member) of the Words collection object. The item (word) in the Words collection object includes both the word and the space or spaces after the word.

As you loop through each and every word of the paragraph you use the following expression to test each word of the first paragraph to see if it is not “;” and not “.”. You use the Visual Basic function Trim to trim off any leading blank or trailing blank of the word you use for the test because each item (word) in the Words collection object includes both the word and the space or spaces after the word.

```
If (Trim(ActiveDocument.Paragraphs(1).Range.Words(intIndex)) <> ";") And _
```

```
    (Trim(ActiveDocument.Paragraphs(1).Range.Words(intIndex)) <> ".") Then
```

If the word is not “;” and not “.” you put the word into the table by the following code.

```
objTable.Cell(intIndex2, 1).Range.Text = _
```

```
    ActiveDocument.Paragraphs(1).Range.Words(intIndex)
```

After this you increment the counter `intIndex2` by 1 with the following code and go on to process the next word in the paragraph until there are no more words to process in the paragraph. The counter `intIndex2` is used to keep track of the number of entries you’ve added to the table (excluding “;” and “.”).

```
intIndex2 = intIndex2 + 1
```

The following shows the completed code.

Completed Code

```
Public Sub tableExample()
    Dim objTable As Table
    Dim intIndex, intIndex2 As Integer
    ActiveDocument.Paragraphs.Add
    ActiveDocument.Paragraphs.Last.Range.Select
    Set objTable = ActiveDocument.Tables.Add _
        (Range:=Selection.Range, NumRows:=12, NumColumns:=1)
    With objTable
```

```

.Select
With Selection
'Set column width to 1 inch
.Columns.SetWidth _
    ColumnWidth:=InchesToPoints(1#), _
    RulerStyle:=wdAdjustNone
With .Borders
    .InsideLineStyle = wdLineStyleSingle
    .InsideLineWidth = wdLineWidth100pt
    .OutsideLineStyle = wdLineStyleSingle
    .OutsideLineWidth = wdLineWidth100pt
End With
End With
End With
intIndex2 = 1
For intIndex = 1 To (ActiveDocument.Paragraphs(1).Range.Words.Count - 1)
    If (Trim(ActiveDocument.Paragraphs(1).Range.Words(intIndex)) <> ",") And _
        (Trim(ActiveDocument.Paragraphs(1).Range.Words(intIndex)) <> ".") Then
        objTable.Cell(intIndex2, 1).Range.Text = _
            ActiveDocument.Paragraphs(1).Range.Words(intIndex)
        intIndex2 = intIndex2 + 1
    End If
Next
End Sub

```

To try this example follows these steps.

Steps

- 1) Start the Visual Basic Editor.
 - 2) Use the same module as the one in the previous example.
 - 3) Insert a new procedure (macro) and name it tableExample.
 - 4) Type in the above completed code (Figure 5.15).
 - 5) If the document is blank, type in some sentences as that in Figure 5.16.
 - 6) Run the macro and you will see the words of the first paragraph put into a table with one word per row, without the delimiters “,” or “.”.
-

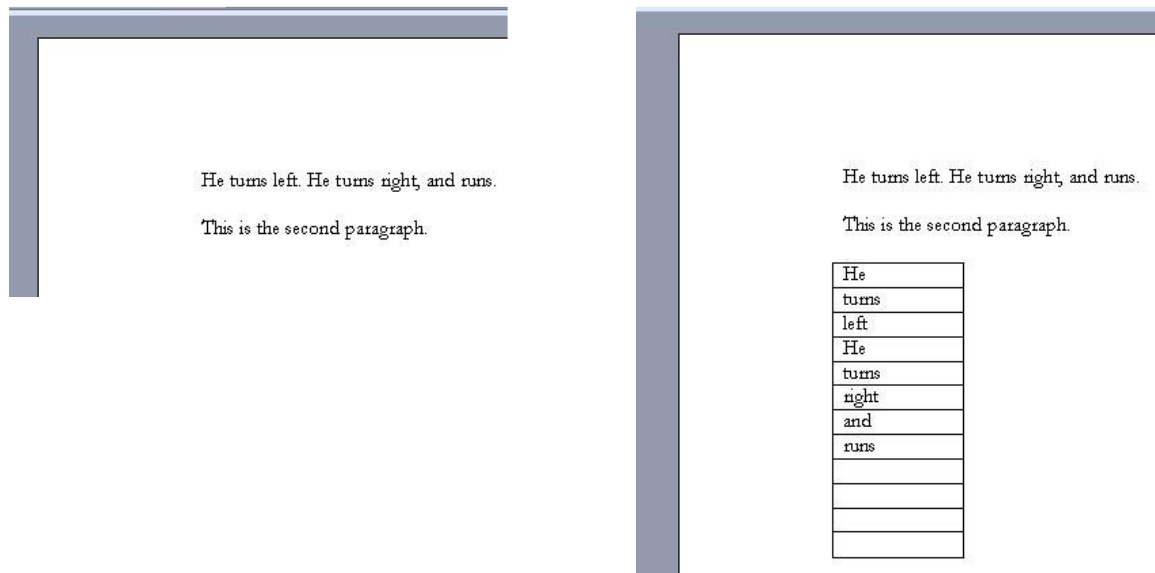
Figure 5.15 *tableExample* macro

```

Option Explicit

Public Sub tableExample()
    Dim objTable As Table
    Dim intIndex, intIndex2 As Integer
    ActiveDocument.Paragraphs.Add
    ActiveDocument.Paragraphs.Last.Range.Select
    Set objTable = ActiveDocument.Tables.Add _
        (Range:=Selection.Range, NumRows:=12, NumColumns:=1)
    With objTable
        .Select
        With Selection
            'Set column width to 1 inch
            .Columns.SetWidth _
                ColumnWidth:=InchesToPoints(1#), _
                RulerStyle:=wdAdjustNone
            With .Borders
                .InsideLineStyle = wdLineStyleSingle
                .InsideLineWidth = wdLineWidth100pt
                .OutsideLineStyle = wdLineStyleSingle
                .OutsideLineWidth = wdLineWidth100pt
            End With
        End With
    End With
    intIndex2 = 1
    For intIndex = 1 To (ActiveDocument.Paragraphs(1).Range.Words.Count - 1)
        If (Trim(ActiveDocument.Paragraphs(1).Range.Words(intIndex)) <> ",") And _
            (Trim(ActiveDocument.Paragraphs(1).Range.Words(intIndex)) <> ".") Then _
            objTable.Cell(intIndex2, 1).Range.Text = _
                ActiveDocument.Paragraphs(1).Range.Words(intIndex)
            intIndex2 = intIndex2 + 1
        End If
    Next
End Sub

```

Figure 5.16 *Result of tableExample* macro before and after

Debugging

Debugging and Error Handling

No matter how carefully you code errors can occur. The error can be in your code or the error can occur in the external environment. For example an external error occurs when your program reads a file that has been deleted by someone else. For handling external errors you write your code to anticipate them. Before you read a file your code should check to see if it exists. If the file doesn't exist your code should provide handling of such situation. Before your code performs the computation of dividing by a number you should check to make sure the number (divisor) is not zero. Suppose your code instructs the user of your program to enter a zip code, your program should check to see if the number being entered is a valid zip code. The key is prevention and anticipation of error and provision in your code of handling the errors if it occurs. It is common sense, like preventive health care and safe driving. I have an instructor of traffic school who tell his students that the key to driving safely is to assume that everyone else is not (driving safely)! You get the idea.

The process of locating and fixing errors in your own code is debugging. Visual Basic provides an abundance of debugging tools to help you do that. In fact more than you and I ever need. From the Visual Basic menu selects Debug to see the debugging tools available (Figure 5.17), or select View / Toolbars / Debug to display the Debug toolbar (Figure 5.18). You will go through several examples of using these tools in the following sections.

Figure 5.17 Visual Basic Debug menu

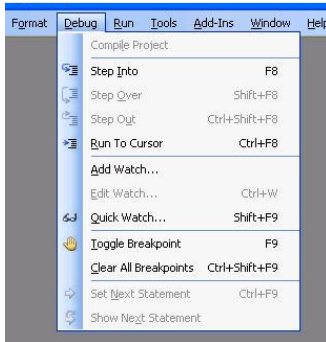


Figure 5.18 Visual Basic Debug toolbar



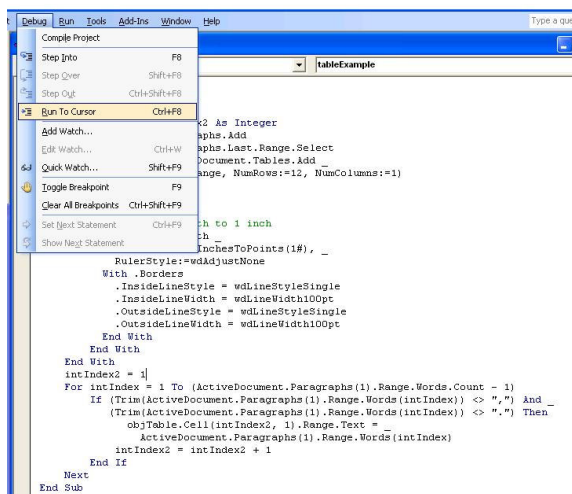
Trace Execution of Your Program

Most debugging involves tracing the execution of portion of your program to locate the error. Let's use the previous table programming example to illustrate that. Make sure that Module1 (Code) window is visible and in-focus (selected). (Module1 is the module that contains the tableExample macro). In the tableExample macro (Sub procedure) places your cursor anywhere within the following statement (Figure 5.19).

```
intIndex2 = 1
```

Figure 5.19 show the cursor displayed at the end of the statement but the cursor at any other position within the statement is fine. From the Visual Basic editor menu select Debug / Run to Cursor (Figure 5.19).

Figure 5.19 *Select Run to Cursor*

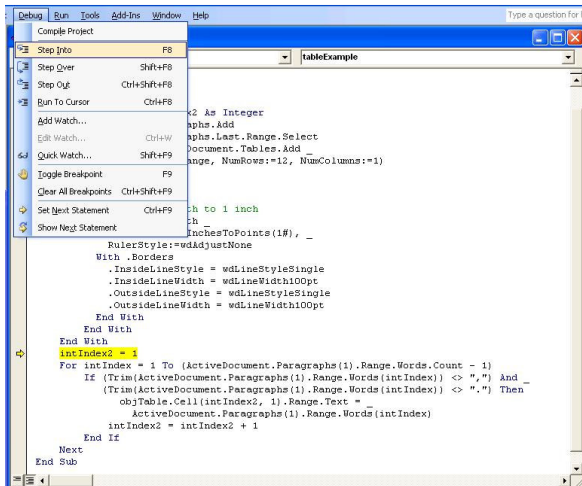


After you've selected Debug / Run to Cursor the program begins execution and then stops at the statement where the cursor was at with a yellow arrow sign and the statement highlighted in yellow (Figure 5.20). The program is suspended at this statement (before the statement is executed).

```
intIndex2 = 1
```

To execute this statement and go on to the next statement select Debug / Step Into from the Visual Basic editor menu or press F8 (Figure 5.20).

Figure 5.20 Program execution stops at cursor

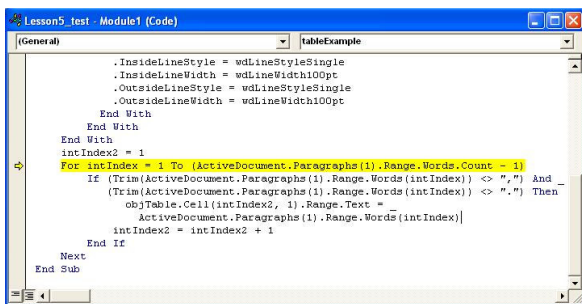


After you've selected Debug / Step Into from the Visual Basic editor menu (or press F8) the yellow arrow shifts to the next statement (Figure 5.21).

```
For intIndex = 1 To (ActiveDocument.Paragraphs(1).Range.Words.Count - 1)
```

Now the program is suspended at this statement (before the statement is executed).

Figure 5.21 Step into

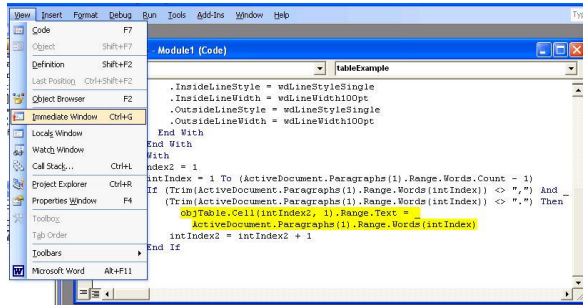


Next press F8 two more times to have the program stop at the following statement (Figure 5.22).

```
objTable.Cell(intIndex2, 1).Range.Text = _
```

```
    ActiveDocument.Paragraphs(1).Range.Words(intIndex)
```

Next select View / Immediate Window from the Visual Basic editor menu (Figure 5.22).

Figure 5.22 *Select Immediate window*

The Immediate window appears (Figure 5.23). In the Immediate window type the following expression (preceded by ?) and press enter (Figure 5.23).

```
?ActiveDocument.Paragraphs(1).Range.Words(intIndex)
```

The Immediate window displays He (Figure 5.23). This tells you that the content of the above expression is He. Put your cursor over `intIndex` in the following statement (the statement before the statement being stopped at) and you will see the box `intIndex = 1` display (Figure 5.23).

```
(Trim(ActiveDocument.Paragraphs(1).Range.Words(intIndex)) <> ".") Then
```

This confirms that the content as represented by

```
?ActiveDocument.Paragraphs(1).Range.Words(intIndex)
```

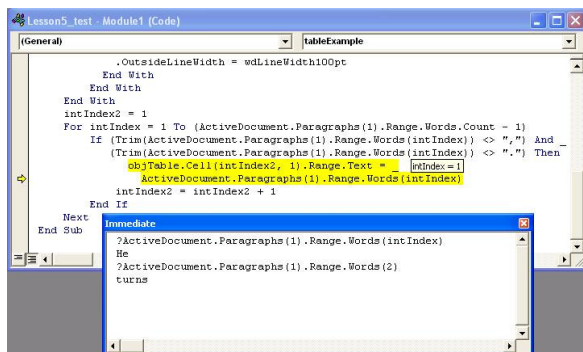
is

```
?ActiveDocument.Paragraphs(1).Range.Words(1)
```

In the Immediate window type the following expression (preceded by ?) and press enter (Figure 5.23).

```
?ActiveDocument.Paragraphs(1).Range.Words(2)
```

The Immediate window displays turns (Figure 5.23). This tells you that the content of the above expression is the text turns.

Figure 5.23 *tableExample macro*

To stop debugging and resume normal execution of your program selects Run / Run Sub/UserForm from the Visual Basic editor menu (or press F5).

Step Into, Step Over and Step Out

Visual Basic debugging provides three types of stepping through your code, Step Into, Step Over and Step Out. Step Into lets you step through your code one statement at a time. You've seen example of using Step Into. Step Over lets you step over a statement which is calling another procedure such as a Sub procedure or a function procedure. For example suppose your program is suspended at the following statement.

```
intNumber = solve_equation (sngArray)
```

In the above statement you are calling solve_equation, a function procedure. If you use Step Into you will step into solve_equation and stops at the first executable statement of solve_equation. If you don't want to step into solve_equation you use Step Over which will execute the above statement and stop at the next statement following the above statement, without stepping into solve_equation. If there are no function or Sub procedure being called in the statement being stopped, using Step Into or Step Over gives the same result.

Step Out lets you step out of the current procedure where the statement being stopped at is located. For example, suppose your macro (Sub procedure) uses a function and the program execution is being stopped at a statement in the function. Using Step Out will resume and finish execution of the function and suspend the program at the statement after the statement which calls (uses) the function. For another example suppose your macro (Sub procedure) does not call (use) any other procedure (Sub or function) and you're being stopped at a statement in the macro. In this situation using the Step Out will resume and finish execution of the macro until it ends.

This concludes Chapter 5.

Chapter 6 Database and Mail Merge Programming

Database programming is a vast topic. It is not possible to give a comprehensive treatment of the topic in this chapter, but you will learn some of the principle and technique of database programming by going through an example. You will also go through an example of using mail merge programming.

