OFFICE AUTOMATION with VBA (Office 97/2000)

CONTENTS

1. INTRODUCTION

This document is intended as an introductory guide to the development of customised applications using Microsoft Office 97 or Office 2000. We concentrate on using the Excel application, although the general skills gained are equally applicable to the other Office applications. It is not an exhaustive review of all of the features of these two packages - there are too many! I hope, however, to be able to describe the essential features of programming in Visual Basic for Applications (VBA) and to illustrate some of its capabilities by means of a range of illustrative examples.

The 'macro' language VBA is a variant of the popular Visual Basic programming language. It offers the programmer the facility to automate and enhance the Office application and to develop a customised application for an end user who may not have the interest or desire to do so for themselves. These techniques can also be used to develop applications which streamline the use of the Office applications for a more expert user - they may be able to apply some level of customisation themselves.

In Office95 the macro language for Word was WordBasic, but this has now been replaced by VBA. VBA can also be used to control Access, Powerpoint and Outlook, as well as an increasing range of third-party applications (i.e. not made by Microsoft), and therefore offers a unified, consistent programming interface.

One of the first questions to ask is - why do you need to program MS Office? Why do you need to do more than use the built-in functions? These are some of the answers:

- Provision of a customised interface - toolbars/menus/controls/dialog sheets
- Provision of specialised functions
- The execution of complex sequences of commands and actions
- Complex data handling procedures
- Interaction with and use of other applications

If all this needs to be added on to Excel and Word to make them useful, why not use Visual Basic itself? Clearly there are pros and cons to using VBA as a development platform, and it is important to be informed of these:

Pros:
- Through VBA, Excel and Word provide a wide range of in-built functions
- It provides a convenient data display tool - the Excel worksheet
- It has powerful in-built charting facility.
- It provides simple access to the other built-in features, e.g. spell-checking
- Distribution is easier - so long as the target user has a copy of Office

Cons:
- There are a more limited set of graphical controls available than in VB
- In many cases data have to be written to cells (in Excel) before they can be used in charts and some functions.
- There is a certain overhead in running Office apps - these are not lightweight apps!

This material will provide you with the information necessary to develop customised applications that can carry out complex tasks with a high degree of automation. This might be as simple as an added worksheet function that Excel does not provide, or it might involve developing dialog boxes, coding and interacting with other applications and Windows itself. It will be assumed that you are familiar with the BASIC programming language, although if you are rusty you'll soon pick it up again!

To make best use of this documentation you should use it in conjunction with the sample programs provided via the unit's web pages. You can access these either by following the links from the Maths main page:

http://www.shu.ac.uk/maths/

or by going directly to

http://maths.sci.shu.ac.uk/units/ioa/

The best way to learn the language (as I'm sure you all know!) however, is to have a particular task or problem to solve. You can then develop skills in VBA as necessary.

Before considering coding - i.e. writing VBA programs - you should first try recording a sequence of actions using the macro recorder. Examination of the code this creates will give some insight into the structure of the language, but you should note that it will not generate *efficient* code!

How to Use These Notes

This booklet of notes is intended to be used as a reference source - you will probably find it rather DRY reading on its own! You should work through the **tutorial exercises** (at your own pace) referring to the notes in this book, and the on-line help, as necessary. The exercises in the tutorials are intended to help you develop an understanding of the various aspects of the Office programming environment and the methods of coding. You will find all relevant material available for download from the web pages for this unit, as given above.

Recommended Reading:

The following books may be worth a look:

MS Office97 Visual Basic Programmers' Guide, MSPress, 1997. £32.49
MS Office97 Visual Basic Step by Step, MSPress, 1997. £32.99
Excel97 Visual Basic Step by Step, Reed Jacobson, MSPress, 1997. £22.99
Word97 Visual Basic Step by Step, ??, MS Press, 1997. £32.99

Web Addresses:

http://maths.sci.shu.ac.uk/units/ioa/	Home page for this unit
http://msdn.microsoft.com/officedev	The Microsoft page!
http://eu.microsoft.com/office/excel/support.htm	Excel support
http://mspress.microsoft.com/	The Microsoft Press web site
http://www.shu.ac.uk/maths/	The SHUMaths home page

1.1 Macro Basics

To illustrate the essential methods of macro generation, we will start by using Excel. The techniques are equally applicable to the other Office applications.

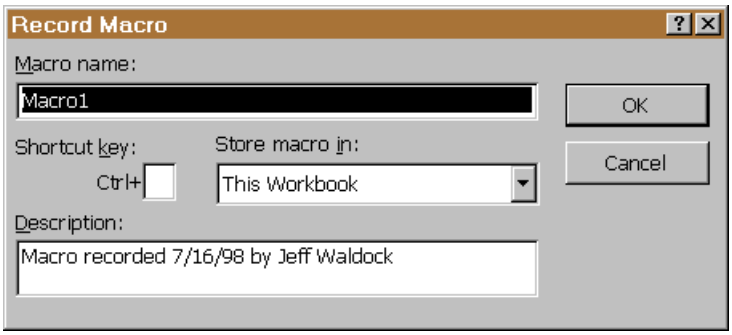
The version of VBA included with Excel is a major overhaul of the version 5 software (included with office 95). Whereas in the previous version it was accessed via a module sheet added to the workbook the VBA 'macro' editor is now a complete development environment.

When you start to write VBA macros, you really need to learn two skills - how to work with VBA and how to deal with the 'host' application, such as Word or Excel. The more you know about Words as a word processor and Excel as a spreadsheet the more effective you can be at developing macros to control them.

Creating a Simple Macro

At the most basic level, automation of Excel could involve recording a sequence of actions using the macro recorder and assigning the resulting macro to a keystroke combination, a toolbar button, a menu item or a control object. To record a macro, carry out the following:

- Select "Tools", "Macro", "Record New Macro" from the menu bar. Alternatively, display the Visual Basic toolbar (right click on the toolbar and select "Visual Basic") A dialog box will appear containing the default name of the macro (macro1).



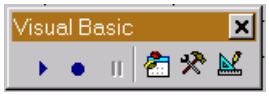
- By default macros you create will not be assigned to a keystroke combination, menu, toolbar or control object - you need to do this manually. You can at this choose to add the new macro as a shortcut key if you wish.
- Call the macro "BoldItalic" and select "OK". A new macro sheet called "BoldItalic" will be generated and a small floating toolbar with the "stop macro" button appears.
- Carry out the sequence of actions you want to record. For example, you might want to select the range of cells "A1:C10", and make the text bold and centred, by clicking on the appropriate buttons on the toolbar.
- Click on the "stop macro" button. The floating toolbar disappears, and the macro recording stops.
- You can test the macro out by selecting the cell range as before, deselecting bold and italic, entering some text into one or more cells in the range, and running the macro (from the menu).

You might well wonder, however, where the code went to! Under Excel 5 and 7 the macro recorder would have created a new module sheet and put the code in there. Under Excel97/2000, you have to go looking for it! Recorded macro code is now

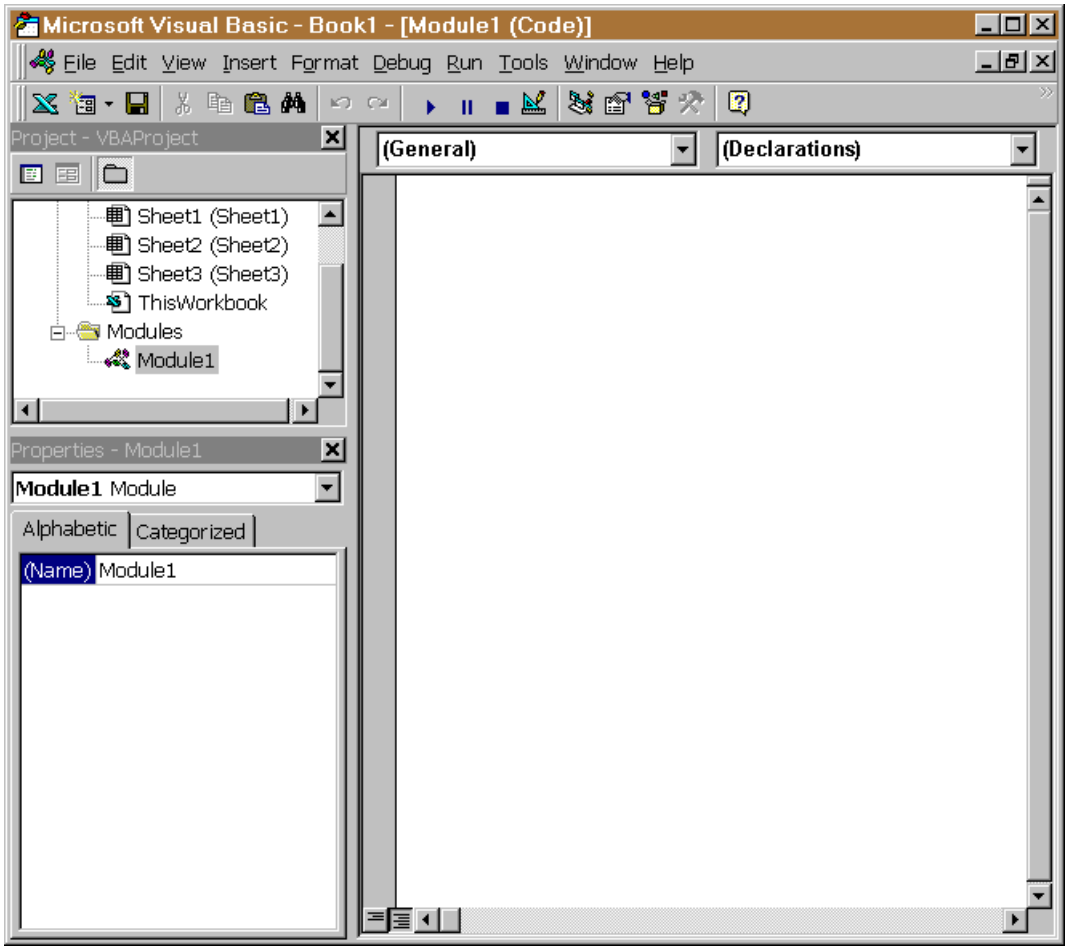
placed in a **Module** which may be accessed from the VBProject explorer. You can also manually add a module and edit it, or edit a pre-existing module.

Code may also be added to **Event procedures** which will be found either within the Worksheet, or on a **UserForm** - more about those later.

If you have not displayed the Visual Basic toolbar, do so now:



Select the fourth icon - this activates the Visual Basic editor.



This editing environment will seem to be rather complex at first, even if you are used to programming previous versions of Excel. We will look at the various parts of this in more detail later, but you will note that in the VBAProject window (top left) are listed the components of your workbook. These objects include the workbook itself, the active sheet, and the modules. The module is where recorded macros will be placed. If you want to make more room to view macros in the editing window, and do not need to see the Project or Properties window, close those down and maximise the Module window.

Double-click on "Module 1" to view the code recorded by our BoldItalic macro:

```
Sub BoldItalic()
'
' BoldItalic Macro
' Macro recorded 29/06/98 by Jeff Waldock
    Range("A1:C10").Select
    Selection.Font.Bold = True
    Selection.Font.Italic = True
    Range("B4").Select
End Sub
```

This is the Visual Basic for Applications (VBA) code generated by the macro recorder. Note that

The procedure begins with "`Sub` *`modulename`*`()`" and end in "`End Sub`".

The lines beginning with an apostrophe (') are comments.

The range of cells required is specified by means of the Range object, to which the Select method has been applied. This follows the general rule of **Object.Action**.

The Selection is made bold and italic by applying the relevant value to the property of the selected object. This follows the general rule of **Object.Property = Value**.

If you now wish to assign this macro to a keystroke combination, do the following:

- Choose "Tools", "Macro", "Macros" from the menu.
- Select your macro from the list provided and click "options"
- You then have the option of assigning the macro to a keystroke combination.

The first time you record a macro it is placed in a new module. Each time you record an additional macro, it is added to the end of the same module, but if you do so after closing and re-opening the workbook, any macros recorded are placed in a new module. There is no way for you to control where the macro recorder places a new macro. You can, of course, edit, replace and the macros afterwards.

Even if you do not want to edit the macros, the location should not be a problem. When you use the macro dialog box to select and edit a macro, it will automatically take you to the correct module.

2. OFFICE OBJECTS

In order to understand the structure of VBA as applied to Office 97/2000, it is necessary to understand the ***object model***. Objects are the fundamental building blocks of the Office applications - nearly everything you do in VBA involves manipulating objects. Every unit of content and functionality in the Office suite - each workbook, worksheet, document, range of text, PowerPoint slide and so on - is an object that you can control programmatically in VBA. When you understand the Office object model you are ready to automate tasks! As stated above, all objects in Office can be programmed or controlled; however there are hundreds altogether, ranging from simple objects such as rectangles and boxes to pivot-tables and charts. It is not necessary to learn them all (!) - you just need to know how to use them, and where to look for information about them - all are fully documented in the on-line VBA help file.

Objects, Properties and Methods

As before we will use Excel to demonstrate the use of Office objects - objects in Word will be dealt with in just the same way.