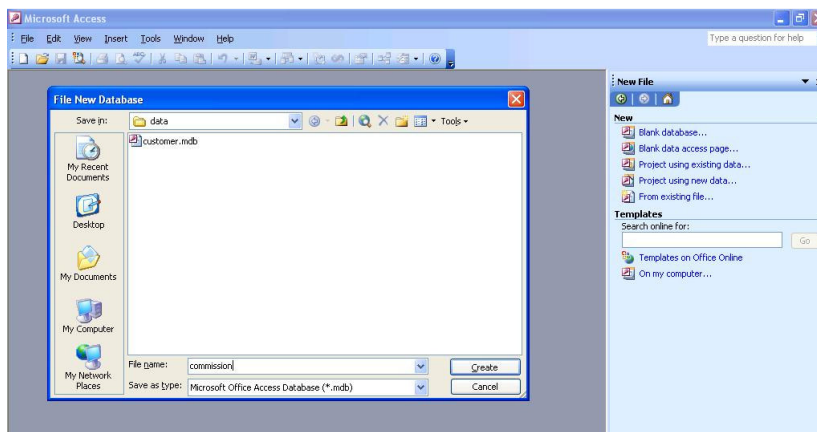
Create Database

You are going to create an Access database for the database programming example in this chapter. To create an Access database follow these steps.

Steps

- 1) Start Access.
- 2) From the Access menu select File / New.
- 3) Select Blank database... from the New File window on the right.
- 4) When the File New Database window appears, name the database as commission and save it in the c:\data folder (Figure 6.1). If you don't have the c:\data folder you have to create the folder first before saving the database file.

Figure 6.1 Create Access database



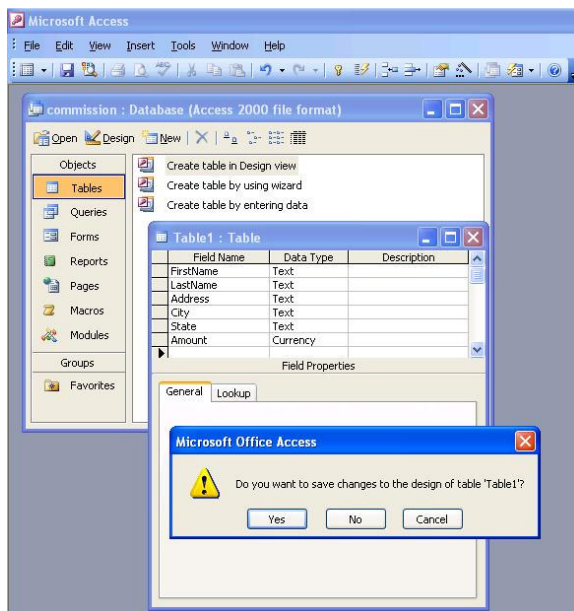
Create Table

You'll next create a table and define some data fields for the table.

Steps

- 1) From the commission: Database window select /Create table in Design View (Figure 6.2).
- 2) The Table1: Table window appears. Type in the 6 Field Names of FirstName, LastName, Address, City, State and Amount. Use the default Data Type of Text for the first 5 data fields. For the Amount data field use the Currency data type (Figure 6.2).
- 3) Close the Table1: Table window and the Microsoft Office Access window appears asking Do you want to save changes to the design of table 'Table1'? Select the Yes button (Figure 6.2).

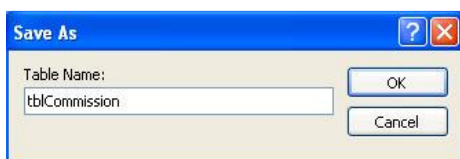
Figure 6.2 Create table



Steps

- 1) The Save As window appears (Figure 6.3).
- 2) Type in tblCustomer as the name for the table and Click the OK button.

Figure 6.3 Table name



Steps

- 1) The Microsoft Office Access window appears asking Do you want to create a primary key now? (Figure 6.4).
- 2) Select the No button.

Figure 6.4 *No primary key*

Note

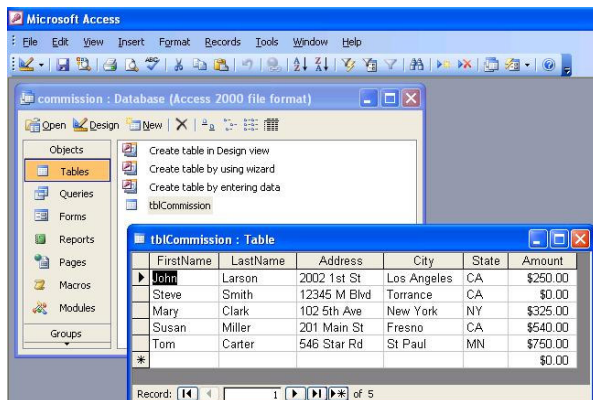
You cannot enter duplicate or null (empty) values in the data field that has been designated as a primary key field in an Access table.

Put Data in Your Table

You'll next put data into the table you've created.

Steps

- 1) From the commission: Database window, select tblCommission to display the tblCommission: Table window (Figure 6.5).
- 2) Enter data for at least 5 employees (Figure 6.5).
- 3) Close the tblCommission: Table window.
- 4) Exit Access.
- 5) Your Access database and the table are now ready for use.

Figure 6.5 *Put data in table*

Database Programming

DAO Object Library

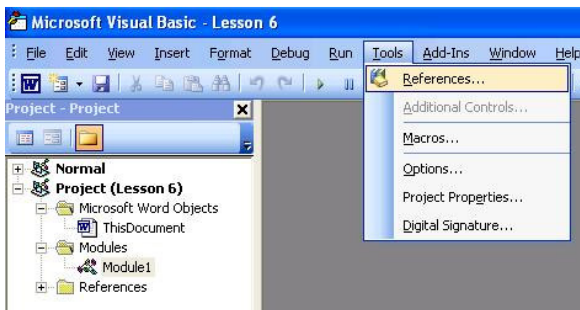
In the following example you'll use DAO objects to read the Access database you've created and display the data in a table format in your Word document. DAO stands for Data Access Objects. You use DAO objects to read, update, create and manipulate databases. Before you can use DAO objects you must establish a reference to the DAO object library.

Follow these steps to establish a reference to DAO object library.

Steps

- 1) Start Visual Basic Editor.
 - 2) On the Tools menu, select References (Figure 6.6).
-

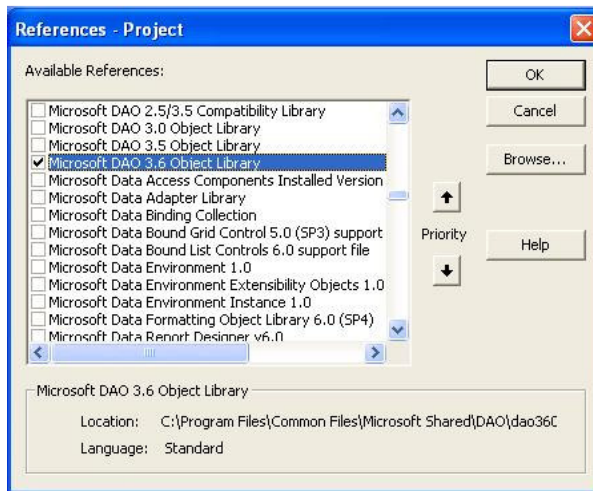
Figure 6.6 Referencing DAO object library



Steps

- 1) The Reference – Project window appears (Figure 6.7).
 - 2) In the Available References box, scroll down to Microsoft DAO 3.6 Object Library and select it (Figure 6.7).
 - 3) Close the References – Project window.
 - 4) You've established a reference to DAO object library.
-

Figure 6.7 DAO 3.6 object library



Help and Documentation on DAO Objects

After you establish a reference to DAO object library you can navigate DAO objects in the Object Browser the same way you navigate other Visual Basic and Words objects. More help and extensive documentation on DAO objects is available when you use the Object Browser of the Visual Basic Editor in an opened Access database. Additionally Access Help provides extensive help and documentation on DAO objects.

Database Programming Example

This example reads the commission table of the Access database you've created and display the data in a table in your Word document. The following discussion walks you through the code with explanation of the objects and logic involved.

Database Object

The Database object is a member class of DAO object library. A Database object represents an open database. You use Database object and its methods and properties to read, update and manipulate an open database. The following code declares dbs as an object variable of the Database class.

```
Dim dbs As Database
```

OpenDatabase Method

The OpenDatabase method is a function method of the Workspace class. As a function method it returns a reference to an object of the Database class. The Workspace class is a member class of DAO object library. When you use the OpenDatabase method you may drop the Workspace term, so the following 2 expressions are equivalent.

```
Workspace.OpenDatabase
```

```
OpenDatabase
```

The following code uses the OpenDatabase method to open the Access database, c:\data\commission.mdb and assign to dbs the reference to the database. That is, the OpenDatabase method returns to dbs (an object variable) an object that represents the Access database, c:\data\commission.mdb.

```
Set dbs = OpenDatabase("c:\data\commission.mdb")
```

From now on you can use the methods and properties of the Database object to manipulate the Access database, c:\data\commission.mdb that is represented by dbs (a Database object). You will next use the OpenRecordset method of the Database object to do that.

Workspace Object

The Workspace class has methods and properties to manage the session when your code interacts with a database. You use the methods and properties of the Workspace class to improve performance, control security and manage transactions of the database.

The following methods (they are all Sub procedure methods) of the Workspace class are always used together to ensure the integrity of the transaction when your code is adding, deleting or updating records to a database that is being used by more than one user.

- 1) BeignTrans
- 2) CommitTrans
- 3) Rollback

Recordset Object

The Recordset object is a member class of DAO object library. You use the methods and properties of the Recordset object to manipulate your database at the records level and for each record at the data field's level. Think of each record of a database as a row of data and for each row of data you have the columns representing the data fields. There are 5 types of Recordset object and you will use the dynaset-type in the following example. A dynaset-type Recordset object is a dynamic set of records that you can use to read, add, change, or delete records from an underlying database table or tables. A dynaset-type Recordset object can contain fields from one or more tables in a database.

The following code declares rst as an object variable of the Recordset class.

Dim rst As Recordset

OpenRecordset Method

The OpenRecordset method is a function method of the Database object. As a function method it returns a reference to an object (representing records of the database) of the Recordset class. The following code uses the OpenRecordset method to assign to rst the reference to the object representing the records selected according to the criteria specified in the SQL statement contained in strSQL (a string variable). The records selected are those records of employees with amount > 0 (commission greater than 0) and the records selected are sorted by the last name.

```
strSQL = "SELECT * FROM tblCommission " _
        & "WHERE Amount > 0 ORDER BY LastName"
```

```
Set rst = dbs.OpenRecordset(strSQL, dbOpenDynaset, dbReadOnly)
```

In this example the OpenRecordset method takes on 3 parameters, strSQL, dbOpenDynaset and dbReadOnly. The strSQL parameter specifies the selection criteria of the records, the dbOpenDynaset parameter specifies the Recordset object be the dynaset-type and the dbReadOnly specifies this is a read-only operations of the records.

From now on rst is a Recordset object representing the records of the employee with amount > 0. You can use the methods and properties of rst (a Recordset object) to manipulate the records (rows) and columns (data fields) of the selected records.

What is SQL?

SQL stands for Structured Query Language. It was originally developed by IBM in the 1970s and has since evolved into a platform independent query language to update and query information in databases. You use SQL statements to update and query databases from Access, SQL Server, Oracle and others. SQL has rich syntax and can perform any operation imaginable on databases. A complete description of the SQL language will fill a complete book of considerable size. You can find help and documentation in Microsoft Jet SQL Reference in Access Help.

In this example you use the SQL Select statement to select the records of the employees whose Amount are > 0. By convention you capitalize SQL keyword such as Select, although you don't have to. You use the SQL clause "Select * From tblCommission" to select all the records from the tblCommission: table. The SQL clause "Where Amount > 0" specifies that you only select the employees whose Amount are > 0. You use the SQL clause "Order by LastName" to sort the selected records in ascending (default) order of the last name of the employee.

```
SELECT * FROM tblCommission WHERE Amount > 0 ORDER BY LastName
```

Get Number of Records

The following code uses the MoveLast method (a Sub procedure method) of the Recordset object to move to the last record of the selected records. The statement (intCount = rst.RecordCount) gets the number of employee records being selected using the RecordCount property (a long integer data type) of the Recordset class. You are going to use this information to add a table with the same number of rows equal to the number of employees selected. The table displays the information of each employee in each row. You use the MoveFirst

method (a Sub procedure method) of the Recordset object to move to the first record of the selected records to get ready for processing of the records later in the program.

```
rst.MoveLast
```

```
intCount = rst.RecordCount
```

```
rst.MoveFirst
```

Selection Object

The following code clears the content of the document. You use the WholeStory method (a Sub procedure method) of the Selection object to select the whole content of the document. You then use the Delete method (a function method) to delete what you've just selected.

```
' Clear the content of the document
```

```
Selection.WholeStory
```

```
Selection.Delete
```

The above code needs further clarification. The Selection object you use in this example is a member class of Word object library. A Selection object represents a selected area in the document, or the insertion point if nothing in the document is selected. Only one Selection object can be active at any time. After the statement (Selection.WholeStory) is executed the Selection object represents the whole content of the active document. That is, the WholeStory method assigns to the Selection object a reference to the content of the document. From now on when you use the Selection object you are referring to the whole content of the document. If the document is empty the Selection object represents the insertion point at the beginning of the document. The (Selection.Delete) statement deletes the object represented by the Selection object, which is the whole content of the document.

The Delete method is a function method that returns an integer value, the number of units being deleted. You can specify the number of units as the number of characters deleted or the number of words deleted. As you are not specifying the number of units being deleted, the returned value will be 1 if there is content in the document or 0 if there is no content in the document (empty document).

The following code is another way of doing the same thing.

```
ActiveDocument.Content.Select
```

```
Selection.Delete
```

Note

A Selection object represents a selected area in the document, or the insertion point if nothing in the document is selected. Only one Selection object can be active at any given time.

Setup Table

The following code creates a new table at the beginning of the document. You create the table by using the Add method (a function method) of the Tables class. The Add method is a function method that takes on some required parameters and some optional parameters. The Range parameter (Range:=Selection.Range) is required and must be a reference to an object of the Range class. You use the Range parameter to tell the Add method where to put the new table. Recall you've deleted the content of the document and the Selection object refers to the position at the beginning of the document, so the new table will be created there.

The NumRows and NumColumns parameters tell the Add method the numbers of row and column of the table to be created. In this case you are creating a table of (intCount + 2) rows and 3 column. The Range, NumRows and NumColumns parameters are required parameters. That means you must supply them when you use the Add method of the Tables class. You're creating the number of rows of the new table equal to the number of records selected (intCount) plus 2, so you can display each employee data per row, plus another 2 rows for the heading and the total amount.

```
' Add new table
```

```
Set tblTable = ActiveDocument.Tables.Add _
```

```
    (Range:=Selection.Range, NumRows:=intCount + 2, NumColumns:=3)
```

```
With tblTable
```

```
    .Select
```

```
    With Selection
```

```
        'Set column width to 1.5 inch
```

```
        .Columns.SetWidth _
```

```
            ColumnWidth:=InchesToPoints(1.5), _
```

```
            RulerStyle:=wdAdjustNone
```

```
        With .Borders
```

```
            .InsideLineStyle = wdLineStyleSingle
```

```
            .InsideLineWidth = wdLineWidth100pt
```

```
            .OutsideLineStyle = wdLineStyleSingle
```

```
            .OutsideLineWidth = wdLineWidth100pt
```

```
        End With
```

```
    End With
```

```
End With
```

The following code writes heading to the first row of the table.

```
' Put in heading for the table

tblTable.Cell(1, 1).Range.Text = "Name"

tblTable.Cell(1, 2).Range.Text = "Address"

tblTable.Cell(1, 3).Range.Text = "Amount"

tblTable.Rows(1).Range.Bold = True
```

Display Data from Database

The following code uses the Do...Loop to loop through each record of the employee to read the data and write them into the table (1 record per row). Recall that you've used MoveFirst method of the Recordset object to move to the first record, and as a result of that rst (object variable of the Recordset class) now refer to the first row (record) of the selected records. As you loop through each record you use the MoveNext method (a Sub procedure method) of the Recordset object to move to the next record. At the beginning of the loop you use the EOF property of the Recordset object to test whether the current record position is after the last record, and if this is so (that means you've read every record) the Do...Loop exits.

You use the expression rst.Fields("LastName").Value to read the last name contained in the LastName data field in the current record. The Fields property of the Recordset object is a collection object but the Fields("LastName") is an object of the Field class. You use the Value property of the Field class to refer to the data (last name of the employee) of the current record.

```
' Loop through the records to pick up the data

' and put it in the table.

sngSum = 0

intIndex = 2

Do While Not rst.EOF

tblTable.Cell(intIndex, 1).Range.Text = _

    rst.Fields("LastName").Value & ", " & _

    rst.Fields("FirstName").Value

tblTable.Cell(intIndex, 2).Range.Text = _

    rst.Fields("Address").Value & ", " & _

    rst.Fields("City").Value & ", " & _

    rst.Fields("State").Value

tblTable.Cell(intIndex, 3).Range.Text = _

    Format(rst.Fields("Amount").Value, "$###,###0.00")
```

```

sngSum = sngSum + rst.Fields("Amount").Value

rst.MoveNext

intIndex = intIndex + 1

```

Loop

You use the expression `tblTable.Cell(intIndex, 1).Range.Text` to refer to the text content of the specified cell of the table. The `Cell` method (a function method) of the `Table` object returns an object of the `Cell` class. The `Range` property of the `Cell` class is an object of the `Range` class. You then use the `Text` property of the `Range` object to refer to the text content of the cell.

Before the `Do...Loop` the integer variable, `intIndex` is set = 1 and the single precision number variable, `sngSum` is set = 0. The variable `intIndex` keeps track of the number of records read and the corresponding row of the table to which the data are written. The variable `sngSum` accumulate the amount of commission amount of the employees and reports the total amount in the last row of the table.

Note

The Fields collection object of a `Recordset` object represents the `Field` objects in a row of data, or in a record. To refer to a `Field` object in a collection object, use `Fields("name")`, which is an object of the `Field` class, where `name` is a valid data field name.

The following code write the total amount of commission amount to the last row of the table and close the object variables, `rst` and `db`.

```

tblTable.Cell(intIndex, 2).Range.Text = "Total = "

tblTable.Cell(intIndex, 3).Range.Text = Format(intSum, "$##,##0.00")

tblTable.Rows(intIndex).Range.Bold = True

rst.Close

db.Close

```

The following is the completed code.

Completed code

```

Public Sub dbExample()
    ' This example read from an Access database and
    ' write the records to a table in the Word
    ' document.

    Dim tblTable As Table, strSQL As String
    Dim intIndex, intCount As Integer

```

```
Dim sngSum As Single
Dim dbs As Database, rst As Recordset

' Open database.
Set dbs = OpenDatabase("c:\data\commission.mdb")

' Sort by last name.
strSQL = "SELECT * FROM tblCommission " _
    & "WHERE Amount > 0 ORDER BY LastName"
Set rst = dbs.OpenRecordset(strSQL, dbOpenDynaset, dbReadOnly)

' Get the number of records so that table can be set up
' with the same number of rows.
rst.MoveLast
intCount = rst.RecordCount
rst.MoveFirst

' Clear the content of the document
Selection.WholeStory
Selection.Delete

' Add new table
Set tblTable = ActiveDocument.Tables.Add _
    (Range:=Selection.Range, NumRows:=intCount + 2, NumColumns:=3)
With tblTable
    .Select
    With Selection
        'Set column width to 1.5 inch
        .Columns.SetWidth _
            ColumnWidth:=InchesToPoints(1.5), _
            RulerStyle:=wdAdjustNone
    With .Borders
        .InsideLineStyle = wdLineStyleSingle
        .InsideLineWidth = wdLineWidth100pt
        .OutsideLineStyle = wdLineStyleSingle
        .OutsideLineWidth = wdLineWidth100pt
    End With
    End With
End With
```

End With

' Put in heading for the table

tblTable.Cell(1, 1).Range.Text = "Name"

tblTable.Cell(1, 2).Range.Text = "Address"

tblTable.Cell(1, 3).Range.Text = "Amount"

tblTable.Rows(1).Range.Bold = True

' Loop through the records to pick up the data

' and put it in the table.

sngSum = 0

intIndex = 2

Do While Not rst.EOF

tblTable.Cell(intIndex, 1).Range.Text = _
rst.Fields("LastName").Value & ", " & _
rst.Fields("FirstName").Value

tblTable.Cell(intIndex, 2).Range.Text = _
rst.Fields("Address").Value & ", " & _
rst.Fields("City").Value & ", " & _
rst.Fields("State").Value

tblTable.Cell(intIndex, 3).Range.Text = _
Format(rst.Fields("Amount").Value, "\$##,##0.00")

sngSum = sngSum + rst.Fields("Amount").Value

rst.MoveNext

intIndex = intIndex + 1

Loop

tblTable.Cell(intIndex, 2).Range.Text = "Total = "

tblTable.Cell(intIndex, 3).Range.Text = Format(sngSum, "\$##,##0.00")

tblTable.Rows(intIndex).Range.Bold = True

rst.Close

dbs.Close

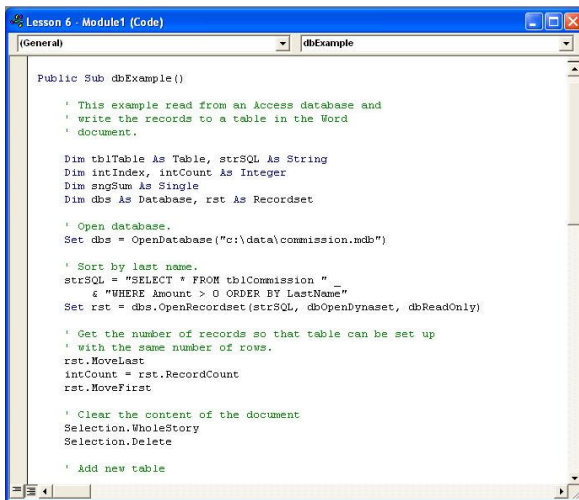
End Sub

To try this example follows these steps.

Steps

- 1) Start a blank document and start Visual Basic Editor.
- 2) Insert a new module and keep the default name as Module1.
- 3) Insert a new procedure (macro) and name it dbExample.
- 4) Type in the above completed code (Figure 6.8).
- 5) Run the macro and see the table displaying the employee information in your document (Figure 6.9).

Figure 6.8 *dbExample* macro



```
Lesson 6 - Module1 [Code]
(General) dbExample

Public Sub dbExample()
    ' This example read from an Access database and
    ' write the records to a table in the Word
    ' document.

    Dim tblTable As Table, strSQL As String
    Dim intIndex, intCount As Integer
    Dim sngSum As Single
    Dim dbs As Database, rst As Recordset

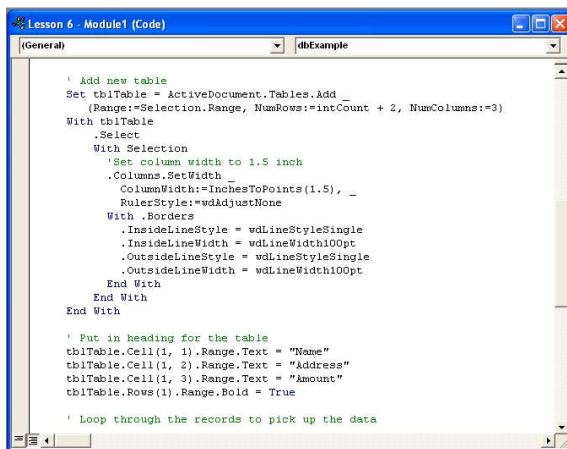
    ' Open database.
    Set dbs = OpenDatabase("c:\data\commission.mdb")

    ' Sort by last name.
    strSQL = "SELECT * FROM tblCommission " _
    & "WHERE Amount > 0 ORDER BY LastName"
    Set rst = dbs.OpenRecordset(strSQL, dbOpenDynaset, dbReadOnly)

    ' Get the number of records so that table can be set up
    ' with the same number of rows.
    rst.MoveLast
    intCount = rst.RecordCount
    rst.MoveFirst

    ' Clear the content of the document
    Selection.WholeStory
    Selection.Delete

    ' Add new table
```



```
Lesson 6 - Module1 [Code]
(General) dbExample

    ' Add new table
    Set tblTable = ActiveDocument.Tables.Add _
    (Range:=Selection.Range, NumRows:=intCount + 2, NumColumns:=3)
    With tblTable
        .Select
        With Selection
            'Set column width to 1.5 inch
            .Columns.SetWidth _
            ColumnWidth:=InchesToPoints(1.5), _
            RulerStyle:=wdAdjustNone
            With .Borders
                .InsideLineStyle = wdLineStyleSingle
                .InsideLineWidth = wdLineWidth100pt
                .OutsideLineStyle = wdLineStyleSingle
                .OutsideLineWidth = wdLineWidth100pt
            End With
        End With
    End With

    ' Put in heading for the table
    tblTable.Cell(1, 1).Range.Text = "Name"
    tblTable.Cell(1, 2).Range.Text = "Address"
    tblTable.Cell(1, 3).Range.Text = "Amount"
    tblTable.Rows(1).Range.Bold = True

    ' Loop through the records to pick up the data
```

```

Lesson 6 - Module1 (Code)
dbExample

' Loop through the records to pick up the data
' and put it in the table.
sngSum = 0
intIndex = 2
Do While Not rst.EOF
    tblTable.Cell(intIndex, 1).Range.Text = _
        rst.Fields("LastName").Value & ", " & _
        rst.Fields("FirstName").Value
    tblTable.Cell(intIndex, 2).Range.Text = _
        rst.Fields("Address").Value & ", " & _
        rst.Fields("City").Value & ", " & _
        rst.Fields("State").Value
    tblTable.Cell(intIndex, 3).Range.Text = _
        Format(rst.Fields("Amount").Value, "$##,##0.00")
    sngSum = sngSum + rst.Fields("Amount").Value
    rst.MoveNext
    intIndex = intIndex + 1
Loop

tblTable.Cell(intIndex, 2).Range.Text = "Total = "
tblTable.Cell(intIndex, 3).Range.Text = Format(sngSum, "$##,##0.00")
tblTable.Rows(intIndex).Range.Bold = True

rst.Close
dbs.Close
End Sub

```

Figure 6.9 Result of dbExample macro

Name	Address	Amount
Carter, Tom	546 Star Rd, St Paul, MN	\$750.00
Clark, Mary	102 5th Ave, New York, NY	\$325.00
Larson, John	2002 1st St, Los Angeles, CA	\$250.00
Miller, Susan	201 Main St, Fresno, CA	\$540.00
	Total =	\$1,865.00

Mail Merge Programming

You use the Mail Merge feature of Word to perform a mail merge operation, or you can write a macro using the MailMerge object to perform the same operation. The following example shows you how to code a simple mail merge macro. One advantage of coding your own mail merge macro is that you have complete programming control of the process. In this example you extract the same information as in the last example from the same Access database you've set up earlier. Unlike the last example where the data are displayed in a table in the document, this example generates a monthly commission check letter for each of the 4 employees who earn a commission last month.

Basically your code builds text and merge fields (data fields) in a letter format to produce a master document. Then you use the MailMerge object to perform the mail merge operation to generate the monthly commission letters for the employees selected.

MailMerge Object

The following code uses the MailMerge property of the Document class (ActiveDocument is an object of the Document class) to return an object of the MailMerge class. You then use the OpenDatabase method (a Sub procedure method) of the MailMerge object to attach a data source to the document. The OpenDatabase method takes on 2 parameters in this case. The parameter Name:= strPath specifies the data source as the Access database, c:\data\commission.mdb you've created earlier and used in the previous example. The parameter SQLstatement:=strSQL passes a SQL statement which specifies that only employees whose commission amount > 0 are to be selected, and the records selected are to be sorted in alphabetical order of the last name of the employee.

```
strPath = "c:\data\commission.mdb"

strSQL = "SELECT * FROM [tblCommission] " _
    & "WHERE Amount > 0 ORDER BY LastName"

With ActiveDocument.mailMerge
    .OpenDataSource Name:= strPath, SQLstatement:=strSQL
```

Selection Object

The following code clears the content of the document and makes the Selection object represent the insertion point at the beginning of the document. See the previous example on a discussion of the technique involved.

```
' Clear content of document

Selection.WholeStory

Selection.Delete
```

The following code uses methods and properties of the Selection object to build texts and add merge fields (data fields) to the main document. You've already cleared the content of the document and the Selection object refers to the beginning of the document. You build and format the text starting from a blank document, which is the main document that remains the same in each letter. The merge fields which are name, address and commission amount of each employee changes in each letter.

You use the code (Selection.Font.Bold = True) to turn the Bold property on and the code (Selection.Font.Bold = False) to turn it off. In this case Bold is a property of the Font object, and Font is a property of the Selection object.

You use the TypeText method (a sub procedure method) of the Selection object to add a line of text.

The macro calls the Sub procedure, add_Para to add 2 new blank paragraphs (2 new paragraph marks). The Sub procedure, add_Para uses the TypeParagraph method (a Sub procedure method) of the Selection object in a For...Next loop to do that.

The macro calls the function procedure, insert_Field to insert a data field (merge field) called FirstName to the main document. After this code is executed the main document will show the data field as <<FirstName>>. If you select it and right click to select the Toggle Field Codes option it will be shown as { MERGEFIELD FirstName}. The insert_Field function uses the Selection.Fields.Add method to do this. Add is a method (a function method) of the Fields object which is a property of the Selection object. The Add method (a function

method) returns an object of the Field class. The Add method takes on 3 parameters. The Range:=Selection.Range parameter tells the Add method to insert the data field to the position the Selection object is referring to. The Type:=wdFieldMergeField parameter specifies that this is a MergeField field code. The Text:=strName parameter gives the data field name. Similarly you insert the data fields for the last name, address, city, state and the amount data fields.

With Selection

```

        .Font.Bold = True

        .TypeText "Monthly Commission Check"

        add_Para 2

        .Font.Bold = False

        insert_Field "FirstName"

        .TypeText " "

        insert_Field "LastName"

        .TypeParagraph

        insert_Field "Address"

        .TypeParagraph

        insert_Field "City"

        .TypeText ", "

        insert_Field "State"

        add_Para 2

        .TypeText "Your monthly commission check amount is "

        .Font.Bold = True

        insert_Field "Amount \# $$$,###.00"

    End With

```

```

Public Sub insert_Field(strName As String)

    Selection.Fields.Add Range:=Selection.Range, _

        Type:=wdFieldMergeField, Text:=strName

End Sub

```

```

Public Sub add_Para(intIndex As Integer)

    Dim int2 As Integer

```

```

For int2 = 1 To intIndex
    Selection.TypeParagraph
Next
End Sub

```

Execute the Mail Merge

You've added text, merge fields (data fields) to setup the main document. Next you are ready to merge the data from the Access database to the main document to create the monthly letters to the employee. The following code uses the Destination property of the MailMerge object to specify that the result of the mail merge is to be sent to a new document. The Execute method (a Sub procedure method) of the MailMerge object performs the specified mail merge operation.

```

.Destination = wdSendToNewDocument
.Execute

```

After the above code is executed a new document will be created. The new document contains the 4 monthly commission letters for the 4 employees.

The following is the completed code.

Completed code

```

Public Sub mailMerge()
    Dim strPath, strSQL As String
    ' Clear content of document
    Selection.WholeStory
    Selection.Delete

    strPath = "c:\data\commission.mdb"
    strSQL = "SELECT * FROM [tblCommission] " _
        & "WHERE Amount > 0 ORDER BY LastName"
    With ActiveDocument.mailMerge
        .OpenDataSource Name:=strPath, SQLstatement:=strSQL
    With Selection
        .Font.Bold = True
        .TypeText "Monthly Commission Check"
        add_Para 2
        .Font.Bold = False
        insert_Field "FirstName"
        .TypeText " "

```

```

insert_Field "LastName"
.TypeParagraph
insert_Field "Address"
.TypeParagraph
insert_Field "City"
.TypeText ", "
insert_Field "State"
add_Para 2
.TypeText "Your monthly commission check amount is "
.Font.Bold = True
insert_Field "Amount \# $$$,###.00"
End With
..Destination = wdSendToNewDocument
.Execute
End With
End Sub

```

```

Public Sub insert_Field(strName As String)
    Selection.Fields.Add Range:=Selection.Range, _
        Type:=wdFieldMergeField, Text:=strName
End Sub

```

```

Public Sub add_Para(intIndex As Integer)
    Dim int2 As Integer
    For int2 = 1 To intIndex
        Selection.TypeParagraph
    Next
End Sub

```

To try this example follows these steps.