All objects have ***properties*** and ***methods***. VBA allows the control of Excel objects through their properties and methods. In general, you use properties to get to content and methods to get to functionality. The Excel Workbook object, for example, has a number of properties, including:

Author	the name of the person who created the workbook
Name	the name of the workbook
Path	the full disk path where the workbook is saved
HasPassword	true or false

Note that the property values may be numerical, string or Boolean, and that they may be specific or apply to several objects. Properties may be obtained or set by appropriate assignment statements in VBA. When doing so, objects and properties are referred to by using a dot to separate them:

```
Workbooks("Book1.XLS").Author="Donald Duck"
```

This code may be incorporated into a VBA macro as follows:

```
Sub SetAuthorName()
      Workbooks("Book1.XLS").Author="Donald Duck"
End Sub
```

Getting property settings works in much the same way:

```
AuthorName=Workbooks("Book1.XLS").Author
```

In addition to properties, objects have ***methods*** - these are actions that can be performed on or by them. Examples of workbook methods are:

Activate	Activates the first window associated with the workbook
Close	Closes the workbook
PrintPreview	Displays workbook in printpreview mode
Save	Saves the workbook
SendMail	Send the workbook as embedded object in an e-mail message

Methods may be called by simply referring to the object and method. Methods may additionally allow or require extra information to be supplied as arguments. The **Close** method of the **Workbook** object has three arguments: **SaveChanges** (true or false), **FileName** and **RouteWorkbook** (true or false). There are several ways to carry out this call:

(1) `Workbooks("Book1.XLS").Close`

In this case the arguments take on default values.

(2) `Workbooks("Book1.XLS").Close True, "Thisbook.XLS", False`

Here the arguments are given - they must be in the correct order.

(3)
```
Workbooks("Book1.XLS").Close saveChanges:=True, fileName:="X.XLS", _
routeWorkbook:=False
```

Here the order does not matter. Note the line continuation character. In practice this is not often necessary since Excel will allow 1024 characters on one line.

Relation of Object Model to User-Interface

You can interact with an application's objects either directly (using the mouse and/or keyboard) or programmatically. In the first case you would navigate the menu tree to find the feature you want; in VBA you navigate through the object model from the top-level object to the object that contains the content and functionality you want to work with. The properties and methods of the object provide access to the content and functionality. For example, the following Excel example sets the value for cell B3 on the Sales worksheet in the Current.xls workbook:

```
Workbooks("Current.xls").Worksheets("Sales").Range("B3").Value=3
```

Since the user interface and VBA provide these two methods of gaining access to Office 97/2000 features, many objects, properties and methods share names with elements of the user interface, and the overall structure of the object model resembles that of the UI. Consequently, for every action you can take in the UI, there's a VBA code equivalent.

It's important to understand an object's place in the object model, because before you can work with an object you must navigate through the hierarchy of the object model to find it.

Referencing Objects: Singular Objects / Objects in Collections

When using the help facility to locate objects in VBA you may notice that there are often two object types offered - usually the singular and plural versions of the same name, such as "workbooks" and "workbook". The first of these is a *collection* object. All VBA objects fall into one of two classes - singular objects, and objects in a collection. Singular objects are referenced by name; objects in a collection are referenced by an index in the collection. To help remember which you are dealing with there are two simple rules:

1. A singular object has only one instance in a given context - i.e. it’s unique.
2. An object in a collection has multiple instances in a given context.

For example, Excel has a singular object named **Application** (Excel itself). The **Font** object is singular because for any given cell there is only one instance of the **Font** object. It does, however, have multiple properties (**Name**, **Bold**, **Italic**, etc). The **Worksheet** object however, is an object in the **Worksheets** collection - many

Worksheets can exist on one **Workbook** file. Similarly the **Chart** object is an object in a collection.

Singular objects are referenced directly:

```
Application.Caption="My leg is broken"
Application.ScreenUpdating=False
```

Objects in collections are referenced by using an index (list number or name):

```
Worksheets(1).Name="Custard Pie"
Charts("Chart1").HasLegend=True
Worksheets(1).Visible=False
Worksheets("Custard Pie").Visible=False
```

Note that both of the last two statements above would have the same effect. The name reference is case-insensitive. You can also create new objects and add them to a collection, usually by using the **Add** method of that collection. To add a new worksheet to a workbook, for example, you would use:

```
Worksheets.Add
```

You can find out how many items there are in a collection using the **Count** property:

```
Msgbox "There are " + Worksheets.Count + "in this workbook"
```

Collections are useful in other ways as well - you can, for example, perform an operation on all objects in a collection, using a `For Each .. Next` loop.

The Range Object - an exception

The **Range** object falls into a grey area between singular objects and objects in collections, and exhibits some characteristics of both.

To set the contents of a particular cell or range of cells, use the **Value** property of the **Range** object. e.g.

```
Range("A1").Value=1
Range("A1:F20").Value=1
```

Worksheet ranges can also have names:

```
Range("A1").Name="Spud"
Range("Spud").Value=1
```

Using the Excel Object Hierarchy

To manipulate the properties and methods of an object, it is sometimes necessary to reference all objects that lie on the hierarchical path to that object. Suppose, for example, that we want to set the **Value** property of a **Range** object representing the first cell in the first **Worksheet** of the first **Workbook** in Excel. Using the full hierarchical path we would have:

```
Application.Workbooks(1).Worksheets(1).Range("A1").Value=1
```

This is not always necessary - how far up the hierarchy you need to start depends upon the context in which the statement is made. The above statement could be made anywhere in Excel under any circumstances. The first object, **Application**, can be dropped since Excel understands that it is not necessary to reference itself!

The current workbook can be referenced implicitly using the fact that only one can be active at one time (this idea applies to other objects in collections):

```
ActiveWorkbook.Worksheets(1).Range("A1").Value=1
```

`ThisWorkbook` may also be used - this will refer to the workbook containing the macro code. If no ambiguity occurs, it is possible to remove successive levels of the hierarchy; the minimal statement would then be:

```
Range("A1")=1
```

(**Value** is the default property of the **Range** object.)

The active range can also be accessed via the **Application** objects **Selection** property. There are several objects that have a **Selection** property - during execution, VBA determines what type of object has been selected and evaluates **Selection** to the appropriate object. When using **Selection**, you cannot use default properties, so the minimal statement is:

```
Selection.Value=1
```

Executing this code assigns 1 to the Value property of the selected range to 1, whether it's `Range("A1")` or `Range("A1:Z256")`.

Sometimes it is inconvenient to refer to cells using the "An" format - we would rather use the numerical values of the rows and columns. This can be done by using the **Cells** property of the **Application** object, as follows:

```
Cells(row,col).Value=1
```

where `row` and `col` are integer values referring to the row and column required.

Getting Help Writing Code

Although you will often know or guess the correct object names and syntax, but for those occasions when this is not the case, Office application include a number of tools to help.

Using the Macro Recorder

If you know how to carry out the task you want to perform by using the user-interface, you can do this with the macro recorder turned on. You will probably want to subsequently edit the code produced by the macro recorder to make it more efficient and robust. But it is a very useful tool for getting the basic structure of your code in place. To get help on a particular keyword, place the insertion point on or next to the word you want help for, and press F1. The help file will automatically load the correct topic, which will give a full explanation of the keyword or function together with examples of its use.

Using the Object Browser

Each Office application contains an *object library* giving information about all of the objects that application contains. The Object Browser lets you browse this information, and may be accessed via the View menu in the VBA editor. You should take the time to have a 'play' with this since it provides a quick way of finding what methods an object supports, for example, and how it may be used in VBA.

Statement-Building Tools

There are a number of built-in tools to help build expressions and statements in VBA. To turn these features on or off in the VBA editor, select one or more of the following check boxes under "Code Settings" on the "Editor" tab in the "Options" dialog box ("Tools" menu).

Option	Effect
Auto Syntax Check	Determines whether VB should automatically verify correct syntax after you enter a line of code
Require Variable Declaration	Determines whether explicit variable declarations are required in modules. Selecting this check box adds the statement "Option Explicit" to general declarations in any new module.
Auto List Member	Displays a list that contains information that would logically complete the statement at the current insertion point location.
Auto Quick Info	Displays information about functions and their parameters as you type.
Auto Data Tips	Displays the value of the variable that the pointer is positioned over. Available only in break mode.
Auto Indent	Repeats the indent of the preceding line when you press ENTER - all subsequent lines will start at that indent. You can press BACKSPACE to remove automatic indents.
Tab Width	Sets the tab width, which can range from 1 to 32 spaces (the default is 4 spaces)

These tools automatically display information and give appropriate options to choose from at each stage of building an expression or statement. For example, with the "Auto List Member" option selected, type the keyword "Application" followed by the dot operator. You should see a box appear listing the properties and methods that apply to the Application object. You can select an item from the list, or just keep typing. Try these out!

3. Visual Basic for Applications (VBA)

This version of BASIC has many of the standard BASIC constructs, as you would expect, but there are a number of additional structures, and some different syntax, for this particular application. As a programmer with some experience of BASIC you will need to familiarise yourself with the syntax (some of which has been seen already), learn about variable scoping and note the new control structures (the **For Each ... Next** structure and **With .. End With**).

Tips for Learning VBA

Before learning VBA it is important to have a good grasp of the relevant Office applications - the more you know about the operation of the application and its capabilities, the better prepared you will be for using VBA.

Learn what you need, when you need it. There is an almost overwhelming volume of material - develop skills in the use of a small fraction of this first and learn other parts as and when necessary.

Use the macro recorder. Office applications provide the facility for recording a sequence of actions - you can then look at the VBA code produced and adapt or copy it for other purposes. Sometimes it is quicker to record a macro than to type it from scratch anyway!

Use Visual Basic help. Press F1 with the insertion point in any keyword takes you straight to that part of the help file.

Variables, Constants and Data Types

The following table lists the data types that VBA supports:

Data Type	Description	Range
Byte	1-byte binary data	0 to 255
Integer	2-byte integer	-32,768 to 32,767
Long	4-byte integer	-2,147,483,648 to 2,147,483,647
Single	4-byte floating point number	-3.4E38 to -1.4E-45 (-ve values) 1.4E-45 to 3.4E38 (+ve values)
Double	8-byte floating point number	-1.79E308 to -4.9E-324 4.94E-324 to 1.79E308
Currency	8-byte number with fixed decimal point	-922,337,203,685,477.5808 to +922,337,203,685,477.5807
String	String of characters	0 to approx 2 billion characters
Variant	Date/time, floating-point number, integer, string or object. 16 bytes, plus 1 byte for each character if the value is a string.	Jan 1 100 to December 31 9999, Numeric - same as double String: same as string Also can contain **Null**, **False**
Boolean	2 bytes	**True** or **False**
Date	8-byte date/time value	Jan 1 100 to December 31 9999
Object	4 bytes	Any object reference
User-defined	dependent on definition	

Setting variables

```
Dim a as integer, b as double
Dim r as Object
Set r=worksheets(1).Range("A1:B10")
r.Value=1
```

In the last example it is more efficient to declare r as a specific object:

```
Dim r as Range
```

By combining object variables with the new VBA **With..Endwith** statement it is possible to produce much more compact code:

```
Worksheets(1).Range("A1:A10").Value=24
Worksheets(1).Range("A1:A10").RowHeight=50
Worksheets(1).Range("A1:A10").Font.Bold=True
Worksheets(1).Range("A1:A10").Font.Name=Arial
Worksheets(1).Range("A1:A10").HorizontalAlignment = xlCenter
Worksheets(1).Range("A1:A10").VerticalAlignment = xlBottom
```

This could be re-written as:

```
Set r=Worksheets(1).Range("A1:A10")
With r
      .Value=24
      .RowHeight=50
      With .Font
            .Bold=true
            .Name=Arial
      End With
      .HorizontalAlignment=xlCenter
      .VerticalAlignment=xlBottom
End With
```

Variable declaration

It is normally recommended that variables are always explicitly declared. This fulfils two purposes

1. it ensures that spelling mistakes are picked up (undeclared variables will be flagged) and
2. since undeclared variables are given the **Variant** type, taking up a minimum of 16 bytes, it ensures that memory is efficiently used.

By placing the declaration `Option Explicit` at the top of your module, you can ensure that Excel requires the explicit declaration of all variables. Alternatively, you can change the default data type from **Variant** to, say, integer by means of the statement `DefInt A-Z` which must also be placed before any subroutines.

Array declaration

Arrays are declared in a similar way to variables. You use the **Private**, **Public**, **Dim** or **Static** keywords and use integer values to specify the upper and lower bounds for the array. You use the **As** keyword to declare the array type. For example:

```
      Dim counters(15) as Integer
      Dim sums(20) as Double
```

User-defined Data Types

You can create your own data types..

```
Type StudentData
      sName as String
      sAge as Integer
      sBorn as Date
End Type
Sub xxx()
```

```
    Dim Student1 as StudentData
    Student1.sName="Teresa Green"
    Student1.sAge=99
    Student1.sBorn=#31/12/1896#
    Msgbox Student1.sName & ", Age " & Student1.sAge & _
    ", Born " & Student1.sBorn & "."
End Sub
```

Note that the Msgbox statement should all on one line - the underscore character has been used as a continuation character.

Variable Scope

This refers to the area in the VBA application in which a variable can be accessed. There are three levels of scope :

(1) procedure level : declarations made within a sub-program

(2) module level : declarations made at the start of a module and

(3) project level : as module level, but using the `Public` keyword

At the procedure level, variables can also be declared using the keyword `Static`, which means that the value of the variable is preserved between calls to the procedure.

Setting an Object Variable

You declare an object variable by specifying for the data type either the generic **Object** type or a specific class name from a referenced class library. For example:

```
Dim mySheet as Object
```

For many reasons it is much better to declare a specific object type, so that VBA can carry out necessary checks to ensure that the object exists etc. For example:

```
Dim mySheet as WorkSheet
Dim myRange as Range
```

In addition to specifying a class name you may need to qualify the object variable type with the name of the application hosting the object:

```
Dim wndXL as Excel.Window
Dim wndWD as Word.Window
Dim appWD as Word.Application
```

To assign an object to an object variable, use the Set statement:

```
Dim myRange as Excel.Range
Set myRange = Worksheets("Sheet1").Range("A1")
```

If you do not use the Set statement, VBA will not realise you are trying to set an object reference, and think instead that you are trying to assign the value of the default property to the variable. For instance:

```
myRange = WorkSheets("Sheet1").Range("A1")
```

will result in the contents of the cell A1 being stored in the variable myRange.