Steps

- 1) Start a blank document and start Visual Basic Editor.
- 2) Insert a new module and keep the default name as Module1.
- 3) Insert a new procedure (macro) and name it mailExample.
- 4) Type in the above completed code (Figure 6.10).
- 5) Run the macro and see the generated master document (Figure 6.11).
- 6) See the 4 monthly commission letters generated (Figure 6.12).

Figure 6.10 *mailExample* macro

```
Lesson 6 - Module1 (Code)
mailExample

Public Sub mailExample()
    Dim strPath, strSQL As String
    ' Clear content of document
    Selection.WholeStory
    Selection.Delete

    strPath = "c:\data\commission.mdb"
    strSQL = "SELECT * FROM [tblCommission] " _
    & "WHERE Amount > 0 ORDER BY LastName"

    With ActiveDocument.MailMerge
        .OpenDataSource Name:=strPath, SQLStatement:=strSQL
        With Selection
            .Font.Bold = True
            .TypeText "Monthly Commission Check"
            add Para 2
            .Font.Bold = False
            insert_Field "FirstName"
            .TypeText " "
            insert_Field "LastName"
            .TypeParagraph
            insert_Field "Address"
            .TypeParagraph
            insert_Field "City"
            .TypeText ", "
            insert_Field "State"
            add Para 2
            .TypeText "Your monthly commission check amount is "
            .Font.Bold = True
            insert_Field "Amount \# ###,###.00"
        End With
        .Destination = wdSendToNewDocument
        .Execute
    End With
End Sub
```

```
Lesson 6 - Module1 (Code)
insert_Field

.Execute
End With
End Sub

Public Sub insert_Field(strName As String)
    Selection.Fields.Add Range:=Selection.Range, _
    Type:=wdFieldMergeField, Text:=strName
End Sub

Public Sub add_Para(intIndex As Integer)
    Dim int2 As Integer
    For int2 = 1 To intIndex
        Selection.TypeParagraph
    Next
End Sub
```

Figure 6.11 *Master document after mail merge*

Monthly Commission Check

«FirstName» «LastName»

«Address»

«City», «State»

Your monthly commission check amount is «Amount»

Figure 6.12 Result of mailExample macro



This Concludes Chapter 6.

Chapter 7 File Programming

FileSystemObject Object

In this chapter you will go through several examples of using the FileSystemObject object to learn some of the technique and principles in working with folders and files, including Word document files and non-Word files.

Create New Text File

This example creates a new text file, write 2 lines to it, and close the file.

FileSystemObject Object

The following code declares fs as an object variable and uses the CreateObject method (a function method) of the Interaction class to assign to fs a reference to the Scripting.FileSystemObject object. The assignment makes fs an object of the FileSystemObject class. From now on fs can use the methods and properties of the FileSystemObject object to manipulate the files system of the Windows operating system.

```
Dim fs As Object
```

```
Set fs = CreateObject("Scripting.FileSystemObject")
```

The Interaction class is a member module of VBA Library. When using the CreateObject method you may drop the reference to the Interaction class and simply use CreateObject instead of Interaction.CreateObject. That is, the following two expressions are equivalent:

```
Set fs = Interaction.CreateObject("Scripting.FileSystemObject")
```

```
Set fs = CreateObject("Scripting.FileSystemObject")
```

The following code declares afile as an object variable and uses the CreateTextFile method (a function method) of the FileSystemObject object to create a new text file called testfile.txt in the folder, c:\. The CreateTextFile method also returns a TextStream object that can be used to read from or write to the file. From now on fs is a TextStream object so you can use the properties and methods of the TextStream class to read from or write to the file, c:\data\testfile.txt. You specify the True parameter to indicate that an existing file can be overwritten.

```
Dim afile As Object
```

```
Set afile = fs.CreateTextFile("c:\data\testfile.txt", True)
```

If you run the above code and the folder, c:\data don't exist you will get a run-time error. After the above code is executed and if you take no further action, the text file created by the above code will remain as a text file with no content in it.

For a complete listing of methods and properties of the FileSystemObject object consult Visual Basic Editor / Visual Basic Help / Table of Contents / Microsoft Visual Basic Documentation / Visual Basic Language Reference / Objects / FileSystemObject Object.

TextStream Object

The following code uses the WriteLine method of the TextStream object to write 2 lines of text to the file, c:\data\testfile.txt.

```
afile.WriteLine ("This is sentence 1.")
```

```
afile.WriteLine ("This is sentence 2.")
```

For a complete listing of methods and properties of the TextStream object consult Visual Basic Editor / Visual Basic Help / Table of Contents / Microsoft Visual Basic Documentation / Visual Basic Language Reference / Objects / TextStream Object.

The following shows the completed code.

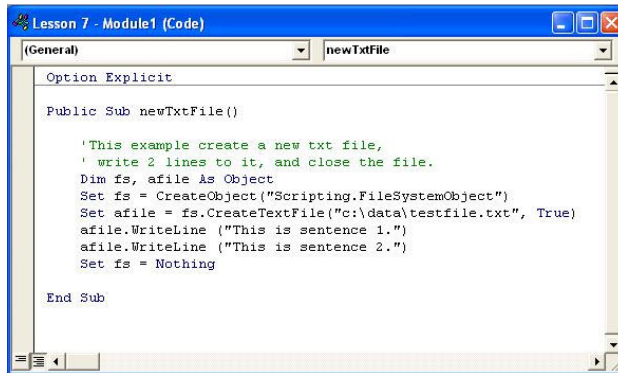
Completed Code

```
Public Sub newTxtFile()  
    'This example create a new txt file,  
    ' write 2 lines to it, and close the file.  
    Dim fs, afile As Object  
    Set fs = CreateObject("Scripting.FileSystemObject")  
    Set afile = fs.CreateTextFile("c:\data\testfile.txt", True)  
    afile.WriteLine ("This is sentence 1.")  
    afile.WriteLine ("This is sentence 2.")  
    Set fs = Nothing  
End Sub
```

To try this example follows these steps.

Steps

- 1) Start a blank document and start Visual Basic Editor.
 - 2) Insert a new module and keep the default name as Module1.
 - 3) Insert a new procedure (macro) and name it newTxtFile.
 - 4) Type in the above completed code (Figure 7.1).
 - 5) Run the macro and see the new text file, c:\data\testfile.txt created with 2 lines of texts (Figure 7.2).
-

Figure 7.1 *newTxtFile* macro


```

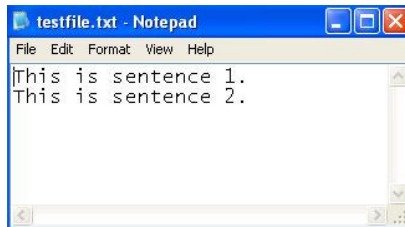
Option Explicit

Public Sub newTxtFile()

    'This example create a new txt file,
    ' write 2 lines to it, and close the file.
    Dim fs, afile As Object
    Set fs = CreateObject("Scripting.FileSystemObject")
    Set afile = fs.CreateTextFile("c:\data\testfile.txt", True)
    afile.WriteLine ("This is sentence 1.")
    afile.WriteLine ("This is sentence 2.")
    Set fs = Nothing

End Sub

```

Figure 7.2 *New text file created*


```

File Edit Format View Help
This is sentence 1.
This is sentence 2.

```

List Directory and Files

This example lists the folders and files of a directory.

Folder Object

The following code uses the `CreateObject` method (a function method) of the `Interaction` class to assign to `fs` a reference to the `Scripting.FileSystemObject` object. The assignment makes `fs` an object of the `FileSystemObject` class. From now on `fs` can use the methods and properties of the `FileSystemObject` object to manipulate the files system of the Windows operating system.

```
Set fs = CreateObject("Scripting.FileSystemObject")
```

The following code uses the `GetFolder` method of the `FileSystemObject` object to return a `Folder` object that represents the folder, `c:\`. From now on `fd` is an object of the `Folder` class so you can use the methods and properties of the `Folder` object to manipulate the folder, `c:\`.

```
Set fd = fs.GetFolder("c:\")
```

The following code uses the SubFolders property of the Folder object to list all the folders (files excluded) in the directory, c:\. The expression fd.SubFolders returns a Folders collection object consisting of all the folders contained in the directory, c:\, including hidden and system folders.

```
' List all folders (not file) in a directory

str = "Folders in " & "c:\" & vbCrLf _
      & "-----" & vbCrLf

For Each f2 In fd.SubFolders

    str = str & f2.Name

    str = str & vbCrLf

Next

MsgBox str
```

The following code uses the Files property of the Folder object to list all the files (folders excluded) in the directory, c:\. The expression fd.Files returns a Files collection object consisting of all the files contained in the directory, c:\, including hidden and system files.

```
' List all files (not folder) and folders in a directory

str = "Files in " & "c:\" & vbCrLf _
      & "-----" & vbCrLf

For Each f2 In fd.Files

    str = str & f2.Name

    str = str & vbCrLf

Next

MsgBox str
```

For a complete listing of methods and properties of the Folder object consult [Visual Basic Editor / Visual Basic Help / Table of Contents / Microsoft Visual Basic Documentation / Visual Basic Language Reference / Objects / Folder Object](#).

The following is the completed code.

Completed Code

```
Sub listDirectory()
    Dim fs, f2, fd, str
    Set fs = CreateObject("Scripting.FileSystemObject")
    Set fd = fs.GetFolder("c:\")

    ' List all folders (not file) in a directory
    str = "Folders in " & "c:\" & vbCrLf _
        & "-----" & vbCrLf
    For Each f2 In fd.SubFolders
        str = str & f2.Name
        str = str & vbCrLf
    Next
    MsgBox str

    ' List all files (not folder) and folders in a directory
    str = "Files in " & "c:\" & vbCrLf _
        & "-----" & vbCrLf
    For Each f2 In fd.Files
        str = str & f2.Name
        str = str & vbCrLf
    Next
    MsgBox str
End Sub
```

To try this example follows these steps.

Steps

- 1) Start a blank document and start Visual Basic Editor.
- 2) Use the same module as the one in the previous example.
- 3) Insert a new procedure (macro) and name it listDirectory.
- 4) Type in the above completed code (Figure 7.3).
- 5) Run the macro and the message box displaying a listing of the folders and files of the directory, c:\ (Figure 7.4).

Figure 7.3 *listDirectory* macro

```
Lesson 7 - Module1 (Code)
listDirectory

Option Explicit

Sub listDirectory()

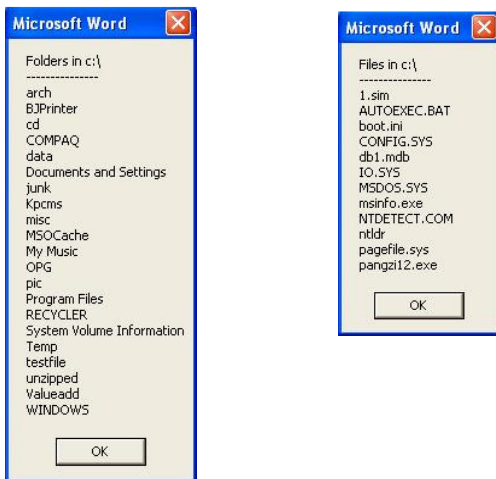
    Dim fs, f2, fd, str
    Set fs = CreateObject("Scripting.FileSystemObject")
    Set fd = fs.GetFolder("c:\")

    ' List all folders (not file) in a directory
    str = "Folders in " & "c:\" & vbCrLf _
        & "-----" & vbCrLf
    For Each f2 In fd.SubFolders
        str = str & f2.Name
        str = str & vbCrLf
    Next
    MsgBox str

    ' List all files (not folder) and folders in a directory
    str = "Files in " & "c:\" & vbCrLf _
        & "-----" & vbCrLf
    For Each f2 In fd.Files
        str = str & f2.Name
        str = str & vbCrLf
    Next
    MsgBox str

End Sub
```

Figure 7.4 *Result of listDirector* macro



Processing Multiple Word Documents

This example loops through all the Word document files in a given directory, replace every occurrence of John with Tom and write to a log file.

The following code uses the `CreateObject` method (a function method) of the `Interaction` class to assign to `fs` a reference to the `Scripting.FileSystemObject` object. The assignment makes `fs` an object of the `FileSystemObject` class. From now on `fs` can use the methods and properties of the `FileSystemObject` object to manipulate the files system of the Windows operating system.

```
Set fs = CreateObject("Scripting.FileSystemObject")
```

Create New Folder for Output Files

The following code creates a new folder (if it does not already exist), `change_dir` in the directory, `c:\testfile\` where the Word documents (to be processed) resides. You first check to see if the folder, `change_dir` already exists by using the `FolderExists` method of the `FileSystemObject` object. If the folder does not exist you use the `CreateFolder` method of the `FileSystemObject` object to create the folder, `change_dir` in the directory, `c:\testfile\`.

```
strPath = "c:\testfile\"
If Not fs.FolderExists(strPath & "change_dir") Then
    fs.CreateFolder (strPath & "change_dir")
End If
```

Create Log File

The following code uses the `CreateTextFile` method (a function method) of `fs` (a `FileSystemObject` object) to create a new text file, `logfile.txt` in the folder, `c:\testfile\`. The `CreateTextFile` method also returns to `ftxt` a `TextStream` object that can be used to read from or write to the file, `logfile.txt`. From now on `ftxt` is a `TextStream` object so you can use the properties and methods of the `TextStream` object to read from or write to the file, `c:\testfile\change_dir\logfile.txt`. You specify the `True` parameter to indicate that an existing file can be overwritten.

```
Set ftxt = fs.CreateTextFile(strPath & "change_dir\" & "logfile.txt", True)
```

You will use this log file to record the activities when you process the Word documents in the directory.

Processing Multiple Word Documents

The following For Each...Next loop reads each Word document files in the directory, c:\testfile\, replace every occurrence of John with Tom, save the changed Word document files to the folder, c:\testfile\change_dir . You use the GetFolder method of the FileSystemObject object to return a Folder object that represents the folder, c:\testfile\. The Files property of the Folder object returns a Files collection object consisting of all the files contained in the folder, c:\testfile\. You then use the For Each...Next loop to loop through all the Word document files in the Files Collection object representing the folder, c:\testfile\.

```
For Each fdoc In fs.GetFolder(strPath).Files
```

```
.....
```

```
.....
```

```
Next
```

The following code uses the Open method (a function method) of the Documents collection object to open a Word document file with the file name of fdoc.Name (fdoc.Name contains the name of the Word file) in the folder, c:\testfile\ as specified by the strPath string variable. The Open method returns an object (of the Document class) representing one of the Word document in the folder and assigns the reference to doc (an object variable). The Name property of the fdoc File object returns the name of one of the Word document file belonging to the Collection object representing all the Word document files in the folder c:\testfile\. The Word document is opened for read purpose only, as specified by the ReadOnly:=True parameter.

```
Set doc = Documents.Open(FileName:=strPath & fdoc.Name, ReadOnly:=True)
```

The following code uses the WriteLine method of ftxt (a TextStream object) to write to the log file, c:\testfile\change_dir\logfile.txt, the name of the Word document being read at the moment and the words count of the Word document.

```
ftxt.writeline ("***** File read *****")
```

```
ftxt.writeline ("Name of file = " & strPath & fdoc.Name)
```

```
ftxt.writeline ("Number of words in the document = " & doc.Words.Count)
```

The following code tests to see if the Word document contains the text John and if so record the finding in the log file. You use the Execute method (a function method) of the Find property of the Content property which is a Range object. The Content property of doc (a Document object) represents the content of the document.

```
If doc.Content.Find.Execute(FindText:="John") Then _
```

```
    ftxt.writeline ("Word found.")
```

The following code uses the Execute method (a function method) of the Find property to replace every occurrence of John by Tom.

```
doc.Content.Find.Execute FindText:="John", _
```

```
    ReplaceWith:="Tom", Replace:=wdReplaceAll
```

The following code uses the SaveAs method (a Sub procedure method) of doc (a Document object) to save the changed Word document with the specified name in the folder, c:\testfile\change_dir\. The Visual Basic

function Len(fdoc.Name) returns the number of characters of the name of the Word document. The expression Len(fdoc.Name) - 4 returns the number of characters of the name of the Word document, minus 4. The Visual Basic function Left(fdoc.Name, (Len(fdoc.Name) - 4)) returns the name of the Word document truncating the last 4 characters that is, the .doc portion. Basically what you are doing here is taking the name of the Word document, for example file1.doc, throw away the .doc portion and concatenate with _change.doc to form the name of the changed Word document as file1_change.doc, which is then saved in the folder, c:\testfile\change_dir\.

```
doc.SaveAs FileName:=strPath & "change_dir\" & _
    Left(fdoc.Name, (Len(fdoc.Name) - 4)) & _
    "_change" & ".doc"
```

The following code closes the Word document and go on to the next Word document in the folder until every Word document is processed.

```
For Each fdoc In fs.GetFolder(strPath).Files
```

```
    .....
```

```
    doc.Close
```

```
    ftxt.writeline (" ")
```

```
Next
```

The following is the completed code.

Completed Code

```
Sub procDoc()
    ' Loop through all doc.files in the directory
    ' replace every occurrence of "John" with "Tom"
    ' and write to a log file.
    Dim doc As Document
    Dim strPath As String
    Dim fs, fdoc, ftxt As Object
    Set fs = CreateObject("Scripting.FileSystemObject")
    strPath = "c:\testfile\"
    If Not fs.FolderExists(strPath & "change_dir") Then
        fs.CreateFolder (strPath & "change_dir")
    End If
    Set ftxt = fs.CreateTextFile(strPath & "change_dir\" & "logfile.txt", True)
    For Each fdoc In fs.GetFolder(strPath).Files
        Set doc = Documents.Open(FileName:=strPath & fdoc.Name, ReadOnly:=True)
        ftxt.writeline ("***** File read *****")
        ftxt.writeline ("Name of file = " & strPath & fdoc.Name)
        ftxt.writeline ("Number of words in the document = " & doc.Words.Count)
```

```

If doc.Content.Find.Execute(FindText:="John") Then _
    ftxt.writeline ("Word found.")
doc.Content.Find.Execute FindText:="John", _
    ReplaceWith:="Tom", Replace:=wdReplaceAll
doc.SaveAs FileName:=strPath & "change_dir\" & _
    Left(fdoc.Name, (Len(fdoc.Name) - 4)) & _
    "_change" & ".doc"
doc.Close
ftxt.writeline (" ")

```

Next

Set fs = Nothing

End Sub

To try this example follows these steps.

Steps

- 1) Create the folder, c:\testfile\ and 3 Word documents, file1.doc, file2.doc, file3.doc in the folder, c:\testfile\ (Figure 7.6).
- 2) Start a blank document and start Visual Basic Editor.
- 3) Use the same module as the one in the previous example.
- 4) Insert a new procedure (macro) and name it procDoc.
- 5) Type in the above completed code (Figure 7.5).
- 6) Run the macro and all the occurrence of John will be replaced by Tom and the 3 changed Word document files and the log file are written to the folder, c:\testfile\change_dir\ (Figure 7.7).

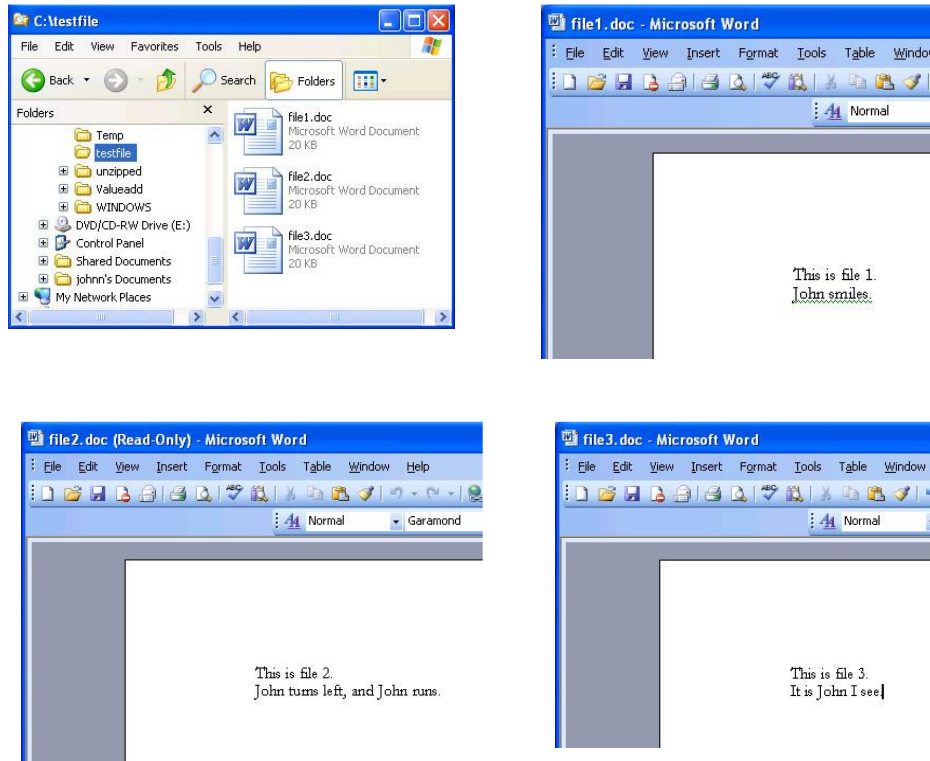
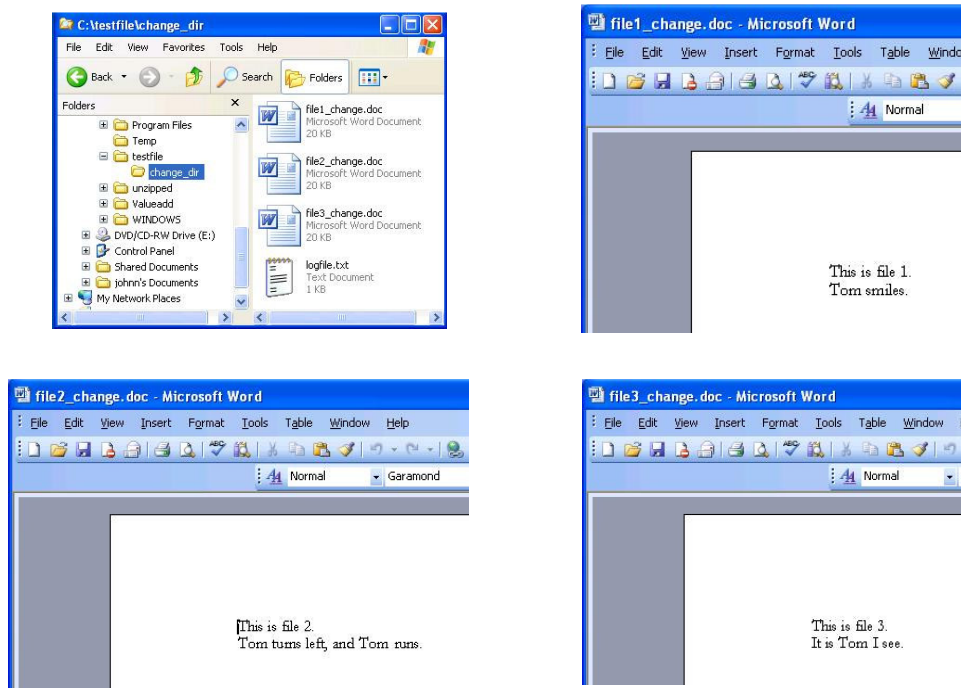
Figure 7.5 *procDoc* macro

```

Option Explicit

Sub procDoc()
    ' Loop through all doc.files in the directory
    ' replace every occurrence of "John" with "Tom"
    ' and write to a log file.
    Dim doc As Document
    Dim strPath As String
    Dim fs, fdoc, ftxt As Object
    Set fs = CreateObject("Scripting.FileSystemObject")
    strPath = "c:\testfile\"
    If Not fs.FolderExists(strPath & "change_dir") Then
        fs.CreateFolder (strPath & "change_dir")
    End If
    Set ftxt = fs.CreateTextFile(strPath & "change_dir\" & "logfile.txt", True)
    For Each fdoc In fs.GetFolder(strPath).Files
        Set doc = Documents.Open(FileName:=strPath & fdoc.Name, ReadOnly:=True)
        ftxt.writeline ("***** File read *****")
        ftxt.writeline ("Name of file = " & strPath & fdoc.Name)
        ftxt.writeline ("Number of words in the document = " & doc.Words.Count)
        If doc.Content.Find.Execute(FindText:="John") Then _
            ftxt.writeline ("Word found.")
            doc.Content.Find.Execute FindText:="John", _
                ReplaceWith:="Tom", Replace:=wdReplaceAll
        doc.SaveAs FileName:=strPath & "change_dir\" & _
            Left(fdoc.Name, (Len(fdoc.Name) - 4)) & _
            "_change" & ".doc"
        doc.Close
        ftxt.writeline (" ")
    Next
    Set fs = Nothing
End Sub

```


Figure 7.6 Input Word document files in *c:\testfile* folderFigure 7.7 Changed Word document files in *c:\testfile\change_dir* folder



```
logfile.txt - Notepad
File Edit Format View Help
***** File read *****
Name of file = c:\testfile\file1.doc
Number of words in the document = 11
Word found.

***** File read *****
Name of file = c:\testfile\file2.doc
Number of words in the document = 16
Word found.

***** File read *****
Name of file = c:\testfile\file3.doc
Number of words in the document = 13
Word found.
```

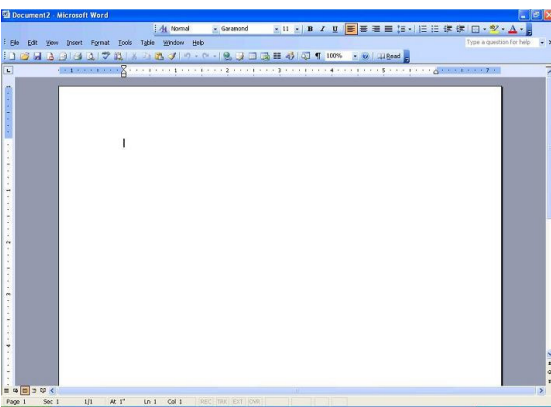
This concludes Chapter 7.

Chapter 8 Form Programming

What is a Form?

A form is a user interface that accepts user's input, interact with the user and perform some operation. In Visual Basic a form is also called a Dialog Box. You've seen plenty of form in action when you use computers. For example when you start the Word application the Microsoft Word window is a form (Figure 8.1). Most forms display as a window like this one.

Figure 8.1 *A Microsoft Word window*



For another example, Select Table / Insert / Table from the Word menu and the Insert Table window appear (Figure 8.2). This is also a form. You will go through creating a form similar to this in the following section to learn the principle and technique of form programming.

Figure 8.2 *Word Insert Table dialog box*



Form and Controls

Forms are objects. When you create a form you create an instance of a class. The UserForm class is a member class of MSForms library. You will use the UserForm form to create a dialog box similar to the Insert Table dialog box of Word (Figure 8.2). A UserForm form can contain other objects such as controls. A Control is an object you place on a form that has its own properties, methods, and events. You use controls to receive user input, display output, and trigger event procedures. Because controls are objects you manipulate them using methods, properties and events associated with the controls. You will learn to use some of the controls in the two examples in this chapter.

You've seen two examples of forms, the Microsoft Word window when you start Word (Figure 8.1) and the Insert Table dialog box when you want to insert a new table in your Word document (Figure 8.2). Both are forms that contain controls but the Word window (form) is much more complicated and contains many more controls and objects than the Insert Table dialog box.

Form Programming Example – Insert a New Table

This example illustrates the principle and technique of form programming. It displays a form (dialog box) in which you input certain criteria to insert a table in your document. The form you see in this example is similar to the Word Insert Table dialog box (Figure 8.2).

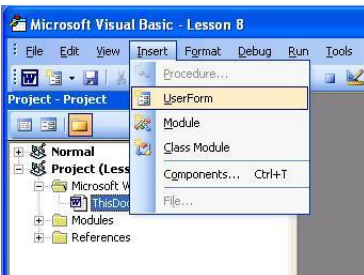
UserForm Object

You first create a UserForm form. The form contains various controls you place on the form. These controls facilitate you or other users to interact with the form.

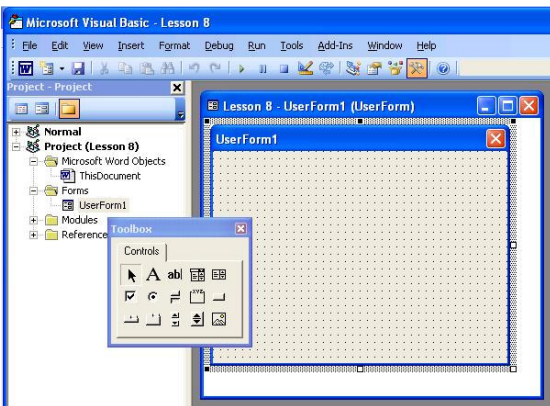
Follows these step to create a UserForm form.

Steps

- 1) Start a blank document and start Visual Basic Editor.
 - 2) From the Visual Basic Editor menu select Insert / UserForm (Figure 8.3).
-

Figure 8.3 *Insert UserForm form*

A UserForm form and the Toolbox appears (Figure 8.4). The default name of the UserForm form is UserForm1 (Figure 8.4).

Figure 8.4 *UserForm form*

Toolbox and Controls

Figure 8.5 shows the Toolbox with the Controls which you drag and drop on a form. Figure 8.6 shows a list of the different type of controls available in the tool box. You can program your own custom controls and add them to the toolbox. In this example you will use the Label, CheckBox, CommandButton, Frame, OptionButton, ComboBox controls in your form.

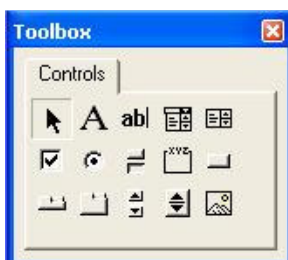















Figure 8.5 *Toolbox*

Figure 8.6 *Controls in Toolbox*

Icon	Control	Icon	Control	Icon	Control
	CheckBox		Label		Select Objects
	ComboBox		ListBox		SpinButton
	CommandButton		MultiPage		TapStrip
	Frame		OptionButton		TextBox
	Image		ScrollBar		ToggleButton

Change Name of UserForm Form and Caption Property

You've inserted a new UserForm form and the default name and the default Caption property had been set to the same name, UserForm1 (Figure 8.4). Next you will change the name of the UserForm form from UserForm1 to frmTable and change the Caption property of the UserForm form from UserForm1 to Insert Table (Figure 8.7).

Note

The caption property is a property of the UserForm form whereas the name of the UserForm form is not a property or a method of the UserForm form.

Follow these steps to change the name and the Caption property of the UserForm form.

Steps

- 1) Right click anywhere in the UserForm form and select Properties to display the Properties window of the UserForm1 form (Figure 8.7).
- 2) Change the name of the UserForm form from UserForm1 to frmTable (Figure 8.7). See the change being reflected in both the Properties window and the UserForm form (Figure 8.8).
- 3) Change the Caption property from UserForm1 to Insert Table (Figure 8.7). See the change being reflected in both the Properties window and the UserForm form (Figure 8.8).