3.1 Control Structures

Execution flow in VBA can be controlled by a number of in-built structures. If left to its own devices, VBA will process the lines of code in sequential order. Two sets of common structures are included with all languages to help the programmer modify the flow of logic: **decision structures** and **loop structures**.

Decision Structures

```
If .. Then
If .. Then .. Else
If .. Then .. ElseIf .. EndIf
Select Case .. End Select
```

Loop Structures

```
While .. Wend
Do While .. Loop
Do Until .. Loop
Do .. Loop Until
Do .. Loop While
For .. Next
For Each .. Next
```

Each of these is probably already familiar to you, with the possible exception of the last one. The **For Each ... Next** structure is very powerful and allows you to loop through all of the objects in a collection, or all the elements in an array. e.g.

```
Option Base 1
Sub xxx()
      Dim StudentName(3) as String
      Dim Student as Variant
      StudentName(1)="John Smith"
      StudentName(2)="Paul Smith"
      StudentName(3)="Anne Smith"
      For Each Student in StudentName
            Msgbox Student
      Next
End Sub
```

One of the benefits of this structure is that you don't need to know in advance how many elements have been filled in the array. Its real power, however, is in the manipulation of collections of objects:

```
Sub xxx()
      Dim SheetVar as Worksheet
      For Each SheetVar in ActiveWorkbook.Worksheets
            Msgbox SheetVar.Name
      Next
End Sub
Sub xxy()
      Dim x as Integer
      Dim Book as Workbook
      For x=1 to 10
            Workbooks.Add
      Next
      Windows.Arrange
      Msgbox "Workbooks have been arranged"
      For Each Book in Application.Workbooks
            If Book.Name<>ThisWorkbook.Name then Book.Close
      Next
```

```
      ActiveWindow.WindowState=xlMaximised
End Sub
Sub xxz()
      Dim Cell as Range
      Dim R as Range
      Set R=Range("A1:F20")
      For Each Cell in R
            Cell.Value=25
      Next
End Sub
```

Formulas & Functions

The Excel spreadsheet consists of two layers - the value layer and the function layer. The value layer is active by default, so that the results of any formulae entered in cells are displayed. When the formula layer is active the formulas are displayed. You can select this by choosing "Tools", "Options", "View", "Formulas". Alternatively, you can toggle between the two by using [Ctrl ~].

In Excel you enter a formula by using the = sign, followed by an expression which may include a reference to other cells. This reference is by default in A1 notation. For example, a formula which inserts the value in cell B3 multiplied by 10 would be =B3*10.

To set Excel to accept formulas in R1C1 notation, choose "Tools", "Options" and "R1C1". This formula is then =R3C2*10.

This notation allows the use of relative referencing by using square brackets. For example, the following formula takes the value of a cell three rows down and one to the right and divides it by 5:

=R[3]C[1]/5

Entering Formulas and Functions in VBA

```
Range("B12").Formula="=AVERAGE("B1:B11")
```

Creating your own worksheet functions using VBA

You can call a VBA function directly from a formula in a cell. For example, you can use the factorial function below as follows:`=Factorial(5)`

```
Function Factorial(Intl as Variant)
      Dim x as Integer
      Factorial=1
      If (Not (IsNumeric(Intl)) or Int(Intl)<>Intl or Intl<0) then
            Msgbox "Only non-negative Integers allowed"
            Factorial="#NUM!"
      Else
            For x=1 to Intl
                  Factorial=Factorial*x
            Next
      End If
End Function
```

4. EXCEL OBJECTS AND COLLECTIONS

VBA supports a set of objects that correspond directly to elements in EXCEL. For example, the **Workbook** object represents a workbook, the **Worksheet** object represents a worksheet and a **Range** object represents a range of cells. To perform a task in VBA you return an object that represents the appropriate Excel element and then manipulate it using that object's properties and methods. For example, to set the value of a cell using the **Value** property of the **Range** object:

```
Worksheets("Sheet1").Range("A1").Value=3
```

A **collection** is an object that contains a group of related objects. The **Worksheets** collection object contains **Worksheet** objects, for example. Each object within a collection is called an *element* of that collection. Because collections are objects, they have properties and methods, just as singular objects do. In the above illustration, the **Worksheets** method returns a **Worksheet** object (the method actually returns one member of the **Worksheets** collection). When you want to work with a single object, you usually return one from a collection. The property or method you use to return the object is called an *accessor*. Many accessors take an **index** that selects one object from the collection.

Building an Expression to Return an Object

You can either type the expression from scratch, or use the macro recorder to generate the expression, and modify the result as necessary. For example, suppose you need help building an expression that changes the font style and font size for the title of a chart. The macro recorder might produce the following:

```
Sub macro1()
      ActiveChart.ChartTitle.Select
      With Selection.Font
            .Name="Times New Roman"
            .FontStyle="Bold"
            .Size=24
            .Strikethrough=False
            .Superscript=False
            .Subscript=False
            .OutlineFont=False
            .Shadow=False
            .Underline=xlNone
            .ColorIndex=xlAutomatic
            .Background=xlAutomatic
      End With
End Sub
```

You can now modify this, by

1. changing the **ActiveChart** property to the **Charts** method and use an index for the argument - this will allow you to run the macro from any sheet in the workbook.
2. removing the selection-based code. You also need to move the **With** keyword.
3. removing the properties of the **Font** object that the macro should not change
4. changing the procedure name

After modification, the macro now reads:

```
Sub FormatChartTitle()
      With Charts(1).ChartTitle.Font
            .FontStyle="Bold"
```

```
            .Size=24
      End With
End Sub
```

Applying Properties and Methods to an Object

As seen earlier, you retrieve or change the attributes of an object by getting or setting its properties. Methods perform actions on objects. Many properties have **built-in** constants as their values - for example, you can set the **HorizontalAlignment** property to **xlCenter**, **xlLeft**, **xlRight** and so on. The help topic for any given property contains a list of built-in constants you can use. It is recommended that you should use the built-in constant rather than the value the constant represents because the value may change in future versions of VBA.

Using Properties and Methods which are Unique to Collections

The **Count** property and **Add** method are useful. The **Count** property returns the number of elements in a collection. For example, the following code uses the **Count** property to display the number of worksheets in the active workbook:

```
Sub NumWorksheets()
      MsgBox "No of worksheets in this workbook : " & _
            ActiveWorkbook.Worksheets.Count
End Sub
```

The **Count** property if useful when you want to loop through the elements in a collection, although in most cases the **For Each ... Next** loop is recommended.

The **Add** method creates a new element in a collection, and returns a reference to the new object it creates. This reference may be used in any required action on the new object. For example:

```
Sub CreateScratchWorksheet()
      Set newSheet = Worksheets.Add
      newSheet.Visible=False
      newSheet.Range("F9").Value="some text"
      newSheet.Range("A1:D4").Formula="=RAND()"
      MsgBox newSheet.Range("A1").Value
End Sub
```

Declaring and Assigning Object Variables

Object variables may be declared in the same way as other variables:

```
Dim mySheet as Object
```

This uses the generic **Object** type specifier. This is useful when you don't know the particular type of object the variable will contain. You can also declare an object using a specific class name, e.g.

```
Dim mySheet as Worksheet
```

An object is assigned to an object variable using the **Set** statement. For example, the following code sets the object variable **myRange** to the object that refers to cell A1 on Sheet1:

```
Set myRange=Worksheets("Sheet1").Range("A1")
```

You must use the **Set** statement whenever you want an object variable to refer to an object. If you forget this, several errors may occur. Most will give an informative error, but if you omit the declaration of the object variable *and* forget to use **Set**, then Excel tries to assign the value contained within the object to your "object" variable.

This may succeed, in that it does not give an error, but your macro will clearly not be doing what was intended!

Looping on a Collection

There are several different ways in which you can loop on a collection, however the recommended method is to use a **For Each ... Next** loop, in which VBA automatically sets an object variable to return every object in the collection.

```
Sub CloseWorkbooks
      Dim wb as Workbook
      For Each wb in Application.Workbooks
            If wb.Name <> ThisWorkbook.Name then wb.Close
      Next
End Sub
```

Performing Multiple Actions on an Object

Procedures often need to perform several different actions on the same object. One way is to use several statements:

```
ActiveSheet.Cells(1,1).Formula="=SIN(180)"
ActiveSheet.Cells(1,1).Font.Name="Arial"
ActiveSheet.Cells(1,1).Font.Bold="True"
ActiveSheet.Cells(1,1).Font.Size=8
```

This is easier to read, and more efficient, using the **With ... End With** statement

```
With ActiveSheet.Cells(1,1)
      .Formula="=SIN(180)"
      .Font.Name="Arial"
      .Font.Bold="True"
      .Font.Size=8
End With
```

The object reference can be a reference to a collection. This example sets the font properties of all the text boxes on the active sheet:

```
With ActiveSheet.TextBoxes
      .Font.name="Arial"
      .Font.Size=8
End With
```

Working with the WorkBook Object

When you open or save a file in Excel, you're actually opening or saving a workbook. In VBA the methods for manipulating files are methods of the **Workbook** object or the **WorkBooks** collection.

To open a Workbook, use the **Open** method as follows:

```
Sub OpenBook()
      Set myBook = WorkBooks.Open(Filename:="Book1.XLS")
MsgBox myBook.Worksheets(1).Range("A1").value

End Sub
```

The above procedure opens a file called Book1.XLS in the current working directory and displays the contents of cell A1 on the first worksheet in a message box.

Instead of "hard-coding" the location of Book1.XLS, you may want to give the user the chance to locate the file to open themselves. The **GetOpenFilename** method displays the standard file open dialog box, and returns the full path of the selected file:

```
Sub DemoGetOpenFile()
    Do
         fName = Application.GetOpenFileName
```

```
    Loop Until fName<>False
    MsgBox "Opening " & fName
    Set myBook = WorkBooks.Open(Filename:=fName)
End Sub
```

You create a new workbook by applying the **Add** method to the **WorkBooks** collection. Remember to set the return value of the **Add** method to a variable so you can subsequently refer to the new object. When you first save a workbook, use the **SaveAs** methods; subsequently you can use the **Save** method:

```
Sub CreateAndSave()
    Set newBook = WorkBooks.Add
    Do
        fName = Application.GetSaveAsFileName
    Loop Until fName <> False
    NewBook.SaveAs FileName:=fName
End Sub
```

Working with the Range Object

The **Range** object can represent a single cell, a range of cells, an entire row or column, a selection containing multiple ranges, or a 3-D range. It is unusual in that it can represent both a single cell and multiple cells. There is no separate collection for the **Range** object - it is both a single object and a collection.

Using the Range Method

One of the most common ways to return a **Range** object is to use the **Range** method. The argument to a **Range** method is a string that's either an A1-style reference or the name of a range:

```
Worksheets("Sheet1").Range("A1").Value=3
Range("B1").Formula="=5-10*RAND()"
Range("C1:E3").Value=6
Range("A1","E3").ClearContents
Range("myRange").Font.Bold=True
Set objRange=range("myRange")
```

Using the Cells Method

The **Cells** method is similar to the **Range** method, but takes numeric arguments instead of string arguments. It takes two arguments, the row and column respectively of the cell required.

```
Worksheets("Sheet1").Cells(1,1).Value=3
Cells(1,1).Formula="=5-10*RAND()"
Range("C1:E3").Value=6
```

The Cells method is very useful when you want to refer to cells using loop counters.

Combining the **Range** and **Cells** Methods

In some situations you may need to create a Range object based on top and bottom rows and left and right columns, given as numbers. The following code returns a Range object that refers to cells A1:D10 on Sheet1:

```
Set myObj=Worksheets("Sheet1").Range(Cells(1,1),Cells(10,4))
```

Using the Offset Method

It is sometime necessary to return a range of cells that's a certain number of rows and columns from another range of cells. The `Offset` method takes an input `Range` object, and `RowOffset` and `ColumnOffset` arguments, returning a new range. The following code determines the type of data in each cell of a range:

```
Sub ScanColumn()
    For Each c In Worksheets("Sheet1").Range("A1:A10").Cells
        If Application.IsText(c.Value) Then
            c.Offset(0, 1).Value = "Text"
        ElseIf Application.IsNumber(c.Value) Then
            c.Offset(0, 1).Value = "Number"
        ElseIf Application.IsLogical(c.Value) Then
            c.Offset(0, 1).Value = "Boolean"
        ElseIf Application.IsError(c.Value) Then
            c.Offset(0, 1).Value = "Error"
        ElseIf c.Value = "" Then
            c.Offset(0, 1).Value = "(blank cell)"
        End If
    Next c
End Sub
```

Using the **CurrentRegion** and **UsedRange** Properties

These two properties are very useful when your code operates on ranges whose size is unknown and over which you have no control. The current region is a range of cells bounded by empty rows and empty columns, or by a combination of empty rows/columns and the edges of the worksheet. The used range contains every non-empty cell on the worksheet. The **CurrentRegion** and **UsedRange** properties apply to the **Range** object; there can be many different current regions on a worksheet but only one used range.

```
Set myRange=Worksheets("Sheet1").Range("A1").CurrentRegion
myRange.NumberFormat="0.0"
```

Looping on a Range of Cells

There are several ways of doing this, using either the **For Each … Next** or the **Do .. Loop** statements. The former is the recommended way:

This example loops through the range A1:D10 setting any number whose absolute value is less than 0.01 to zero:

```
Sub RoundtoZero()
      For Each r in Worksheets("Sheet1").Range("A1:D10").Cells
            If Abs(r.Value)<0.01 then r.Value=0
      Next
End Sub
```

Suppose you want to modify this code to loop over a range of cells that the user selects. One way of doing this is to use the **InputBox** method to prompt the user to select a range of cells. The **InputBox** method returns a **Range** object that represents the selection. By using the ***type*** argument and error handling, you can ensure that the user selects a valid range of cells before the input box is dismissed.

```
Sub RoundtoZero()
   Worksheets("Sheet1").Activate
   On Error GoTo PressedCancel
   Set r=Application.InputBox(prompt:="Select a range of cells", Type:=8)
   On Error GoTo 0
   For Each c in r.Cells
      If Abs(c.Value) < 0.01 then c.Value=0
   Next
   Exit Sub
   PressedCancel:
   Resume
```

```
End Sub
```

You could, of course, use the **UsedRange** or the **CurrentRegion** properties if you don't want the user to have to select a range.