Figure 8.7 Change UserForm form name and Caption property

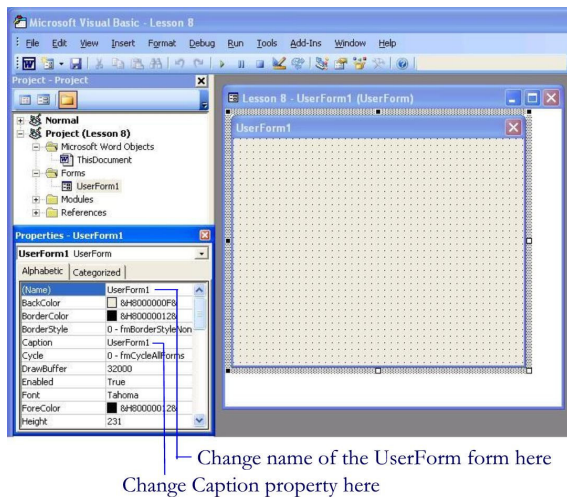
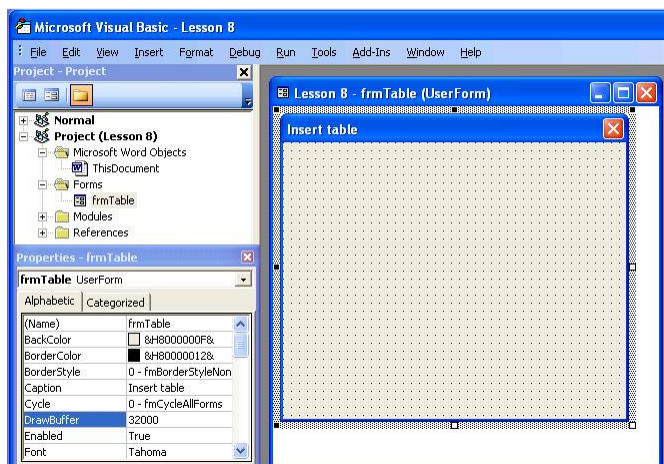


Figure 8.8 New UserForm form name and Caption property



Add Controls to UserForm Form

Next you'll add Controls to the UserForm form. You'll add the Label, CheckBox, CommandButton, Frame, OptionButton and ComboBox controls to the UserForm form. To add these controls to the UserForm form follow these steps.

Steps

- 1) If the Toolbox is not visible, make sure the UserForm form is in focus (that is, selected by clicking it), then select View / Toolbox from the Visual Basic editor menu.
- 2) Drag and drop a Label control from the Toolbox into the UserForm form (Figure 8.9). After you drop the Label control on the form you can use the mouse to resize it and move it around.
- 3) From the Toolbox, drop another Label control and 2 ComboBox controls into the form.

-
- 4) From the Toolbox drop a Frame control into the form and then drop 2 OptionButton controls into the Frame control. You must follow the order exactly by dropping the Frame control first and then dropping the 2 OptionButton controls into the Frame control. If you don't follow the exact order of dropping the Frame control first followed by the 2 OptionButton controls inside the Frame control your code won't work.
 - 5) From the Toolbox drop 1 CheckBox control and 2 CommandButton controls into the form.
 - 6) Rearrange, resize and move the controls until it look like that in Figure 8.10. All the controls have default Caption properties. The 2 ComboBox controls have no Caption properties.
-

Figure 8.9 *Label control*

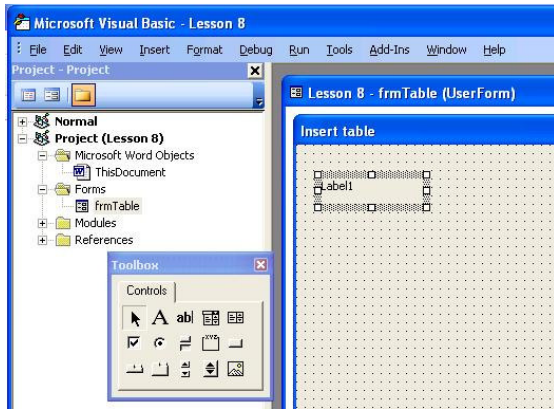
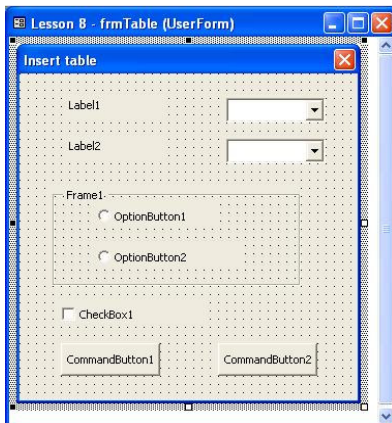


Figure 8.10 *Label, CheckBox, CommandButton, Frame, OptionButton and ComboBox controls*



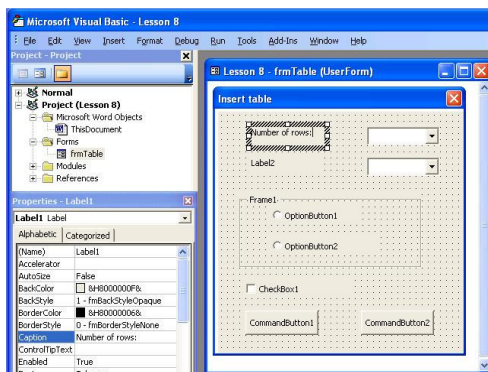
Change Name and Caption Property of Control

After you've dropped the Label, CheckBox, CommandButton, Frame, OptionButton and ComboBox controls into the UserForm form they have default names and default Caption properties. Next you'll change the Caption properties of the two Label controls but keep the default names. Follow these steps to do that.

Steps

- 1) Display the Properties window of the 1st Label control by either selecting the Label control or by right clicking it and select Properties.
- 2) Change the Caption property of the Label control to (Number of rows:) (Figure 8.11).
- 3) Keep the default name of the Label control as Label1.
- 4) Similarly change the Caption property of the 2nd Label control to (Number of columns:) and keep the default name as Label2.

Figure 8.11 Change Label control's Caption property

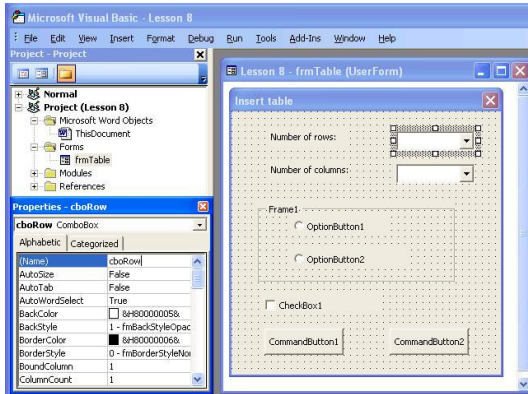


Next you'll change the names of the two ComboBox controls. The ComboBox control has no Caption property. Follow these steps to do that.

Steps

- 1) Display the Properties window of the 1st ComboBox control by either selecting the ComboBox control or by right clicking it and select Properties.
- 2) Change the name of the 1st ComboBox control to cboRow (Figure 8.12).
- 3) Similarly change the name of the 2nd ComboBox control to cboColumn.

Figure 8.12 *Change name of ComboBox control*



Next you'll change the names or Caption properties of the remaining controls on the form. Follow these steps to do that.

Steps

- 1) Change the names and Caption properties of the two OptionButton, the CheckBox and the two CommandButton controls to that as shown in Figure 8.14. The new Caption properties of these controls display like that in Figure 8.13.
- 2) Change the Caption property of the Frame control to Border setting (Figure 8.13) and keep the default name as Frame1.
- 3) The Caption properties of all the controls on the UserForm form should display like that in Figure 8.13.

Figure 8.13 *Your finished UserForm form*

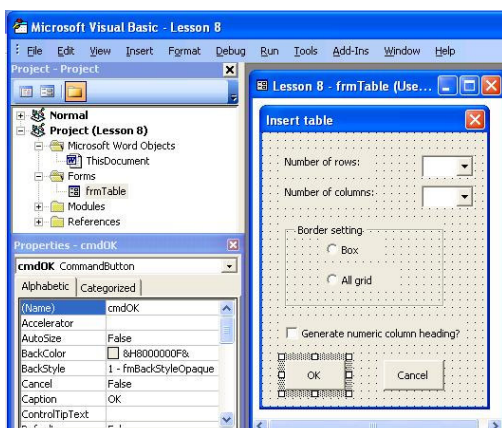


Figure 8.14 *Names and Caption properties of controls*

Control	Name	Caption Property
Label	Label1	Number of rows:
Label	Label2	Number of columns:
ComboBox (1 st)	cboRow	(None)
ComboBox (2 nd)	cboColumn	(None)
Frame	Frame1	Border setting
OptionButton	optBox	Box
OptionButton	optAll	All grid
CheckBox	chkNum	Generate numeric column heading?
CommandButton	cmdOK	OK
CommandButton	cmdCancel	Cancel

Show Method of UserForm Object

Next you'll create a new Macro for this form programming example. The macro (when invoked) will load and display the UserForm form, frmTable you've set up. Recall that you've given the name of frmTable to the UserForm form, which in essence make frmTable an object of the UserForm class. That means that frmTable has available all the methods and properties of the UserForm class.

The following code uses the Show method of the UserForm object to display the UserForm form, frmTable.

Completed Code

```
Public Sub addTable()
    frmTable.Show
End Sub
```

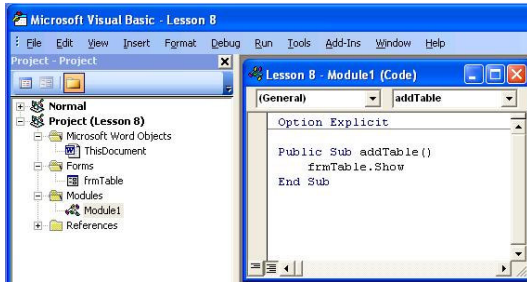
For a complete listing of the methods and properties of the UserForm class, consult Visual Basic Help.

Follow these steps to add a new Macro for this example.

Steps

- 1) Insert a new module (by selecting Insert / Module from the Visual Basic editor menu) and keep the default module name as Module1.
- 2) With the Module1 window in focus (selected) insert a new procedure (macro) and name it addTable.
- 3) Type in the above completed code (Figure 8.15) and save it.

Figure 8.15 *addTable* macro



Initialize Event of UserForm Object

The UserForm object, besides having methods and properties, also has events associated with it. Next you'll code an event procedure (event handler) to perform some action when the UserForm form, frmTable is initialized (loaded into the computer memory and displayed). The action to perform includes populating the ComboBox control, cboRow with an array of integers from 1 to 30 and populating the ComboBox control, cboColumn with an array of integers from 1 to 10. You do these using the AddItem method (a Sub procedure method) of the ComboBox control (object). In addition to that the Text properties of the two ComboBox controls are set equal to an initial value of 1. You set the Value property of the OptionButton control, optAll to be True (=1).

For a complete listing of the events associated with the UserForm object consult the Object Browser and Visual Basic Help.

The following shows the completed code for the Initialize event procedure for the UserForm form, frmTable.

Completed Code

```
Private Sub UserForm_Initialize()  
    Dim intIndex  
    For intIndex = 1 To 30  
        cboRow.AddItem intIndex  
    Next  
    cboRow.Text = 1  
    For intIndex = 1 To 10  
        cboColumn.AddItem intIndex  
    Next  
    cboColumn.Text = 1  
    optAll.Value = True  
End Sub
```

Follow these steps to code the event procedure for the Initialize event of the UserForm form, frmTable.

Steps

- 1) Make sure the frmTable (UserForm) window is visible, if not select the folder, frmTable in the Project Explorer.
- 2) Display the frmTable (Code) window by double clicking anywhere in the frmTable (UserForm) window, or right clicking it and select View Code.
- 3) The code window has two dropdown boxes. The left dropdown box contains a list of all the controls (cboColumn, cboRow, chkNum) and UserForm. Select UserForm (Figure 8.16).
- 4) With the left dropdown box of the code window displaying UserForm, the right dropdown box contains a list of events associated with the UserForm object. Select Initialize, the Initialize event of the UserForm object (Figure 8.16).
- 5) Visual Basic editor automatically generates the heading `Private Sub UserForm_Initialize()` and the ending `End Sub` of the event procedure for you. Type in the above code (Figure 8.17) between the heading and the ending of the event procedure.

Figure 8.16 *Initialize event*

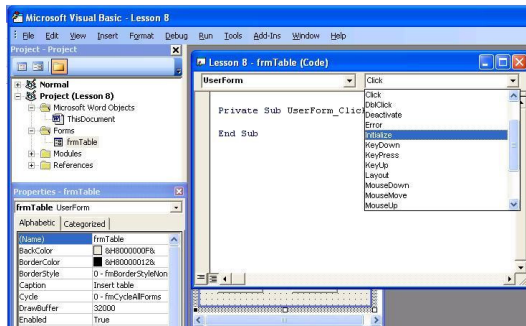
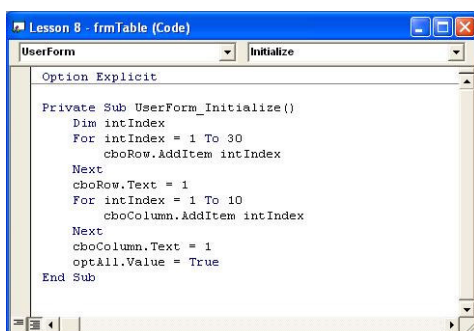


Figure 8.17 *UserForm_Initialize event procedure*



Click Event of CommandButton Control, cmdOK

Next you'll code an event procedure to perform some operation when you click the OK button on the UserForm form, frmTable. The operation to perform is to insert a table according to some criteria you provide through the other controls on the UserForm form. Your clicking of the OK button (a CommandButton control) on the UserForm form, frmTable triggers the Click event of the CommandButton control and the code you write in the event procedure will be called upon to perform the operation.

The following code uses the Add method (a function method) of the Tables class to insert a table into the document and assign the reference to the inserted table to objTable, an object variable of the Table class. The number of rows and the number of columns of the new table are specified by the Text properties of the ComboBox controls, cboRow and cboColumn respectively. Recall that you've coded to populate these two ComboBox controls in the event procedure when the UserForm form, frmTable is displayed (initialized) and you've also set the Text properties of these two controls to 1. If you don't select other values for the number of rows and columns the value of 1 will be used in the following code for cboRow.Text and cboColumn.Text. If you select other values (other than the default value of 1) the values you select will be used in the following code for cboRow.Text and cboColumn.Text.

```
Set objTable = ActiveDocument.Tables.Add _  
    (Range:=Selection.Range, NumRows:=cboRow.Text, _  
    NumColumns:=cboColumn.Text)
```

You've put in two OptionButton controls inside a Frame control on the UserForm form, frmTable and name these two controls optBox and optAll. The fact that these two OptionButton controls are inside a Frame control means that only one of the Value properties of the two OptionButton controls can be True (or 1). That is, when you click one of the OptionButton you make the Value property of that OptionButton control equal to True (=1) and the Value property of the other OptionButton control get set to False (=0) automatically. So the following code set the border setting of the table to be a box (no inside grid line) when optBox.Value is True (=1), that is, when you select the OptionButton control, optBox on the UserForm form, frmTable. It set the border setting of the table to be all grid (not a box) when optBox.Value is False (=0), that is, when you select the OptionButton control, optAll.

With .Borders

```
    If optBox.Value Then  
        .InsideLineStyle = wdLineStyleNone  
    Else  
        .InsideLineStyle = wdLineStyleSingle  
    End If
```

You've put in a Checkbox control on the UserForm form, frmTable and name it chkNum. If you check (select) the CheckBox control, chkNum on the UserForm form, frmTable chkNum.Value is set to True (=1). In such a case the For...Next loop generates a numeric column heading in the first column.

```
    If chkNum.Value Then  
        For intIndex = 1 To cboRow.Text  
            objTable.Cell(intIndex, 1).Range.Text = intIndex
```

Next

End If

The following shows the completed code for the Click event procedure for the CommanButton control, cmdOK.