5. CODE OPTIMISATION

VBA is very flexible - there are usually several ways to accomplish the same task. When you are writing "one-off" macros you will probably be happy enough to get one that does what is required; when writing one that will be used many times, or will be used by a group of students, you will probably want to ensure that the most efficient coding is used. The following techniques describe ways in which you can make your macros smaller and faster.

Minimising OLE references

Every VBA method or property call requires one or more OLE calls, each of which takes time. Minimise the number of such calls.

Use Object Variables

If you find you are using the same object reference many times, set a variable for the object and use that instead.

Use the With Statement

Use the **With** statement to avoid unnecessary repetition of object references without setting an explicit object variable.

Use a For Each ... Next Loop

Using a **For Each ... Next** loop to iterate through a collection or array is faster than using an indexed loop. In most cases it is also more convenient and makes your macro smaller and easier to debug.

Keeping Properties and Methods Outside Loops

Your code can get variable values faster than it can property values. You should therefore assign a variable to the property of an object outside the loop and use that within a loop, rather than obtaining the property value each time within the loop. For example:

```
For iLoop=2 to 200
      Cells(iLoop,1).Value=Cells(1,1).Value
Next
```

this is inefficient and should be replaced by:

```
cv=Cells(1,1).Value
For iLoop=2 to 200
      Cells(iLoop,1).Value=cv
Next
```

If you're using an object accessor inside a loop, try moving it outside the loop:

```
For c=1 to 1000
      ActiveWorkbook.Sheets(1).Cells(c,1)=c
Next
```

should be replaced by:

```
With ActiveWorkbook.Sheets(1)
      For c=1 to 1000
            .Cells(c,1)=c
      Next
End With
```

Using Arrays to Specify Multiple Objects

Some of the methods that operate on objects in collections allow you to specify an array when you want to operate on a subset of objects in a collection. The following example calls the **Worksheets** method and the **Delete** method three times each

```
Worksheets("Sheet1").Delete
Worksheets("Sheet2").Delete
Worksheets("Sheet4").Delete
```

This could be reduced to one call to each by using an array:

```
Worksheets(Array("Sheet1", "Sheet2", "Sheet4")).Delete
```

Using Collection Index Numbers

Most object accessor methods allow you to specify an individual object in a collection wither by name or by index number. Using the number is much faster than using the name. Set against this, however, is the fact that using the name makes your code easier to read, and will specify the object uniquely (the number could change).

Minimising Object Activation and Selection

Most of the time your code can operate on objects without activating or selecting them. If you use the macro recorder a lot to generate your VBA code, you will be accustomed to activating or selecting an object before doing anything to that object. The macro recorder does this because it must follow your keystrokes as you select and activate sheets and cells. You can, however, write much faster and simpler VBA code that produces the same results without activating or selecting each object before working with it. For example, filling cells C1:C20 on Sheet5 with random numbers (using the **AutoFill** method) produces the following macro recorder output:

```
Sheets("Sheet5").Select
Range("C1").Select
ActiveCell.FormulaR1C1="=RAND()"
Selection.AutoFill Destination:=Range("C1:C20"), Type:=xlFillDefault
Range("C1:C20").Select
```

All of the Select calls are unnecessary. You can replace the above with:

```
With Sheets("Sheet5")
    .Range("C1").FormulaR1C1="=RAND()"
    .Range("C1").AutoFill Destination:=Range("C1:C20"),
       Type:=xlFillDefault
End With
```

Keep in mind that the macro recorder records exactly what you do - it cannot optimise anything on its own. The recorded macro uses the AutoFill method because that's how the user entered the random numbers, but it can be done more efficiently in code:

```
Sheets("Sheet5").Range("C1:C20").Formula="=RAND()"
```

When you optimise recorded code, think about what you are trying to do with the macro. There is often a faster way to do something in VBA code that what has been recorded from keystrokes and actions taken by the user.

Removing Unnecessary Recorded Expressions

Another reason the macro recorder produces inefficient code is that ti cannot tell which options you've changed in a dialog box - it therefore explicitly sets all available options. For example, selecting cells B2:B14 and then changing the font style to bold using the Format Cells dialog box produces this code:

```
Range("B2:B14").Select
With Selection.Font
      .Name="Arial"
      .FontStyle="Bold"
      .Size=10
      .Strikethrough=False
      .Superscript=False
      .Subscript=False
      .OutlineFont=False
      .Shadow=False
      .Underline=xlNone
      .ColorIndex=xlAutomatic
End With
```

Setting the cell format to bold can be carried out with a single line of code without selecting the range:

```
Range("B2:B14").FontStyle=Bold
```

Minimising the Use of Variant Variables

Although you may find it convenient to use variant types in your code, it is wasteful of storage and slower to process such variables. Declare your variables explicitly wherever possible.

Use Specific Object types

If you declare object variables with the Object generic type, VBA may have to resolve their references at run-time; if you use the specific object declaration VBA can resolve the reference at compile-time.

Use Constants

Using declared constants in an application makes it run faster, since it evaluates and stores the constant once when the code is compiled.

Use Worksheet Functions When Possible

An Excel worksheet function that operates on a range of cells is faster than VBA code doing the same thing on those cells. For example:

```
For Each c in Worksheets(1).Range("A1:A200")
      totalVal=totalVal+c.Value
Next
```

should be replaced by

```
totVal=Application.Sum(Worksheets(1).Range("A1:A200"))
```

Using Special-Purpose VBA Methods

There are also several special-purpose VBA methods that offer a concise way to perform a specific operation on a range of cells. Like worksheet functions these specialised methods are faster than the general-purpose VBA code that accomplishes the same task. For example, the following code changes the value in each cell in a range in a relatively slow way:

```
For Each c in Worksheets(1).Range("a1.a200").Cells
      If c.Value=4 then c.Value=4.5
Next
```

The following code uses the **Replace** method, and is much faster:

```
Worksheets(1).Range("a1:a200").Replace "4", "4.5"
```

For more information about special-purpose VBA methods, see the relevant Help topic and look at the list of the object's methods.

Turning off Screen Updating

A macro that affects the appearance of a worksheet or chart will run faster when screen updating is turned off. Set the **ScreenUpdating** property to **False**:

```
Application.ScreenUpdating=False
```

Excel automatically sets the ScreenUpdating property back to True when your macro ends.

6. ACTIVEX CONTROLS AND DIALOG BOXES

Some applications that you develop may require the initial selection of options and choices. To achieve this you can use a range of controls, such as buttons, check boxes, option buttons and list boxes to create a custom user interface. This section discusses how to use controls and dialog boxes to manage the way the user interacts with your application. The following section discusses the use of menus and toolbars to achieve a similar result. Each of these enhancements has its advantages and disadvantages - you must decide which is the most appropriate.

Choosing the Best User Interface Enhancement

Controls, such as buttons and check boxes, can be placed on worksheets or chart sheets next to the data they access so that using them causes only minimal disruption. On the other hand, since controls are tied to one sheet you need to re-create them if you want to access the macros from elsewhere.

If you need to display a single message, or ask the user for a single piece of information, you can use a message box or input box. These pre-defined dialog boxes are easy to create and use, but have only restricted uses.

You can place controls on a dialog sheet to create a custom dialog box - these are useful when you want to manage a complex interaction between the user and the application. They do not, however, offer the quickest access to commands and, since they are stored on a separate sheet, can interrupt the flow of work.

Whereas dialog boxes offer the user a set of complex options and return information to the user, and controls offer the most visually obvious connection to the data on which they act, menus and toolbars offer a quicker and more convenient way to expose options and commands to the user.

Using Built-in Dialog Boxes

Before adding custom controls or dialog sheets to your application, consider whether a built-in dialog box meets your needs.

Using Message and Input Dialog Boxes

The following table lists the functions and methods for adding pre-defined dialog boxes to your VBA application:

Use this	To do this
MsgBox function	Display a message and return a value indicating the command button the user clicked
InputBox function	Display a prompt and return the text the user typed
InputBox method	Display a prompt and return the information the user entered. this is similar to the InputBox function, but provides additional functionality, such as requiring the input to be of a specified type.

Message Box

This creates a simple dialog box that can display a short message, an icon and a predefined set of buttons. The simplest message box contains only a message string and an "OK" button:

```
MsgBox "This is a message"
```

The general syntax is: **MsgBox** <string>,<buttons>,<title>, where:

`string` is the text string you want in the message.
`buttons` is a numeric value which determines the buttons and icon shown (see table)
`title` is the string appearing in the title bar of the message box.

The numeric value of buttons is determined according to the following table:

Input values		Return values	
string	value	string	value
vbOKOnly	0	vbOK	1
vbOKCancel	1	vbCancel	2
vbAbortRetryIgnore	2	vbAbort	3
vbYesNoCancel	3	vbRetry	4
vbYesNo	4	vbIgnore	5
vbRetryCancel	5	vbYes	6
vbCritical	16		
vbQuestion	32		
vbExclamation	48		
vbInformation	64		
vbDefaultButton1	0		
vbDefaultButton2	256		
vbDefaultButton3	512		
vbApplicationModal	0		
vbSystemModal	4096		

You can use the built-in constants (Excel will automatically insert the correct numerical value) or use the numbers - the former is recommended.

Once you have selected what combination of buttons and icon you require, you construct the value for *buttons* by adding together the values. The return value will determine the button chosen by the user.

Example: Create a message box with "YES", "NO" and "CANCEL" buttons, with the question mark icon, make the "NO" button default, and select it as **ApplicationModal** (require a value to be selected before being allowed to continue with the application):

```
buttons=vbYesNoCancel + vbQuestion + vbDefaultButtons2
x=MsgBox("Do you want a game of Scrabble?",buttons,"Game query")
Select Case x
      Case vbYes
            MsgBox "You chose yes!"
      Case vbNo
            MsgBox "You chose no!"
      Case vbCancel
            MsgBox "You chose cancel!"
End Select
```

InputBox

The **InputBox** function creates and displays a simple dialog box containing a prompt, an edit box, and OK and Cancel buttons. If you require a more elaborate dialog box, you must create a custom dialog box using a dialog sheet (see later). The return value from the **InputBox** function is the string entered by the user. If the input box is empty, or if the user pressed Cancel the return value is an empty string. The following displays a simple input box:

```
radius=InputBox("Enter the circle radius:", "Circle Radius")
```

The **InputBox** method of the **Application** object is similar but allows you to specify the desired data type for the data entry (a range, or a string for example). If the user enters data with an incorrect type, Excel displays a message indicating this.

If a data type is specified, the return value from the **InputBox** method has that data type if the user pressed Enter or OK. If the data type is not specified the return value is a string. In either case, the return value is **False** if the user pressed Cancel or Esc to cancel the dialog box. The full syntax includes the facility to specify the screen location of the input box and context-sensitive help (see the Help file) but the main parameters are:

```
result=Application.InputBox(Prompt:="....", Type:=n)
```

The value of n may be one of the following:

0	Formula
1	Number
2	String
4	Logical
8	Range
16	Error
64	Array of values

The following code uses the **InputBox** method to ask the user for a search range and a search value. the search range must be a valid **Range** reference and the search value must be a number.

```
Sub CountValues()
     cellCount=0
     Set rangeToSearch = Application.InputBox(Prompt:="Enter the range
            to search", type:=8)    'type=8 - must be a range object
     searchValue = Application.InputBox(Prompt:="Enter the search
            value", Type:=1)        'type=1 - must be a number
     If searchValue=False then Exit Sub         'user clicked Cancel
     For Each c in rangeToSearch
         If c.Value=searchValue then cellCount=cellCount+1
     Next
     MsgBox cellCount
End Sub
```

Displaying Built-in Excel Dialog Boxes

In addition to the message box and input box, Excel has over 200 other built-in dialog boxes each of which allows the user to perform actions. For example, the built-in File Open dialog box allows the user to open a file, and the Clear dialog box allows the user to clear a range of cells.

The **Dialogs** method returns a built-in dialog box. This method takes as argument a built-in constant that begins with "xlDialog" and corresponds to a dialog box name. For example, the constant for the File Find dialog box is **xlDialogFileFind**. The **Show** method displays the dialog box. To display the built-in File Open dialog box, with the default directory set to C:\Windows\Excel:

```
Application.Dialogs(xlDialogOpen).Show("C:\Windows\Excel")
```

The dialog box remains open until the user has dismissed or cancelled it - the return value is **True** if the user clicked OK or pressed ENTER, **False** if the user clicked Cancel or pressed ESC. Built-in dialog boxes are fixed - you cannot modify them in any way - but you can create a custom dialog box that looks just the same (see later).

6.1 USING ACTIVEX CONTROLS

You can place controls, such as buttons, check boxes, list boxes and so on, on worksheets, chart sheets or userforms - not on a module. Placing controls on a userform creates a custom dialog box. Controls on worksheets and charts have the benefit of being closely tied to the data they use; custom dialog boxes are best when you want to use the same set of controls with a number of different worksheets - you retain generality.

Using Custom Controls in your Application

The controls may be selected from the Forms toolbar. You select the control by clicking on it and draw it on your sheet (click on the sheet and drag until the control's outline is the size and shape you want). You can size it automatically to fit cells by holding down the ALT key as you draw it.

You can also add a control using the **Add** method for the appropriate control collection. Arguments specify the position of the top left corner of the control and its height and width:

```
Worksheets("Sheet1").Buttons.Add 50, 25, 100, 20
```

After placing the control, a dialog box appears from which you can set the initial properties of the control. You can also assign a macro/VBA procedure to the control - when the user clicks the button, check box or whatever, Excel runs the associated procedure.

You can also link a control directly to a cell on a worksheet without using procedures. This way you can simplify the user interface for a worksheet so that the user can set options using the mouse rather than by typing data into a cell. For example, you can link a checkbox to a cell, and when the check box is selected that call contains the value **True**, when de-selected it contains the value **False**. Your procedure can make use of this value in carrying out its calculation(s). See later section "Linking Controls to Worksheet Cells".

Selecting a Control

Selecting a control is different from *clicking* it. The latter runs the associated macro, the former allows editing of the properties of the control. You select a control either by right-clicking it or by holding down the CTRL key while you left-click it. In code you can select a control using the **Select** method.

Setting Control Properties

Default properties are initially assigned to controls, but these may be changed by selecting "Format Object" from the pop-up menu that appears when you right-click on the control.

Assigning Code to Controls

You can assign a procedure to a control either at the time the control is created or by selecting the appropriate item from the pop-up menu. The code will be executed when the appropriate action occurs to the control. This can also be achieved in code:

```
With WorkSheets(1)
      .buttons(1).OnAction="Macro1"
End With
```

The above code will assign the "Macro1" procedure to execute whenever the first button on the worksheet is clicked. TIP: you can use the Caller property of the Application object to determine which button was pressed. For buttons, the Caller property returns the button name.

Linking Controls to Worksheet Cells

Some controls can be linked to worksheet cells. If the cell value changes the control value changes, and vice versa. Any formulas referencing the linked cell will therefore be recalculated when you change the control. Worksheet cells can be linked to check boxes, list boxes, drop-down list boxes, option buttons, scroll bars and spinners. The link is a property of the control, not of the linked cell. You can link one cell to several controls, but a control can only be linked to one cell. You can reference the linked cell in any other cell where you want to use its value. Cells may be linked in VBA code as follows:

```
Worksheets("Sheet1").CheckBoxes("Check Box 3).LinkedCell="Sheet1!A5"
```

List boxes and drop-down list boxes also use the **ListFillRange** property, which sets the worksheet range used to fill the list box. e.g.

```
Worksheets("Sheet1").ListBoxes("List Box 2).ListFillRange="Sheet1!a5:a10"
```

Check Boxes

The value in the linked cell can be checked (**True**), unchecked (**False**) or greyed-out (#N/A), reflecting the state of the check box. Entering one of these values in the linked cell changes the value of the check box accordingly; manually changing the check box changes the value in the linked cell.

Option Buttons

For grouped option buttons that are all linked to the same cell, the value in the linked cell shows the ordinal number of the option button that is turned on. For example, if an option button group contains four buttons and the third one has been selected, the value in the linked cell will be 3. Note that for option buttons in an option button group, selections are mutually exclusive. Also, changing the value in the linked cell will change the selection; setting the value of the linked cell to <1 or >number in the group will de-select all buttons in the group.

List Boxes

For a single-select list box the value in the linked cell shows the ordinal number of the selection in the list. For a multiple-selection list box the value in the linked cell has no meaning.

Scroll Bars

The value in the linked cell specifies the current value of the scroll bar control. The Format Object dialog box allows you to specify minimum and maximum values for the scroll bar and the amount by which the position value changes when the user clicks the arrows or the scroll bar. You can also specify these values using the Min, Max,

SmallChange and LargeChange properties. Changing the value in the linked cell changes the position of the scroll box; values less than the specified minimum or greater than the specified maximum move the scroll box to the minimum or maximum positions respectively.

Spinners

The value in the linked cell represents the value of the spinner. Unlike a scroll bar, a spinner has no visible position indicator. However, you set minimum and maximum values for a spinner in just the same way, and the spinner value increases when you click the up arrow, and decreases when you click the down arrow by an amount specified by the **SmallChange** property.

7. WORKING WITH EVENTS

Visual programming is all about responding to **events** - that is why it is often referred to as event-driven programming. The basic idea is that small chunks of code are attached to event procedures - the action of clicking a button called Button1, for example, will run a piece of code called Button1_click.

An *event* in Excel is the occurrence of an action, such as opening a workbook, switching to a sheet, using a particular key combination or recalculating a worksheet. Some events are initiated by Excel and some by the user, but in each case by assigning procedures to these events you can enhance the way users interact with your application.

There are three main classes of event-driven procedures, organised according to the way in which they are associated with events. You can associate a procedure with...

1. the action of clicking a button or other object placed on a worksheet. This is carried out by using the "Assign Macro" command on the object's properties menu
2. one of a specific set of events by giving the procedure a special automatic procedure name that begins with "Auto_"
3. a defined event for an object by setting an OnEvent property of the object (such as the **OnWindow** or **OnCalculate** property) to the procedure name.

If you've used Visual Basic or a similar event-driven language (Delphi, JBuilder etc) you are already familiar with this type of programming. Most of the code in these languages is written to respond to events, such as when the user clicks on a button or a list box. In previous versions of Excel you may have used properties such as **OnSheetActivate** or **OnEntry** to cause a macro to run when a sheet is activated or changed. This is also event-driven programming. Office 97/2000 expands on the list of events and adds event procedures that receive arguments.

In Excel 97/2000 you can write event procedures at the worksheet, chart, workbook or application level. For example, the **Activate** event occurs at the sheet level, and the **SheetActivate** event is available at both the workbook and application levels. The **SheetActivate** event for a workbook occurs when any sheet in that workbook is activated. At the application level, this event occurs when any sheet in any open workbook is activated.

Creating Automatic Procedures

An *automatic procedure* runs automatically whenever one of a specific set of either workbook-level or worksheet-level events occurs.

Workbook-Level Automatic Procedures

The following table lists the automatic procedures relating to workbook-level events:

Procedure Name	**Event that causes the procedure to run**
Auto_Open	User opens the workbook containing the procedure
Auto_Close	User closes the workbook containing the procedure
Auto_Add	User installs the add-in that contains the procedure, or the Installed property of the add-in is set to True.
Auto_Remove	User removes the add-in that contains the procedure or the Installed property of the add-in is set to False.

Note that these macros do not operate if the action is carried out by a macro.

Note also that you can prevent an automatic procedure from running by holding down the SHIFT key while carrying out the action manually.

CREATING ON_EVENT PROCEDURES

Certain objects in VBA have properties and methods that are associated with specific events. The **Button** object, for example, has an **OnAction** property which is associated with clicking the button; the **Application** object has an **OnRepeat** method, which is associated with clicking Repeat on the Edit menu. These events are generated by Excel, sometimes as a result of user input. Other events, such as the arrival of data from another application via OLE are not (see the **OnData** property).

If you associate a procedure with one of these properties or methods, that procedure - called an *OnEvent procedure* or an *event handler* - will run whenever the event associated with the method or property occurs.

Note that, although they still work, these events have been superseded by new functions and methods in Office 97/2000. See "Worksheet, Chart, Workbook and Application Events" later in this section.

The following table lists the properties and methods you can use to trap events:

Property or Method	**Event causing the associated procedure to run**
OnAction	Clicking a control or graphic object, clicking a menu command or clicking a toolbar button
OnCalculate	Recalculating a worksheet
OnData	The arrival of data from another application by way of DDE or OLE
OnDoubleClick	Double-clicking anywhere on a chart sheet, dialog sheet, module or worksheet
OnEntry	Entering data using the formula bar or editing data in a cell
OnKey	Pressing a particular key or key combination
OnRepeat	Clicking repeat on the Edit menu
OnSheetActivate	Activating a chart sheet, dialog sheet, module, worksheet, workbook or Excel itself
OnSheetDeactivate	Deactivating a chart sheet, dialog sheet, module, worksheet, workbook or Excel itself
OnTime	Waiting until a specific time arrives, or waiting for a specific time delay
OnUndo	Clicking Undo on the Edit menu
OnWindow	Activating a window

There are several steps involved in creating and using any type of OnEvent procedure:

1. You must create the procedure (the event handler) you want to run when the specified event occurs
2. Elsewhere in your VBA code you must associate the event handler with the event you want to respond to. This is called trapping the event.
3. When you want to stop trapping this event, you must disassociate the event from the event handler.

To associate an event with an OnEvent procedure

Set the property associated with the event - or set the procedure argument of the method associated with the event - to the name of the OnEvent procedure:

```
Activesheet.Buttons("MyButton").OnAction="ButtonClickHandler"
Application.OnRepeat text:="Paste Again", procedure:="PasteAgain"
```

To disassociate an event from an OnEvent procedure

Set the property associated with the event - or set the procedure argument of the method associated with the event - to the empty string "":

```
Activesheet.Buttons("MyButton").OnAction=""
Application.OnRepeat text:="Paste Again", procedure:=""
```

The following sections show how event trapping is carried out for each of the OnEvent procedures:

OnAction Property

Example given above

OnCalculate Property

You can, for example, use an OnCalculate handler to update column widths when new data are recalculated, as shown in the following example:

```
Sub TrapCalculate()
      Application.OnCalculate="FitColumns"
End Sub

Sub FitColumns
      Columns("A:H").EntireColumn.AutoFit
End Sub
```

OnEntry Property

An OnEntry event handler runs when the user either enters data on a worksheet using the formula bar or edits data in a cell. It runs after the user enters data in a cell or in the formula bar and then either presses ENTER, selects another cell or clicks in the enter box on the formula bar. The following code checks data entered in a cell on column B of the specified worksheet:

```
Sub TrapEntry()
   ActiveWorkbook.Worksheets("Sheet1").OnEntry = "ValidateColB"
End Sub

Sub ValidateColB()
   With ActiveCell
      If .Column=2 then
           If IsNumeric(.Value) then
            If .Value<0 or .Value>255 then
                 MsgBox "Entry must be between 0 and 255"
                 .Value=""
            End If
           Else
            'non-numeric entry
            MsgBox "Entry must be a number between 0 and 255"
```

```
            .Value=""
           End If
      End If
   End With
End Sub
```

OnKey Method

An OnKey event handler runs when the user presses a specified key combination. Unlike many other OnEvent handlers the OnKey handler does run if you use the **SendKeys** method to simulate sending keystrokes to Excel under program control.

The following code runs the DoReports procedure when the user presses F12:

```
Sub TrapKeys()
   Application.OnKey key:="{F12}", procedure:="DoReports"
End Sub
```

OnTime Method

This event handler only works if Excel is running and the workbook containing the OnTime event handler is loaded. For example, to accumulate and print a set of reports every day at noon:

```
Sub TrapTime()
   Application.OnTime earliestTime:=TimeValue("12:00:00"), _
    procedure:=DoReports
End Sub
```

Note: with the OnTime procedure, a user can work until the specified time. With the Wait method, the user cannot interact with Excel until the wait period is over:

```
'wait 15 seconds
Application.Wait Now+TimeValue("00:00:15")
```

OnWindow Property

The OnWindow event handler runs whenever the user switches to a window. To associate a procedure with switching to any window in the Excel application:

```
Application.OnWindow = "AllWindowHandler"
```

7.1 Worksheet Events

Worksheet, chart sheet, and workbook event procedures are available for any open sheet or workbook. Note that to write event procedures for an embedded chart or for the Application object, you must create a new object using the WithEvents keyword in a class module.

Use the EnableEvents property to enable or disable events. For example, using the Save method to save a workbook causes the BeforeSave event to occur. You can prevent this by setting the EnableEvents property to False before you call the Save method.

```
Application.EnableEvents = False
ActiveWorkbook.Save
Application.EnableEvents = True
```

Events on sheets are enabled by default. To view the event procedures for a sheet, right-click the sheet tab and click View Code on the shortcut menu. Select the event name from the Procedure drop-down list box.

Worksheet-level events occur when the user activates a worksheet, or changes a worksheet cell, as shown below:

Event	**Occurs when ..**
Activate	the user activates the sheet. Use this instead of the **OnSheetActivate** property
BeforeDoubleClick	the user double-clicks a worksheet cell. Use this event instead of **OnDoubleClick**.
BeforeRightClick	the user right-clicks a worksheet cell.
Calculate	the user recalculates a worksheet. Use instead of **OnCalculate**.
Change	the user changes a cell formula. Use instead of **OnEntry**.
Deactivate	the sheet is active and the user activates a different sheet. Does not occur when the user shifts focus from one window to another showing the same sheet. Use instead of **OnSheetDeactivate**.
SelectionChange	the user selects a worksheet cell.

See the help file for more details about each event.

This example adjusts the size of columns A-F whenever the worksheet is recalculated:

```
Private Sub WorkSheet_Calculate()
   Columns("A:F").AutoFit
End Sub
```

7.2 Chart Events

Chart events occur when the user activates or changes a chart. Events on chart sheets are enabled by default. To view the event procedures for a sheet, right-click the sheet tab and select View Code from the shortcut menu. Select the event name from the Procedure drop-down list box.