Completed Code

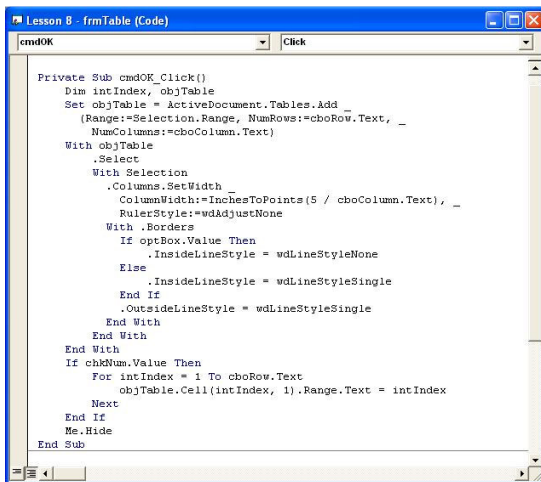
```
Private Sub cmdOK_Click()  
    Dim intIndex, objTable  
    Set objTable = ActiveDocument.Tables.Add _  
        (Range:=Selection.Range, NumRows:=cboRow.Text, _  
        NumColumns:=cboColumn.Text)  
    With objTable  
        .Select  
        With Selection  
            .Columns.SetWidth _  
                ColumnWidth:=InchesToPoints(5 / cboColumn.Text), _  
                RulerStyle:=wdAdjustNone  
            With .Borders  
                If optBox.Value Then  
                    .InsideLineStyle = wdLineStyleNone  
                Else  
                    .InsideLineStyle = wdLineStyleSingle  
                End If  
                .OutsideLineStyle = wdLineStyleSingle  
            End With  
        End With  
    End With  
    If chkNum.Value Then  
        For intIndex = 1 To cboRow.Text  
            objTable.Cell(intIndex, 1).Range.Text = intIndex  
        Next  
    End If  
    Me.Hide  
End Sub
```

Follow these steps to code the event procedure to handle the Click event of the CommandButton control, cmdOK.

Steps

- 1) Make sure the frmTable (UserForm) window is visible, if not select the frmTable folder in the Project Explorer.
- 2) Display or make visible the frmTable (Code) window by double clicking anywhere in the frmTable (UserForm) window, or right clicking the form window and select View Code.
- 3) The code window has two dropdown boxes. The left dropdown box contains a list of all the controls (cboColumn, cboRow, chkNum) and UserForm. Select cmdOK (Figure 8.18).
- 4) With the left dropdown box of the code window displaying cmdOK, the right dropdown box contains a list of events associated with the CommandButton control, cmdOK. Select Click, the Click event of the CommandButton control, cmdOK.
- 5) Visual Basic editor automatically generates the heading Private Sub cmdOK_Click() and the ending End Sub of the event procedure for you. Type in the above code (Figure 8.18) between the heading and the ending of the event procedure.

Figure 8.18 cmdOK_Click event procedure



```
Lesson 8 - frmTable (Code)
cmdOK Click
Private Sub cmdOK_Click()
    Dim intIndex, objTable
    Set objTable = ActiveDocument.Tables.Add _
        (Range:=Selection.Range, NumRows:=cboRow.Text, _
        NumColumns:=cboColumn.Text)
    With objTable
        .Select
        With Selection
            .Columns.SetWidth _
                ColumnWidth:=InchesToPoints(5 / cboColumn.Text), _
                RulerStyle:=wdAdjustNone
        With Borders
            If optBox.Value Then
                .InsiLineStyle = wdLineStyleNone
            Else
                .InsiLineStyle = wdLineStyleSingle
            End If
            .OutsiLineStyle = wdLineStyleSingle
        End With
    End With
    End With
    If chkNum.Value Then
        For intIndex = 1 To cboRow.Text
            objTable.Cell(intIndex, 1).Range.Text = intIndex
        Next
    End If
    Me.Hide
End Sub
```

Click Event of CommandButton Control, cmdCancel

Next you'll code an event procedure to unload the UserForm form, frmTable from the computer memory when you select the Cancel button on the UserForm form. Your clicking of the Cancel button (a CommandButton control) on the UserForm form, frmTable triggers the Click event of the CommanButton control and the code you write in the event procedure will be called upon to perform the operation.

The following code uses the Unload statement (a Visual Basic statement, not a method or property of an object) to remove the current active UserForm form from the computer memory. Me is a special object that refers to the currently active UserForm form, which is frmTable.

Completed Code

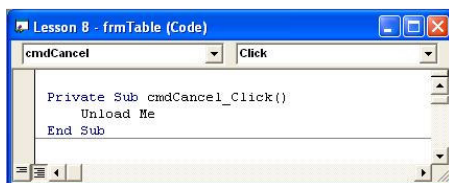
```
Private Sub cmdCancel_Click()
    Unload Me
End Sub
```

Follow these steps to code the event procedure to handle the Click event of the CommandButton control, cmdCancel.

Steps

- 1) Make sure the frmTable (UserForm) window is visible, if not select the folder, frmTable in the Project Explorer.
- 2) Display or make visible the frmTable (Code) window by double clicking anywhere in the frmTable (UserForm) window, or right clicking it and select View Code.
- 3) The code window has two dropdown boxes. The left dropdown box contains a list of all the controls (cboColumn, cboRow, chkNum) and UserForm. Select cmdCancel (Figure 8.19).
- 4) With the left dropdown box of the code window displaying cmdCancel, the right dropdown box contains a list of the events associated with the CommandButton control, cmdCancel. Select Click, the Click event of the CommandButton control, cmdCancel (Figure 8.19).
- 5) Visual Basic editor automatically generates the heading Private Sub cmdCancel_Click() and the ending End Sub of the event procedure for you. Type in the above code (Figure 8.19) between the heading and the ending of the event procedure.

Figure 8.19 *cmdCancel_Click event procedure*



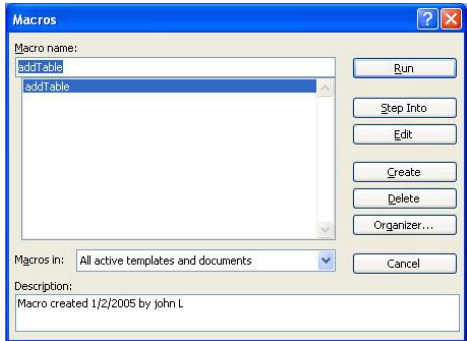
Run Macro

Now you're ready to see your macro in action. To run the addTable macro follows these steps.

Steps

- 1) From the Word menu select Tools / Macro /Macros.
- 2) When the Macro window appears selects addTable in the Macro Name pane and selects the Run button (Figure 8.20).

Figure 8.20 Run addTable macro



Steps

- 1) The Insert Table dialog box appears (Figure 8.21).
- 2) Scroll down the ComboBox controls to select 5 as the number of rows and 4 as the number of columns for the new table you want to insert.
- 3) Select the All grid OptionButton control.
- 4) Check the CheckBox control to generate numeric column heading for your new table. Select the OK button.
- 5) The new table with 5 rows and 4 columns and numeric column heading has been generated in your document (Figure 8.22).

Figure 8.21 Insert Table dialog box

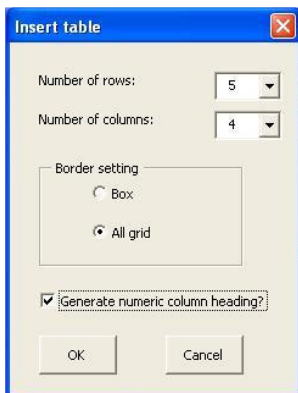


Figure 8.22 *New table added*

1			
2			
3			
4			
5			

Form Programming Example – Build Table of Index

You'll go through one more example in this chapter to learn about form programming. You'll use a Label control, a TextBox control and two CommanButton controls in this example to code a macro which help you search words (word or words) and add them to the index entries. The macro then updates the table of index. If there is no table of index it will be created. The only new control you use in this example is the TextBox control. You've used the other controls, the Label and the CommandButton controls in the previous example.

Follow these steps to add the new UserForm form and the controls.

Steps

- 1) Start a new Word document with a few pages of text.
- 2) Start Visual Basic Editor.
- 3) Insert a new UserForm form and change its name from the default name of UserForm1 to frmIndex. Change its Caption property from the default property of UserForm1 to Mark Index Entry (Figure 8.23).
- 4) Use the Toolbox to drag and drop one Label control, one TextBox control and two CommanButton controls into the UserForm form, frmIndex. Change the names and Caption properties of the controls as that shown in Figure 8.24. The UserForm form, frmIndex with these controls should like that in Figure 8.23.
- 5) See Figure 8.25 for the names and Caption properties of the Label control, Label1, the TextBox control, txtText, and the two CommandButton controls, cmdFind and cmdExit.

Figure 8.23 UserForm form frmIndex

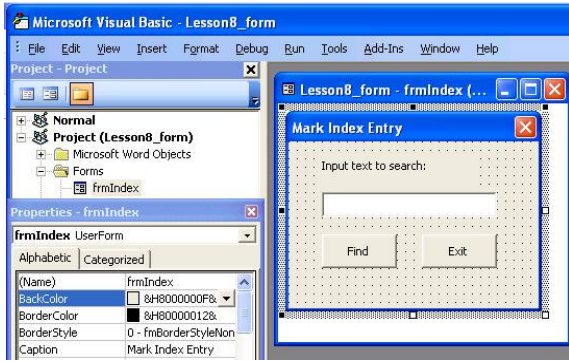
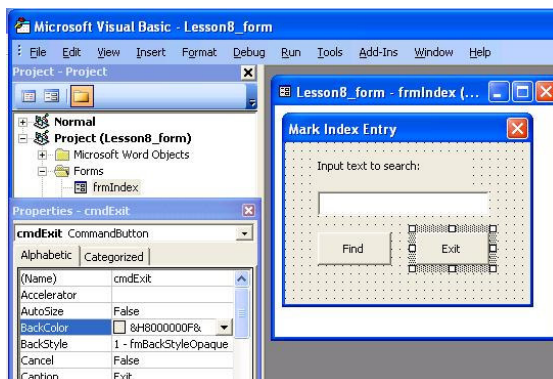
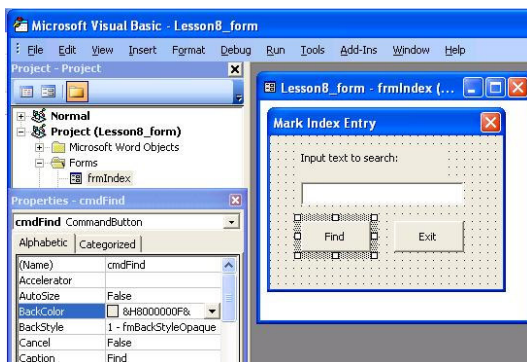
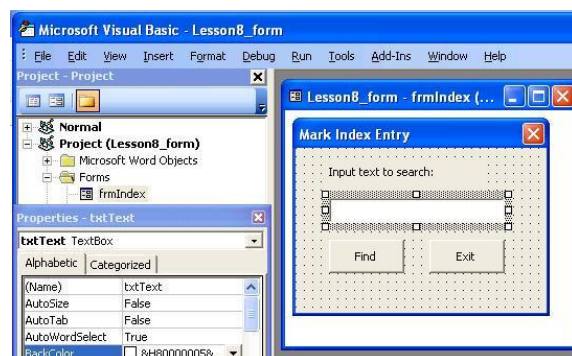
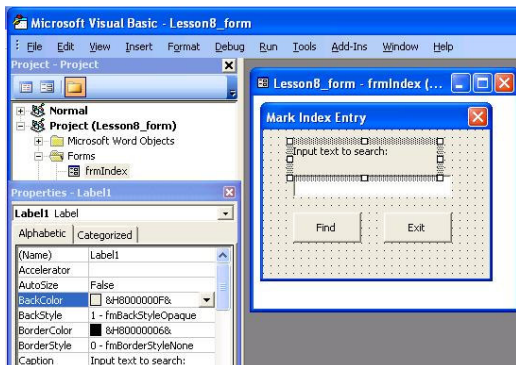


Figure 8.24 Names and Caption properties of controls

Control	Name	Caption Property
Label	Label1	Input text to search:
TextBox	txtText	(None)
CommandButton	cmdFind	Find
CommandButton	cmdExit	Exit

Figure 8.25 Label, TextBox, and CommandButton controls



Show Method of UserForm Object

Next you'll create a new Macro for this form programming example. The macro (when invoked) will load and display the UserForm form, frmIndex you've set up. Recall that you've given the name of frmIndex to your UserForm form, which in essence make frmIndex an object of the UserForm class. That means frmIndex has available all the methods and properties of the UserForm class.

The following code uses the Show method of the UserForm object to display the UserForm form, frmIndex.

Completed Code

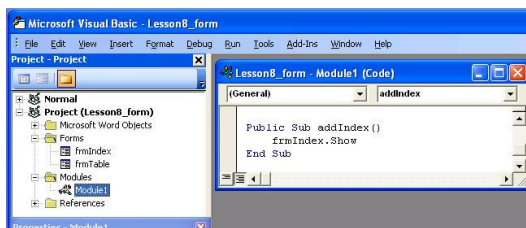
```
Public Sub addIndex()
    frmIndex.Show
End Sub
```

Follow these steps to add a new Macro for this example.

Steps

- 1) Insert a new module (select Insert / Module from the Visual Basic editor menu) and keep the default module name as Module1.
- 2) With the Module1 window in focus (selected) insert a new procedure (macro) and name it addIndex.
- 3) Type in the above completed code (Figure 8.26) and save it.

Figure 8.26 *addIndex* macro



Click Event of CommandButton Control, cmdFind

Next you'll code an event procedure to perform some operation when you click the Find button on the UserForm form, frmIndex after you input some text in the Text box control, txtText. The operation to perform is to search the document for the occurrence of the input words (word or words) and set the words found to be an index entry depending on the decision by you. If you decide you want the words found in the document to be inserted as an index entry then the table of index can be updated, again, depending on the decision by you.

The following code takes what you input into the Textbox control, txtText and assign it to strInput, a string variable. The words you type into the Textbox control, txtText becomes the Text property of the Textbox control, txtText. The Visual Basic function Trim is used to trim off any leading blank and trailing blank of the text string you input into the Textbox control, txtText.

```
strInput = Trim(txtText.Text)
```

The following For...Next loop loops through every paragraph of the active document to search for the words you input into the Textbox control, txtText.

```
Set doc = ActiveDocument

For Each para In doc.Paragraphs
.....
.....

Next
```

The following code uses the Find property of the Selection object to search for the occurrence of the words contained in strInput, a string variable. You use the Select method (a sub procedure method) of the Range class to select the current paragraph. The Select method returns the reference to the current paragraph to the Selection object. You use the Find property (object) of the Selection object (which now represents the current paragraph) to specify the criteria of your search. The (Forward = True) property of the Find object specifies that you search forward through the paragraph. You use the ClearFormatting method (a Sub procedure method) to remove any formatting from the current paragraph being searched. The (MatchWholeWord = True) property specifies that the find operation locates only entire words and not text that's part of a larger word. The (MatchCase = False) property specifies the find operation is not case sensitive. The Execute method (a function method) runs the specified find operation and returns True (and assigns to the Boolean variable blnFound) if the search is successful. If the search is successful the words found is automatically selected, that is, the words become the Selection object so any subsequent use of the Selection object refers to the words being searched and found.

```
para.Range.Select

With Selection.Find

    .Forward = True

    .ClearFormatting

    .MatchWholeWord = True

    .MatchCase = False

    blnFound = .Execute(FindText:=strInput)

End With
```

If the words being input is found the Boolean variable, blnFound is set to True and the following code uses the MsgBox function to prompt you to ask if you want the found words (now selected) to be marked as an index entry.

```
If blnFound Then

    strMsg = "" & strInput & "" & " found, " & _

        "mark as index entry?"

    intAnswer = MsgBox(strMsg, intBtn)
```

If you click the Yes button in the MsgBox window that you want the found (selected) words to be marked as an index entry the following code uses the MarkEntry method (a function method) of the Indexes object to mark the selected words as an index entry. The (Range:=Selection.Range) parameter is required to specify the location of the index entry. The XE field is inserted after the selected words. The (Entry:=Selection.Range.Text) parameter specifies that the selected words is to appear in the index.

```
If intAnswer = vbYes Then
    ' User chose Yes.
    doc.Indexes.MarkEntry Range:=Selection.Range, _
        Entry:=Selection.Range.Text
```

The following code uses the MsgBox function to ask you if you want to update the index table.

```
strMsg = "Update index table?"
intAnswer = MsgBox(strMsg, intBtn)
```

If you click the Yes button in the MsgBox window that you want to update the index table the following code checks (using the Count property of the Indexes object) to see if an index table exists. If there is no index table in the document (that is, doc.Indexes.Count = 0) a new index table is created. You create the new index table using the Add method (a function method) of the Indexes object. Before you create the new index table you set the object variable, range2 to be the content of the document. Then you use the Collapse method (a Sub procedure method) of the Range object to collapse rang2 that refers to the entire content of the document. The (Direction:=wdCollapseEnd) parameter specifies that after the collapse rang2 is to be located at the end of the document, that is where you'll insert the new index table. You then use the Add method of the Indexes object to add the new index table at the end of the document.

```
If intAnswer = vbYes Then
    If doc.Indexes.Count = 0 Then
        Set range2 = doc.Content
        range2.Collapse Direction:=wdCollapseEnd
        doc.Indexes.Add Range:=range2, Type:=wdIndexRunin
    End If
```

After you've made sure that an index table exists the following code uses the Update method (a Sub procedure method) of the Index class to update all the index tables in the document.

```
For Each idxTable In doc.Indexes
    idxTable.Update
Next
```

The following code uses the MsgBox function to ask if you want to continue searching. If you click the No button on the MsgBox window you will exit this Click event procedure and return to the UserForm form, frmIndex. If you click the Yes button on the MsgBox window you will go on to search the same words (which you input) in the next paragraph in the For...Next loop.

```
For Each para In doc.Paragraphs
    .....
    strMsg = "Search more?"
    intAnswer = MsgBox(strMsg, intBtn)
    If intAnswer = vbNo Then Exit Sub
    .....
Next
```

The following is the completed code of the event procedure to handle the Click event of the CommandButton control, cmdFind.