Like worksheet-level events, chart-level events occur when the user activates or changes a chart, as shown below:

Event	Occurs when ..
Activate	the user activates the chart sheet (does not work with embedded charts). Use this instead of the **OnSheetActivate** property
BeforeDoubleClick	the user double-clicks a chart. Use this event instead of **OnDoubleClick**.
BeforeRightClick	the user right-clicks a chart.
Calculate	the user plots new or changed data.
Deactivate	the sheet is active and the user activates a different sheet. Does not occur when the user shifts focus from one window to another showing the same sheet. Use instead of **OnSheetDeactivate**.
DragOver	the user drags data over the chart
DragPlot	the user drags a range of cells over the chart
MouseDown	the user clicks a mouse button while the mouse pointer is positioned over the chart
MouseMove	the user moves the pointer over the chart
MouseUp	the user releases a mouse button while the pointer is positioned over the chart
Resize	the user changes the size of the chart
Select	the user selects a chart element
SeriesChange	the user changes the value of a chart data point

Events for chart sheets are available by default in the VBA editor. To write event procedures for an embedded chart, you must create a new object using the **WithEvents** keyword in a class module. For more information, *see Using Class Modules with Events*.

This example changes a point's border colour when the user changes the point value.

```
Private Sub Chart_SeriesChange(ByVal SeriesIndex As Long, _
ByVal PointIndex As Long)
    Set p = ActiveChart.SeriesCollection(SeriesIndex).Points(PointIndex)
    p.Border.ColorIndex = 3
End Sub
```

7.3 Workbook Events

Workbook events occur when the workbook changes or when any sheet in the workbook changes. Events on workbooks are enabled by default. To view the event procedures for a workbook, right-click the title bar of a restored or minimised workbook window and click View Code on the shortcut menu. Select the event name from the Procedure drop-down list box.

Event	Occurs ..
Activate	when the user activates the workbook
AddInInstall	when the user installs the workbook as an add-in. Use instead of the **Auto_Add** macro.
AddInUninstall	when the user uninstalls the workbook as an add-in. Use instead of the **Auto_Remove** macro.
BeforeClose	before the workbook closes. Use instead of the **Auto_Close** macro.
BeforePrint	before the workbook is printed
BeforeSave	before the workbook is saved. Use instead of the **OnSave** property
Deactivate	when the workbook is active and the user activates a different workbook.
NewSheet	after the user creates a new sheet
Open	when the user opens the workbook. Use instead of the **Auto_Open** macro.
SheetActivate	when the user activates a sheet in the workbook. Use instead of the **OnSheetActivate** property.
SheetBeforeDoubleClick	when the user double-clicks a worksheet cell (not used with chart sheets). Use instead of the **OnDoubleClick** property.
SheetBeforeRightClick	when the user right-clicks a worksheet cell (not used with chart sheets).
SheetCalculate	after the user recalculates a worksheet (not used with chart sheets). Use instead of the **OnCalculate** property.
SheetChange	when the user changes a cell formula (not used with chart sheets). Use instead of the **OnEntry** property.
SheetDeactivate	when the user activates a different sheet in the workbook. Use instead of the **OnSheetDeactivate** property.
SheetSelectionChange	when the user changes the selection on a worksheet (not used with chart sheets).
WindowActivate	when the user shifts focus to any window showing the workbook. Use instead of the **OnWindow** property.
WindowDeactivate	when the user shifts focus away from any window showing the workbook. Use instead of the **OnWindow** property.
WindowResize	when the user opens, resizes, maximises or minimises any window showing the workbook.

This example maximises Microsoft Excel when the workbook is opened

```
Sub Workbook_Open()
      Application.WindowState = xlMaximized
End Sub
```

7.4 Application Events

Application events occur when a workbook is created or opened or when any sheet in any open workbook changes.

Event	Occurs ..
NewWorkbook	when the user creates a new workbook
SheetActivate	when the user activates a sheet in an open workbook. Use instead of the **OnSheetActivate** property.
SheetBeforeDoubleClick	when the user double-clicks a worksheet cell in an open workbook (not used with chart sheets). Use instead of the **OnDoubleClick** property.
SheetBeforeRightClick	when the user right-clicks a worksheet cell in an open workbook (not used with chart sheets).
SheetCalculate	after the user recalculates a worksheet in an open workbook (not used with chart sheets). Use instead of the **OnCalculate** property.
SheetChange	when the user changes a cell formula in an open workbook (not used with chart sheets). Use instead of the **OnEntry** property.
SheetDeactivate	when the user deactivates a sheet in an open workbook. Use instead of the **OnSheetDeactivate** property.
SheetSelectionChange	when the user changes the selection on a sheet in an open workbook.
WindowActivate	when the user shifts focus to an open window. Use instead of the **OnWindow** property.
WindowDeactivate	when the user shifts focus away from an open window. Use instead of the **OnWindow** property.
WindowResize	when the user resizes an open window.
WorkbookActivate	when the user shifts the focus to an open workbook
WorkbookAddInInstall	when the user installs a workbook as an add-in.
WorkbookAddInUninstall	when the user uninstalls the workbook as an add-in
WorkbookBeforeClose	before an open workbook is closed.
WorkbookBeforePrint	before an open workbook is printed.
WorkbookBeforeSave	before an open workbook is saved.
WorkbookDeactivate	when the user shifts focus away from an open workbook
WorkbookNewSheet	when the user adds a new sheet to an open workbook.
WorkbookOpen	when the user opens a workbook.

Using Class Modules with Events

To write event procedures for the Application object, you must create a new object using the **WithEvents** keyword in a class module.

Unlike sheet events, embedded charts and the **Application** object do not have events enabled by default. Before you can use events with an embedded chart or with the **Application** object, you must create a new class module and declare an object of type **Class** or **Application** with events. You use the **Class Module** command (**Insert** menu) in the VBA editor to create a new class module.

To enable the events of the **Application** object, you would add the following declaration to the class module:

```
Public WithEvents App as Application
```

After the new object has been declared with events, it appears in the **Object** box in the class module, and you can write event procedures for the new object. When you select the new object in the **Object** box, the valid events for that object are listed in the **Procedure** box.

Before the procedures will run, however, you must connect the declared object in the class module to the **Application** object. You can do this from any module by using the following declaration (where "EventClass" is the name of the class module you created to enable events).

```
Public X as New EventClass
```

After you have created the `X` object variable (an instance of the EventClass class), you can set the App object of the EventClass class equal to the Microsoft Excel **Application** object.

```
Sub InitializeApp()
   Set X.App = Application
End Sub
```

After you run the InitializeApp procedure, the App object in the EventClass class module points to the MSExcel Application object, and the event procedures in the class module will run whenever the events occur.

Although this all seems like hard work, one advantage is that you can use the same event procedure for many objects. For example, suppose that you declare an object of type **Chart** with events in a class module, as follows:

```
Public WithEvents cht As Chart
```

You can then use the following code to cause the event procedures to run whenever an event occurs for either chart one or chart two.

```
Dim C1 as New EventClass
Dim C2 as New EventClass
Sub InitializeCharts()
      Set C1.cht = Worksheets(1).ChartObjects(1).Chart
      Set C1.cht = Worksheets(1).ChartObjects(1).Chart
End Sub
```

You can declare **Worksheet** or **Workbook** objects with events in a class module and use the events in the new class with several sheets, in addition to the default event procedures. You might use this technique to write an **Activate** event handler that runs only when either sheet1 or sheet5 is activated. Or you can use a **Chart** object declared in a class module to write an event handler for both embedded charts and chart sheets.

8. Using Custom Dialog Boxes - The UserForm

When you need to manage a complex interaction between the user and your application, it is often necessary to create a custom dialog box. This allows you to present all the options together, making it easier to use and read than controls placed directly on a chart or worksheet. When the options have been selected the dialog box is dismissed - this way, the controls only take up screen space as long as they are needed.

In Office 97/2000, the concept of the dialog box has been refined from the Office 95/Excel 7 implementation to that of the **UserForm**. In effect, this means that you can generate custom forms or windows in a similar way to stand-alone Windows programs, such as Visual Basic, so that in principle , your VBA macro could run entirely separately from the worksheet, chartsheet, document, or whatever. The 'dialog box' becomes a vehicle both for the input of user choices and selection of options, as well as for the display of program output. This can also be an interactive process, in which the display can be dynamic.

Creating and managing a custom dialog box is more complex than using controls directly on a sheet however, and you must consider the trade-off between the convenience to the user and the amount of work involved for you as the programmer.

Excel, Word and PowerPoint share powerful new tools for creating custom dialog boxes - you only need to learn the mechanism for generating custom dialogs once and the resulting forms can be used with any of these applications.

Once you have created a custom dialog box, you can add ActiveX controls (previously known as OLE controls) to it in the same way that we saw previously that you could place them directly onto a worksheet. You determine the way in which custom dialog boxes and controls respond to specific user actions - for example, clicking a control or changing its value - by writing event procedures that run whenever a specific event occurs.

Designing Custom Dialog Boxes

To create a custom dialog box you must create a **UserForm** to contain controls, add controls to the form, set the properties for the controls, and write the code that generates the necessary response to form and control events.

Creating a New Dialog Box

Every custom dialog box in your project is a UserForm. New UserForms contain a title bar and an empty area in which to place controls. To create one, click **UserForm** on the **Insert** menu in the VBA Editor.

Use the **Properties** window to set the properties for the form - that is, its name and appearance.

Adding Controls to a Custom Dialog Box

Use the **Toolbox** to do this. If it's not already visible, click **Toolbox** on the **View** menu. You add controls by either dragging them from the toolbox, or by selecting and then drawing them on the form. You can resize and move them as required once they appear on the form.

Note: dragging a control (or a number of "grouped" controls) from a custom dialog box back to the Toolbox creates a template of that control which you can then reuse later. This is useful for producing a standard look and feel for your applications.

After you've added controls to the form, use the commands on the **Format** menu, or the buttons on the **UserForm** toolbar in the VBAEditor to adjust the alignment and spacing of the controls. Use the **Tab Order** dialog box (**View** menu) to set the tab order of the controls on the form.

➢ **Test Exercise 1:** Design and run a custom dialog box

- Create a new UserForm
- Insert a Frame control on this form
- Add three OptionButton controls to the Frame
- Click Run Sub/UserForm on the Run menu, or select the run button on the toolbar

The custom dialog box is displayed, and you should be able to use the option buttons.

- Click the close button on the UserForm title bar to exit run mode and return to design mode.

Setting Control and Dialog Box Properties at Design Time

You can set some control properties at design time - before any macros are run. In design mode, right-click a control and then click **Properties** on the shortcut menu to display the **Properties** window. Property names are listed in the left-hand column in the window, and property values are listed in the right-hand window. You set a property value by typing in the new value in the space to the right of the property name.

➢ **Test Exercise 2:** Set control properties in design mode

1. Create a new UserForm
2. Add an Image control, a **CommandButton** control and a few other controls (you choose!)
3. Right click the **Image** control, select **Properties**, and find **Picture** in the list of properties. Browse for and select a picture file by clicking the ellipsis button (…) and clicking **OK**.
4. Click the **CommandButton** you added; the list of properties in the **Properties** window changes to those of command buttons. Modify the **Caption** property to read "Send Order".
5. Change the name of the command button to **cmdSendOrder**.
6. Change the **ControlTiptext** for the button to "Click here to send order".
7. Put "s" as the **Accelerator** property for the button. This is the shortcut key that allows the keyboard to be used to select the button.
8. Run the form and test it out.

➢ **Test Exercise 3:** Set UserForm properties in design mode

1. Select the UserForm (click on the form background).
2. Try modifying some of the form properties (e.g. **BackColor**, **Caption** etc).

Creating Tabs in Dialog Boxes

If you need a single dialog box to handle lots of controls, you can sort them into categories, create a dialog box with two or more tabs, and place each category on a separate tab. To do this, add a **MultiPage** control to the dialog box and then add controls to each tab (or page). To add, remove, rename or move a page in a **MultiPage** control, right-click one of the pages in design mode and select the relevant command from the shortcut menu.

NB: Do not confuse the **MultiPage** control with the **Tabstrip** control - there is a danger of this as they look similar! Each page of a **MultiPage** control contains a unique set of controls that you add and modify at design time. **TabStrip** controls are intended to provide a quick means of selecting one of several items from a list to

enable other controls to be updates at run-time. TabStrips do not act as containers for other controls.

Writing Code to respond to Dialog Box Events

Each form or control recognises a predefined set of events, which can be triggered either by the user or by the system. For example, a command button recognises a Click event that occurs when the user clicks on the button and a form recognises the Initialize event that occurs when the form is loaded. To specify how a form or control should respond to an event, you write an **event procedure**.

To do this, open the **Code** window by double-clicking the object you wish to write an event procedure for, and select the event name in the **Procedure** box (in the upper-right corner of the window). Event procedures include the name of the UserForm or control. For example, the name of the Click event procedure for the command button Command1 is Command1_Click.

- **Test Exercise 4: Write and run an event procedure for a command button**

1. Create a UserForm and add a **CommandButton**, a **CheckBox** and a **ComboBox**.
2. Access the Properties window for the command button and change its name to **cmdSendOrder**
3. Double-click the button to access the code window - the Click event procedure is displayed since it is the default event for that object.
4. Add this code to display a simple message box: `MsgBox "Hello"`
5. Run the dialog box to see the results.

To see all the events that command buttons recognise, click the down arrow next to the **Procedure** box in the **Code** window. Events that already have procedures written for them appear bold. Click an event name in the list to display its associated procedure.

To see the events for a different control on the same UserForm, or for the Userform itself, click the object name in the **Object** box in the **Code** window, and then click the arrow next to the **Procedure** box.

Using Custom Dialog Boxes

To exchange information with the user by means of a custom dialog box, you must display the dialog box to the user, respond to user actions in the dialog box, then either provide an appropriate response, or (if another procedure is supposed to run after the dialog box is dismissed) retrieve the information the user entered.

Displaying a Custom Dialog Box

When you want to display a custom dialog box to yourself for testing purposes you click Run Sub/UserForm on the Run menu in the VBA Editor. When you want to display the dialog box to a user, however, you use the Show method. This example displays the dialog box named "UserForm1"

```
UserForm1.Show
```

Getting and Setting Properties at Run-Time

If you want to set default values for controls in a custom dialog box, modify controls while a dialog box is visible, and have access to the information that a user enters in the dialog box, you must set and read the values of control properties at run-time.

Setting Initial Values for Controls

To set the initial value, or *default* value, that a control will have every time the dialog box is displayed, add suitable code to the Initialize event procedure for the UserForm that contains the control.

- **Test Exercise 5**: Write and run an Initialize event procedure for a UserForm

1. Create a new UserForm, and add a TextBox, a ListBox and a CheckBox to it.
2. Display the code window by double-clicking the UserForm. Find the Initialize procedure, and add this code to it:

```
Private Sub UserForm_Initialize()
   With UserForm1
      .Textbox1.Text = "Paul Jones"     'sets default text
      .Checkbox1.Value = True           'checks check box by default
      With .ListBox1
         .Additem "North"         'these lines populate the listbox
         .Additem "South"
         .Additem "East"
         .Additem "West"
         .ListIndex = 3           'selects the 4th item in the list
      End With
   End With
End Sub
```

   Note that the first item in arrays and collections is 0 by default.

3. Run the dialog box to see the results.

> **Use Me to Simplify Event Procedure code**
> In the preceding example, you can use the keyword Me instead of the code name of the UserForm. That is, you can replace the statement With UserForm1 with the statement With Me. The Me keyword used in code for a UserForm or a control on the UserForm represents the UserForm itself. This technique lets you use long descriptive names for control while still making code easy to write.

If you want to set the initial value (default value) for a control but you don't want that to be the initial value every time you call the dialog box, you can use VBA code to set the control's value before you display the dialog box that contains the control. The following example uses the Additem method to add data to a list box, sets the value of a text box, and displays the dialog box that contains these controls.

```
Private Sub GetUserName()
   With UserForm1
      .Listbox1.Additem "North"
      .Listbox1.Additem "South"
      .Listbox1.Additem "East"
      .Listbox1.Additem "West"
      .TextBox1.Text = "00000"
      .Show
   End With
End Sub
```

Setting Values to Modify Controls While a Dialog Box is Running

You can set properties and apply methods of controls and the UserForm while a dialog box is running. The following example sets the text (the **Text** property) of TextBox1 to "Hello": TextBox1.Text="Hello"

By setting control properties and applying control methods at run time, you can make changes in a running dialog box in response to a choice the user makes. For example, if you want a particular control to only be available while a particular check box is selected, you can write code that enables that control whenever the checkbox is checked and disables it otherwise.

Enabling a Control

You use the Enabled property of a control to make it available or not:

- **Text Exercise 6: Enable and disable controls at run time**

1. Create a new UserForm. Add a Checkbox control and a Frame control and then place three Optionbutton controls inside the frame.
2. Double-click the checkbox to access its code window. Place the following in its Change event procedure:

```
Private Sub Checkbox1_Change()
  With Me
   If .Checkbox1.Value = True then
     .OptionButton1.Enabled = False
     .OptionButton2.Enabled = False
     .OptionButton3.Enabled = False
   Else
     .OptionButton1.Enabled = True
     .OptionButton2.Enabled = True
     .OptionButton3.Enabled = True
   End If
  End With
End Sub
```

3. Run the dialog box; select and clear the check box to see how changing its state enables and disables the three option buttons.

Instead of making the buttons enabled or disabled, you could make them visible or invisible by toggling the state of the **Visible** property instead.

Setting the Focus to a Control

The control with the focus is the one that responds to keyboard input from the user. You set the focus to a control in a dialog box by using the **SetFocus** method of the control. For example: `Me.Checkbox1.SetFocus`

Displaying and Hiding Parts of a Dialog Box

You can set properties or apply methods of the UserForm itself while a dialog box is running. A common use of this is to expand a UserForm to reveal additional options when the user clicks a command button.

- **Test Exercise 7: Resize a UserForm at run time**

1. Create a new UserForm. Set the Height property to 180.
2. Add a CommandButton at the top of the form and a CheckBox to the bottom. The Top property for the checkbox should be at least 120.

3. Double-Click the userform to view the Code window. Place this code into the UserForm_Initialize procedure: Me.Height = 120. This will ensure that the control at the bottom of the dialog box is initially hidden.
4. Place the following code in the CommandButton1_Click procedure: Me.Height=300-Me.Height
5. Run the example. Clicking the command button should toggle the size of the window so that the checkbox is alternately displayed and hidden.

Browsing Data with a TabStrip Control

You can use a TabStrip control to view different sets of information in the same set of controls in a dialog box. For example, if you want to use one area of a dialog box to display contact information about a group of individuals, you can create a TabStrip control and then add controls to contain the name, address and phone number of each person in the group. You can then add a "tab" to the TabStrip control for each member of the group. After doing this you can write code that will update the controls to display data about an individual when you select that tab.

To add, remove or rename a tab in a tabstrip, right-click the control and use the short-cut menu.

The following example changes the value of textBox1 each time a different tab of TabStrip1 is clicked. The index number of the tab that was clicked is passed to the event procedure:

```
Private Sub TabStrip1_Click(ByVal Index as Long)
  If Index = 0 Then
      Me.TextBox1.text = "1, BogTrotter Lane"
  ElseIf Index = 1 Then
      Me.textBox1.Text= "55, Phlegm Close"
  End If
End Sub
```

Data Validation

There are times when you need to ensure that only data of a certain type are entered into a particular control. You can check the value entered either when the control loses the focus or when the dialog box is closed. This example prevents the user from moving the focus away from the TextBox1 text box without first entering a number:

```
Private Sub TextBox1_Exit(ByVal Cancel as MSForms.ReturnBoolean)
  If Not IsNumeric(TextBox1.Text) Then
      MsgBox "Please enter a numeric value"
      Cancel = True
  End If
End Sub
```

Notice that you set the Cancel argument of the control's **Exit** event procedure to **True** to prevent the control losing the focus.

To verify data before a dialog box closes, include code to check the contents of controls in the same routine that unloads the dialog box. If a control contains invalid data, use an **Exit Sub** statement to exit the procedure before the **UnLoad** statement can be executed.

Getting Values When the Dialog Box Closes

Any data that a user enters in a dialog box is lost when the dialog box is closed. If you return the values of controls in a UserForm after the form has been unloaded, you get the initial values for the controls rather than any values the user may have entered.

If you want to save the data entered in a dialog box by a user, you can do so by saving the information to module-level variables while the dialog box is still running. The following example displays a dialog box and saves the data that was entered:

```
'Code in module to declare public variables
Public comment as String

'code in form
Private Sub CommandButton1_Click()
  Module1.comment = TextBox1.Text
  Unload Me
End Sub

'code in module to display form
Sub ShowForm()
  UserForm1.Show
End Sub
```

Closing a Custom Dialog Box

Dialog boxes are always displayed as *modal*. This means that the user must close the dialog box before doing anything else. Use the **UnLoad** statement to unload a UserForm when the user indicates that they want to close the dialog box. Typically, you would provide a command button in the box that can be clicked to close the dialog box.

Using the Same Dialog Box in Different Applications

Excel, Word and Powerpoint share features for creating custom dialog boxes. You can create a UserForm in one of these applications and share it with the others.

- **To share a UserForm with another application:**

1. In the VBAEditor for the application in which you created the UserForm, right-click the UserForm in the **Project Explorer**, and then click **Export File** on the shortcut menu.
2. Choose a name to export the UserForm as, and click **Save**. The UserForm is saved with the .frm extension.
3. In the VBAEditor for the other application, right-click the target project and click **Import File** on the shortcut menu.
4. Select the name you gave the dialog box when you saved it, and click **Open**.

Note Not every UserForm can be exported and run in another application - if you import a UserForm containing Word-specific code into Excel you'll have problems getting it to work!

NB : For examples, see the sample application DLGTEST.XLS

9. MENUS AND TOOLBARS

An essential part of creating a useful custom application is providing a simple and consistent way for users to interact with the application. The last section dealt with dialog boxes, which allow a complex set of options to be presented together. Toolbars and menus, however, provide quick, convenient and widely accessible ways to expose commands to the user. Commands for performing related tasks can be grouped together, separator bars can be used to divide commands into logical groupings. Submenus and shortcut (popup) menus also offer ways to group related tasks. Toolbars contain graphic buttons that perform frequently used commands.

In Office 97/2000, menus and toolbars are easy to design and modify - all applications in the suite share the same basic customisation interface - the **Customize** dialog box. Furthermore, because all menus and toolbars are represented by the same type of object - the **CommandBar** object - they are easy to customise and control from VBA.

Tools for Modifying the User Interface

There are two tools you can use to customise menus bars and toolbars: the shared **Customize** dialog box and VBA. Although the **Customize** dialog box differs slightly from one Office application to the next, the programmable objects used to modify menu bars and toolbars are the same across all applications. This section describes the **Customize** dialog box and the shared programmable objects, as well as when and how to use these tools.

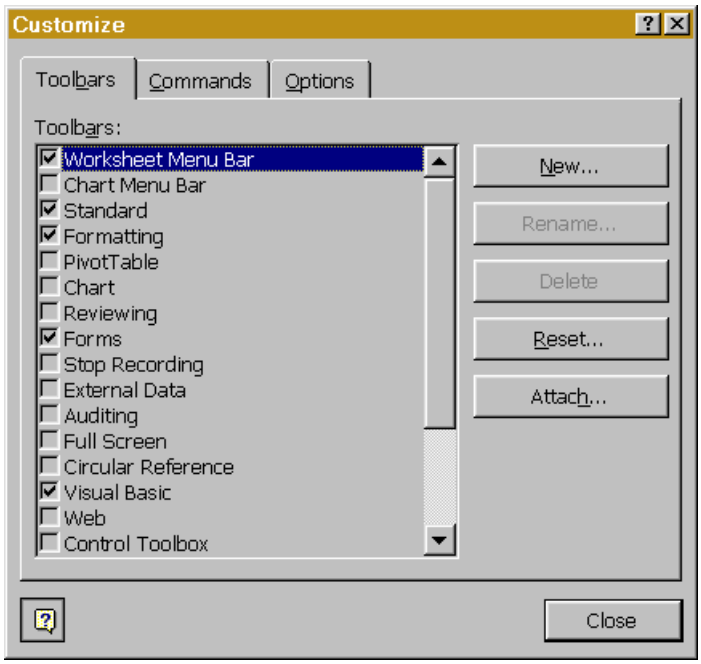
The Customize Dialog Box

The Office applications (excluding Outlook) provide a common interface - the **Customize** dialog box - for making design-time changes to your application. In cases where this method *or* VBA may be used, the **Customize** dialog box method is quicker and easier.

- **To display the customize dialog box**

  On the **View** menu, select **Toolbars**, then **Customize**

The following illustration shows the **Toolbars** tab in the **Customize** dialog box displayed by Excel:



The other Office applications all provide the same basic controls in this dialog box, but each have additional controls specific to that application.

To modify any built-in or custom dialog box, you follow the same basic procedure whichever Office application you are using:

1. In the Toolbars box on the Toolbars tab, select the check box next to the name of the menu bar or toolbar you want to display and modify. When you create a new menu bar or toolbar, it will be automatically displayed.
2. Click any menu item (including menu and sub-menu captions) or toolbar control to select it. The command associated with the control doesn't run while the **Customize** dialog box is open.
3. Right-click the item or control you've selected to display the shortcut menu containing the available options.

While the **Customize** dialog box is open you can rearrange items and controls by dragging and dropping them, and you can add new items and controls from the **Commands** tab.

VBA

In general, to create or modify toolbars, menu bars, and shortcut menus that you want to deliver with your Visual Basic application, you should use the customisation features of the container application. Changes made to toolbars, menu bars, and shortcut menus using the features of the container application are known as "design-time" changes. For information about using the container application to make design-time changes, see the online Help for that application.

You can add and modify toolbars, menu bars, and shortcut menus (and their component parts) by using the CommandBars portion of the Microsoft Office object model in Visual Basic code. You can write code that runs once to create toolbars and menu bars; in effect, the code simulates making design-time changes. In some container applications, however, you may be required to use a combination of this kind of Visual Basic code and the customisation interface to design your Visual Basic application. The following are some common areas where you must use a combination of code and the container application's interface:

- If your container application doesn't provide an interface for adding or modifying edit boxes, drop-down list boxes, or combo boxes on toolbars, you must use Visual Basic code to add and design one of these controls.
- If your container application provides an interface for creating toolbars but doesn't provide one for creating a new menu bar, you'll need to create a menu bar by using Visual Basic. After you've created the menu bar in Visual Basic, you can design menus on that menu bar by using the container application's interface.
- If your container application doesn't provide a way to display custom shortcut menus while the customisation interface is displayed, you must use Visual Basic code to modify those shortcut menus.

You can also write code that exists in your Visual Basic application to make changes to toolbars and menu bars while your application is running (for example, you can write code to disable a command on a menu bar under certain conditions, or to add buttons to a toolbar in response to a user's actions). Changes brought about by your code while your Visual Basic application is running are known as "run-time" changes.

9.1: MENUS

The menu system in each Office application is composed of the entire set of menus and the items on each menu. Each menu is either a menu, a submenu or a shortcut item. Each menu item is usually either a command or a submenu caption. In this section, the term *component* refers to any menu or menu item.

A **menu bar** is a bar at the top of the active window that displays the manes of all the menus that are available in that application at any given time. That is, an Office application can change the menu bar it displays in response to a change in the active window or in response to a VBA instruction. For example, when you edit an Excel chart the menu bar containing the charting menus is automatically displayed.

A **menu** is a list of menu items that appears (drops down) when you click a menu name on the menu bar.