Completed Code

```
Private Sub cmdFind_Click()
    Dim doc, para, strInput, blnFound, range2
    Dim strMsg, intBtn, intAnswer, idxTable
    ' Define buttons.
    intBtn = vbYesNo + vbInformation + vbDefaultButton2
    strInput = Trim(txtText.Text)
    Set doc = ActiveDocument
    For Each para In doc.Paragraphs
        para.Range.Select
        With Selection.Find
            .Forward = True
            .ClearFormatting
            .MatchWholeWord = True
            .MatchCase = False
            blnFound = .Execute(FindText:=strInput)
        End With
        If blnFound Then
            strMsg = "" & strInput & "" & " found, " & _
                "mark as index entry?"
            intAnswer = MsgBox(strMsg, intBtn)
            If intAnswer = vbYes Then
                ' User chose Yes.
```



```

doc.Indexes.MarkEntry Range:=Selection.Range, _
    Entry:=Selection.Range.Text
strMsg = "Update index table?"
intAnswer = MsgBox(strMsg, intBtn)
If intAnswer = vbYes Then
    If doc.Indexes.Count = 0 Then
        Set range2 = doc.Content
        range2.Collapse Direction:=wdCollapseEnd
        doc.Indexes.Add Range:=range2, Type:=wdIndexRunin
    End If
    For Each idxTable In doc.Indexes
        idxTable.Update
    Next
End If
End If
strMsg = "Search more?"
intAnswer = MsgBox(strMsg, intBtn)
If intAnswer = vbNo Then Exit Sub
End If
Next
Exit Sub
End Sub

```

Follow these steps to code the event procedure to handle the Click event of the CommanButton control, cmdFind.

Steps

- 1) Make sure the frmIndex (UserForm) window is visible, if not select the folder, frmIndex in the Project Explorer.
- 2) Display or make visible the frmIndex (Code) window by double clicking anywhere in the frmIndex (UserForm) window, or right clicking the form window and select View Code.
- 3) The code window has two dropdown boxes. The left dropdown box contains a list of all the controls (Label1, txtText, cmdFind, cmdExit) and UserForm. Select cmdFind (Figure 8.27).
- 4) With the left dropdown box of the code window displaying cmdFind, the right dropdown box contains a list of the events associated with the CommandButton control, cmdFind. Select Click, the Click event of the CommandButton control, cmdFind (Figure 8.27).
- 5) Visual Basic editor automatically generates the heading Private Sub cmdFind_Click() and the ending End Sub of the event procedure for you. Type in the above code (Figure 8.27) between the heading and the ending of the event procedure.

Figure 8.27 *cmdFind_Click event procedure*

```
Lesson8_form - frmIndex (Code)
cmdFind Click
Private Sub cmdFind_Click()
    Dim doc, para, strInput, binFound, range2
    Dim strMsg, intBtn, intAnswer, idxTable
    ' Define buttons.
    intBtn = vbYesNo + vbInformation + vbDefaultButton2
    strInput = Trim(txtText.Text)
    Set doc = ActiveDocument
    For Each para In doc.Paragraphs
        para.Range.Select
        With Selection.Find
            .Forward = True
            .ClearFormatting
            .MatchWholeWord = True
            .MatchCase = False
            binFound = .Execute(FindText:=strInput)
        End With
    Next para
End Sub
```

```
Lesson8_form - frmIndex (Code)
cmdFind Click
        .MatchCase = False
        binFound = .Execute(FindText:=strInput)
    End With
    If binFound Then
        strMsg = "" & strInput & "" & " found, " & _
            "mark as index entry?"
        intAnswer = MsgBox(strMsg, intBtn)
        If intAnswer = vbYes Then
            ' User chose Yes.
            doc.Indexes.MarkEntry Range:=Selection.Range, _
                Entry:=Selection.Range.Text
            strMsg = "Update index table?"
            intAnswer = MsgBox(strMsg, intBtn)
            If intAnswer = vbYes Then
                If doc.Indexes.Count = 0 Then
                    Set range2 = doc.Content
                    range2.Collapse Direction:=wdCollapseEnd
                    doc.Indexes.Add Range:=range2, Type:=wdIndexRunin
                End If
                For Each idxTable In doc.Indexes
                    idxTable.Update
                Next
            End If
        End If
        strMsg = "Search more?"
        intAnswer = MsgBox(strMsg, intBtn)
        If intAnswer = vbNo Then Exit Sub
    End If
Next
Exit Sub
End Sub
```

Click Event of CommandButton Control, cmdExit

Next you'll code an event procedure to unload the UserForm form, frmIndex from the computer memory when you click the Exit button on the UserForm form. Your clicking of the Exit button (the CommandButton control) on the UserForm form, frmIndex triggers the Click event of the CommandButton control and the code you write in the event procedure will be called upon to perform the operation.

The following code uses the Unload statement (a Visual Basic statement, not a method or property of an object) to remove the current active UserForm form from the computer memory. Me is a special object that refers to the currently active UserForm form, which is frmIndex.

Completed Code

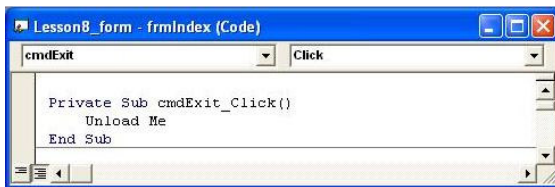
```
Private Sub cmdCancel_Click()
    Unload Me
End Sub
```

Follow these steps to code the event procedure to handle the Click event of the CommandButton control, cmdExit.

Steps

- 1) Make sure the frmIndex (UserForm) window is visible, if not select the folder, frmIndex in the Project Explorer.
- 2) Display or make visible the frmIndex (Code) window by double clicking anywhere in the frmIndex (UserForm) window, or right clicking the form window and select View Code.
- 3) The code window has two dropdown boxes. The left dropdown box contains a list of all the controls (Label1, txtText, cmdFind, cmdExit) and UserForm. Select cmdExit (Figure 8.28).
- 4) With the left dropdown box of the code window displaying cmdExit, the right dropdown box contains a list of the events associated with the CommandButton control, cmdExit. Select Click, the Click event of the CommandButton control, cmdExit (Figure 8.28).
- 5) Visual Basic editor automatically generates the heading Private Sub cmdExit_Click() and the ending End Sub of the event procedure for you. Type in the above code (Figure 8.28) between the heading and the ending of the event procedure.

Figure 8.28 *cmdExit_Click event procedure*



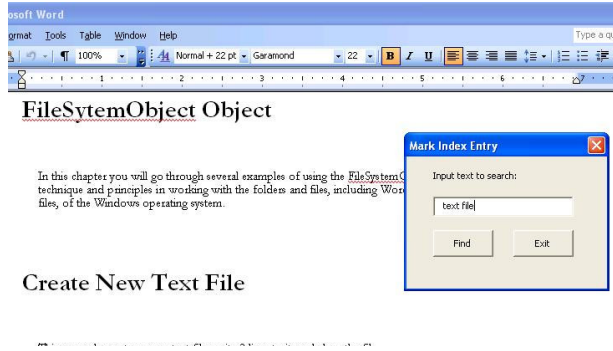
Run Macro

Now you're ready to see your macro in action. To run the addIndex macro follows these steps.

Steps

- 1) From the Word menu select Tools / Macro /Macros.
- 2) When the Macro window appears selects addIndex in the Macro Name pane and click the Run button.
- 3) The Mark Index Entry dialog box appears (Figure 8.29).
- 4) Type in some words. In Figure 8.29 the text string (text file) is typed into the Textbox control, txtText.
- 5) Click the Find button.

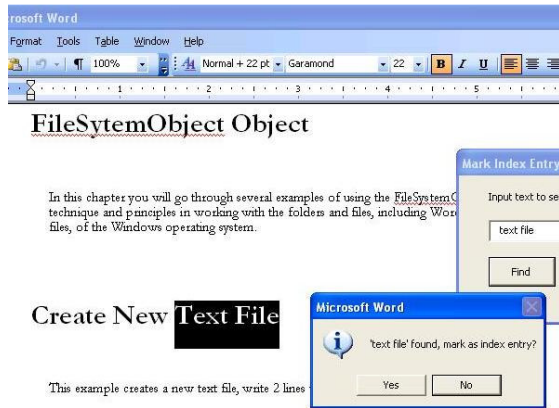
Figure 8.29 Mark Index Entry dialog box



Steps

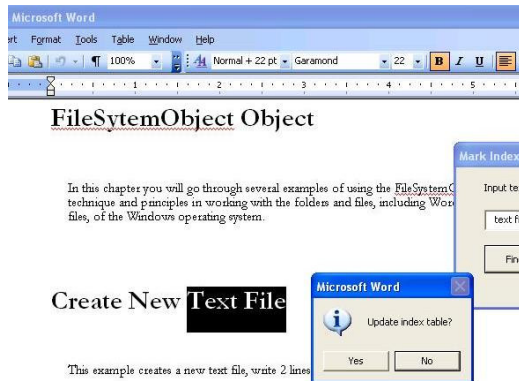
- 1) The MsgBox window appears saying the words “text file” are found and asking you if you want to mark it as an index entry (Figure 8.30).
- 2) The words Text File is selected (Figure 8.30).
- 3) Click the Yes button in the MsgBox window.

Figure 8.30 “text file” found



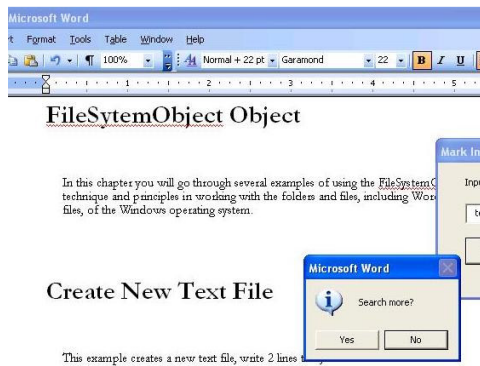
Steps

- 1) The MsgBox window appears asking if you want to update the index table (Figure 8.31).
- 2) Click the Yes button in the MsgBox window.

Figure 8.31 *Update index table?*

Steps

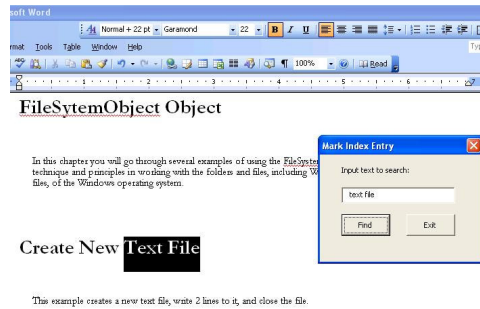
- 1) The MsgBox window appears asking if you want to continue searching (Figure 8.32).
- 2) Click the No button in the MsgBox window.

Figure 8.32 *Search more?*

Steps

- 1) Focus is returned to the Mark Index Entry dialog box (Figure 8.33).
- 2) Click the Exit button to exit the addIndex macro.

Figure 8.33 *Exit addIndex Macro*



Steps


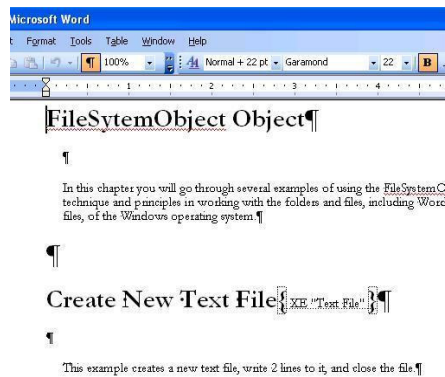
In the document click the Show/Hide toolbar  and see the {XE "Text File"} tag for the index entry (Figure 8.34).

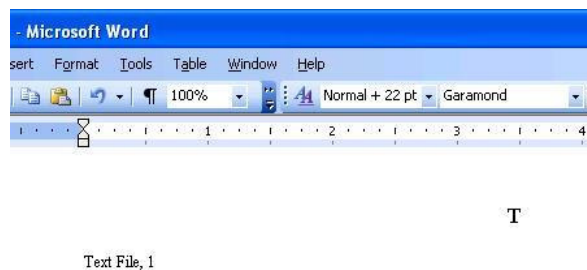
Figure 8.34 {XE "Text File"} index entry



Steps

Scroll to the end of the document to see the index table added with "text File" as the index entry (Figure 8.35).

Figure 8.35 *Index table Added*



This concludes Chapter 8.

Index

- ActiveDcoument object, 44
- Arithmetic Operator, 55
- Array, 83
- assign Method object reference, 25
- assign Property object reference, 25
- BeignTrans, 100
- Boolean data type, 56
- calling procedure, 82
- caption property, UserForm, 132
- case sensitive, variable name, 53
- change Caption property of UserForm form, 132
- change name of UserForm form, 132
- class, 15
- Click Event, CommandButton Object, 140
- collection object, 18
- CommitTrans, 100
- Comparison operator, 57
- concatenate operator &, 62
- Control, 130
- Controls in Toolbox, 132
- create Access database, 95
- CreateFolder method, 123
- CreateTextFile method, 123
- Database Object, 99
- date data type, 64
- Debugging, 90
- Dialog Box, 129
- Do...Loop statement, 80
- Double data type, 54
- dynaset-type Recordset object, 100
- enum, 43
- Error Handling, 90
- event handler, 41
- event procedure, 41
- Events, 15
- Execute method, Find object, 124
- Fields collection object, 105
- Fields("name"), 105
- Files property, 120
- FileSystemObject Object, 117
- Find property, Selection object, 148
- Folder Object, 119
- FolderExists method, 123
- For Each...Next Statement, 75
- For...Next Statement, 73
- form, 129
- function, 31
- Function procedure, 8
- Function Procedure, 82
- GetFolder method, 119, 124
- Icons Used in Object Browser, 37
- If...Then...Else (Block Form) Statement, 68
- If...Then...Else Statement (Single-Line Form), 66
- Initialize Event, UserForm Object, 138
- instance of class, 15
- Integer data type, 54
- Interaction class, 117
- logical operators, 59
- Long data type, 54
- macro, 5
- macro name, 5
- MailMerge Object, 110
- Me, special object, 142
- methods, 15
- module, 5
- MoveFirst method, 102
- MoveLast method, 101
- object, 15
- object hierarchy, 15
- object reference, 21
- Open method, Documents collection object, 124
- OpenDatabase Method, 100

OpenRecordset Method, 101
Option Explicit Statement, 51
primary key, 97
Private procedure, 8
procedure, 7
properties, 15
Property procedure, 8
Public procedure, 8
Rollback, 100
SaveAs method, Document object, 124
Select method, 85
Selection object, 85, 102
Show Method, UserForm Object, 137
Single data type, 54
SQL, 101
Step Into, 94
Step Out, 94
Step Over, 94
string data type, 61
Sub procedure, 8, 31
SubFolders property, 120
syntax, square brackets [], 66
TextStream Object, 118
Unload statement, 142
UserForm Object, 130
variable, 53
Variables Naming Convention, 54
Visual Basic Editor, 2
Visual Basic Naming Rules, 53
With Statement, 81
Words collection object, 87
Workspace Object, 100
WriteLine method, 118, 124