A **submenu** (or *child menu*) is a menu that is attached to the side of another menu (the *parent menu*). Each submenu caption is marked by a right-pointing arrow. Submenus can be added to menus or shortcut menus.

A **shortcut** menu is a floating menu that contain frequently used commands and appears when you right-click on an object.

You can modify the menu system in many ways - by creating new menu bars, adding new menus to built-in or custom menu bars, adding new menu items, adding or modifying shortcut menus, and assigning macros to menu items. You can also restore the built-in menu system to its default state.

Using Text Boxes, List Boxes and Combo Boxes

Although you can add built-in or custom text boxes, list boxes or combo boxes to menus, shortcut menus and submenus, such controls are better suited to toolbars. If you want to all built-in controls, however, use the same techniques for adding built-in commands (see *Adding and Grouping Commands* later in this section). For custom controls, use the techniques given for adding them to toolbars (see *Design-Time Modifications to Toolbars*").

Design-Time Modifications to the Menu System

If you need to create a new menu bar, you can either use the **Customize** box in Access or VBA for the other Office applications.

In VBA you use the **Add** method of the **CommandBars** collection to create a new menu bar; the ***MenuBar*** argument of the **Add** method determines whether the **CommandBar** object you are creating can be displayed as a menu bar. This example generates a new menu bar with the name "New Menu":

```
Set xx = CommandBars.Add(Name:="New Menu", Position:=msoBarTop, _
   MenuBar:=True, Temporary:=False)
```

Adding Menus

You can add a menu to any built-in or custom menu bar, however, because Office applications can display different menu bars in different contexts, you may have to add a command to more than one menu bars to ensure that access to that function is always available.

To use the design-time method, say in Excel, select **Tools**, **Customize** from the menu. Select the **Commands** tab and scroll down until you find **New Menu** and drag it to your menu bar. When you edit the name of the menu, add an ampersand character (&) before the character you want to be the shortcut key.

In VBA you use the **Add** method of the **CommandBarControls** collection to add a menu to the **CommandBar** object representing a particular menu bar. Setting the *type* argument of the **Add** method to **msoControlPopup** indicates that the new control displays a menu - these are known as popup controls. The *before* argument indicates the position of the new menu. The **Caption** property is used to specify the menu name and access key. The following example adds a new menu named "Utilities" to the left of the **Window** menu on the normal worksheet menu bar:

```
Set xx = CommandBars("Worksheet Menu Bar").Controls _
   .Add(Type:=msoControlPopup, Before:=9)
xx.Caption = "&Utilities"
```

To remove the menu, you would you this code:

```
CommandBars("Worksheet Menu Bar").Controls("Utilities").Delete
```

Adding Menu Items and Sub-menus

This is carried out in a very similar way to the process of adding Menus. The process of doing this using the **Customize** box ought to be pretty obvious by now! Using VBA you would use the **Add** method of the **CommandBarControls** collection to add a submenu to the **CommandBar** object representing a particular menu. Also, setting the *type* argument of the **Add** method to **msoControlPopup** indicates that the new control is a sub-menu. This example adds a menu item "Macro 1" (which runs a macro called Macro1) and a sub-menu item "More Menus" at the end of the Utilities menu created above:

```
Set xx1 = CommandBars("Worksheet Menu Bar").Controls("Utilities") _
   .Controls.Add
xx1.Caption = "Macro &1"
xx1.OnAction = "Macro1"

Set xx2 = CommandBars("Worksheet Menu Bar").Controls("Utilities") _
   .Controls.Add(Type:=msoControlPopup)
xx2.Caption = "&More Macros"
```

Restoring Built-In Menu Components

You can restore built-in menu components, but not custom ones. Use the **Reset** method: e.g.

```
CommandBars("Worksheet Menu Bar").Controls("Edit").Reset
```

Run-Time Modifications of the Menu System

In addition to the ability of creating menu components from within VBA, you also have the ability to enable or disable menu components, or to make them visible or invisible. You could also rename them dynamically.

Displaying a Custom Menu Bar

To display a custom menu bar instead of the active menu bar, set the **Visible** property of the **CommandBar** object representing the custom bar to **True**. The custom bar will automatically replace the default one until you change the visibility back to **False** - the default bar is then restored.

Enabling or Disabling Menu Components

Disabled commands can still be seen, but are "greyed out". You can set the **Enabled** property of the item to **True** or **False** to toggle its state, e.g.

```
CommandBars("Worksheet Menu Bar").Controls("File") _
   .Controls.Add("Test Control").Enabled = False
```

If you want to disable all commands on a particular menu, you can disable the menu itself - the following disables the entire **File** menu on the worksheet menu bar:

```
Commandbars("Worksheet Menu Bar").Controls("File").Enabled = False
```

9.2: TOOLBARS

Each Office application provides a system of toolbars containing toolbar controls that the user can click to gain access to frequently-used commands. Each toolbar can appear docked at the top, bottom or sides of the application window, or as a floating window anywhere in the workspace. Each toolbar control is a simple graphical control with which the user can exchange information with your VBA application.

There are several types of controls that may be used on a toolbar:

The most common type is a simple button control that contains a graphic, called the *button image*, visually representing the function of the button. Another type is a button control that contains a graphic and an attached drop-down palette. The user clicks the drop-down arrow to display the palette, selects an option on the palette and then clicks the button control to apply the option. The **Font Color** button is an example of this.

A text box, list box or combo box can also be a toolbar control - the **Font Name** control is an example of this. This type of control is known as a pop-up control, and displays a menu of other controls.

Guidelines for Customising Toolbars

You can create new toolbars, add new buttons to existing toolbars, modify the button image and assign macros, ToolTip text and status bar text to toolbar buttons. In most cases you can choose whether to do this at design time, using the **Customize** dialog, or using VBA at run-time.

You can add menus (pop-up controls) to any built-in or custom toolbar. This is often easier than customising a toolbar with lots of buttons. For an example, see the **Draw** button on the **Drawing** toolbar. To do this, use the same technique as with adding these components to menu bars.

You can also add test, list and combo boxes to built-in or custom toolbars. To do this for built-in text, list and combo boxes, you can use the **Customize** dialog; for custom controls you must use VBA.

Adding a Custom Toolbar

These operations are straightforward and obvious using the **Customize** dialog, so we will concentrate on the VBA techniques.

You use the **Add** method of the **CommandBars** collection to add a new toolbar; setting the *Position* argument of the **Add** method to **msoBarLeft**, **msoBarTop**, **msoBarRight**, **msoBarBottom** or **msoBarFloating** indicates whether the new toolbar is floating, or docked:

```
Set xx=CommandBars.Add(Name:="Custom Tools", Position:=msoBarFloating, _
   MenuBar:=False, Temporary:=False)
xx.Visible = True
```

Adding and Modifying Toolbars

All host applications have an extensive interface for adding and designing custom toolbars (adding built-in buttons, adding macros as buttons, even adding pop-up controls to toolbars). The design-time changes you'll usually make from Visual Basic code are ones that add or modify combo box controls. Otherwise, working with toolbars in code is almost completely limited to making run-time changes (changing the button state, changing the button appearance, changing the button action, and so on).

Making run-time modifications to toolbars

There are several modifications you can make to a toolbar at run time. One of these modifications is to change the state of a command bar button on the toolbar. Each button control has two active states: pushed (True) and not pushed (False). To change the state of a button control, use the appropriate constant for the State property, as explained in the table later in this topic.

Another modification you can make at run time is to change the appearance or action of a button. To change the appearance of a button but not its action, use the **CopyFace** and **PasteFace** properties. These properties are useful if you want to copy the face of a particular button onto the Clipboard or import it into another application to change some of its features. Use the **PasteFace** property to transfer the button image from the Clipboard onto a specific button.

To change a button's action to a function you've developed, assign the custom procedure name to the button's **OnAction** property. The following table lists the most common properties and methods for changing the state, appearance, or action of a button.

Property or method	Description
CopyFace, PasteFace	Copies or pastes the image on the face of a button. Use the **CopyFace** method to copy the face of the specified button to the Clipboard. Use the **PasteFace** method to paste the contents of the Clipboard onto the face of the specified button. The **PasteFace** method will fail if the Clipboard is empty. If the image on the Clipboard is too large for the button face, the image won't be scaled down. Generally, it's more convenient to copy and paste a button face at design time, but you can also make changes to a button face at run time. You can also use the **FaceId** property to assign a different built-in button face to a button.
Id	Specifies the value that represents the button's built-in functionality. For example, a button that copies highlighted text to the Clipboard has an Id value of 19.
State	Specifies the appearance, or state, of the button. Can be one of the following constants: **msoButtonDown**, **msoButtonMixed**, or **msoButtonUp**.
Style	Specifies whether the button face displays its icon or its caption. Can be one of the following constants: **msoButtonAutomatic**, **msoButtonIcon**, **msoButtonCaption**, or **msoButtonIconandCaption**.
OnAction	Specifies the procedure to be run when the user clicks a button, displays a menu, or changes the contents of combo box controls.
Visible	Specifies whether the control is to be displayed or hidden from the user.
Enabled	Enables or disables a command bar; the name of a disabled command bar won't appear in the list of available command bars.

The following example assumes that the first two controls on the **CustomButtons** command bar are buttons. The **HideThem** procedure hides the first button and assigns a procedure to the **OnAction** property of the second button. When the **OnAction** procedure is run, the first control will be made visible and the second control will be hidden. If the second button is pressed while the procedure is running, the procedure will be halted. Note that you must declare both **startBtn** and **stopBtn** as global variables.

```
Sub HideThem()
  Set v = CommandBars("CustomButtons")
  Set startBtn = v.Controls(1)
  With startBtn
```

```
      .Visible = False
      .Caption = "Stop Processing"
    End With
    Set stopBtn = v.Controls(2)
    stopBtn.OnAction = "onActionButtons"
End Sub

Sub onActionButtons()
    stopBtn.Visible = False
    With startBtn
      .Visible = True
      .Style = msoButtonCaption
    End With
    Do While startBtn.State <> True
       'Continue processing sub
    Loop
End Sub
```

Adding and modifying combo box controls

Edit boxes, drop-down list boxes, and combo boxes are powerful new controls you can add to toolbars in your Visual Basic application. However, most container applications require that you use Visual Basic code to design these controls. To design a combo box control, you use the properties and methods described in the following table.

Property or method	Description
Add	Adds a combo box control to a command bar by specifying one of the following **MsoControlType** constants for the Type argument: **msoControlEdit**, **msoControlDropdown**, or **msoControlComboBox**.
AddItem	Adds an item to the drop-down list portion of a drop-down list box or combo box. You can specify the index number of the new item in the existing list, but if this number is larger than the number of items in the list, **AddItem** fails.
Caption	Specifies the label for the combo box control. This is the label that's displayed next to the control if you set the Style property to **msoComboLabel**.
Style	Specifies whether the caption for the specified control will be displayed next to the control. Can be either of the following constants: **msoComboLabel** (the label is displayed) or **msoComboNormal** (the label isn't displayed).
OnAction	Specifies the procedure to be run when the user changes the contents of the combo box control.

The following example adds a combo box with the label "Quarter" to a custom toolbar and assigns the macro named "ScrollToQuarter" to the control.

```
Set newCombo=CommandBars("Custom1").Controls.Add(Type:=msoControlComboBox)
With newCombo
   .AddItem "Q1"
   .AddItem "Q2"
   .AddItem "Q3"
    .AddItem "Q4"
    .Style = msoComboNormal
    .OnAction = "ScrollToQuarter"
End With
```

While your application is running, the procedure assigned to the **OnAction** property of the combo box control is called each time the user changes the control. In the procedure, you can use the **ActionControl** property of the **CommandBars** object to find out which control was changed and to return the changed value. The **ListIndex** property will return the item typed or selected in the combo box.

Adding and Grouping Controls

You can add controls to any toolbar and can visually separate them with lines into groups. The Customize dialog can be used if you wish. Note that you can make a copy of a pre-existing toolbar button by holding down the CTRL key while you select and drag it to the desired location. You can also add custom commands (i.e. macros) to toolbars.

Modifying the Appearance of Toolbar Buttons

The face of a toolbar button can be the image, the name or both - these are set by the "style" in the **Customize** dialog. While this dialog is open you can add or modify a toolbar button image:

To	Do this ..
Use a predefined image	Right-click the button, point to **Change Button Image**, and click the desired image
Copy and paste another button's image	Right-click the button containing the desired image, click **Copy Button Image**, right-click the target button and paste the copied image.
Copy and paste an image from a graphics program	Copy via the clipboard (cut & paste). Best to use a 16x16 pixel image if you can).
Edit the current button's image	Select **Edit Button Ima**ge from the right-click menu
Reset a button's image	Select **Reset Button Image** from the right-click menu.

Grouping Controls

You can group commands by inserting separator lines - use the **Customize** dialog (the **Begin Group**) command.

Using VBA

You use the **Add** method of the **CommandBars** collection to add a new control to a toolbar. To add a built-in control, you specify the ID number of that control in the *ID* argument of the **Add** method. This adds the **Spelling** control to the "Quick Tools" toolbar:

```
Set mySpell = CommandBars("Quick Tools").Controls.Add(Id:=2)
```

For information about determining the built-in command ID numbers of an Office application, see "*Menu Item and Toolbar Control Ids*" later in this section.

To add a custom control, you add a new control and then set the **OnAction** property specifying a VBA procedure. Set the *type* argument of the **Add** method to **msoControlButton** to indicate that the new control is a button, and set the **FaceId** value of the control to the ID of a built-in control whose face you want to copy. The following Excel example adds a button before the **Save** button on the **Standard** toolbar. The procedure "macro1" is executed by the button, and the image is set to that of a grid (ID 987).

```
Set xx = CommandBars("Standard").Controls.Add(Type:=msoControlButton, _
  Before:=3)
xx.OnAction:="macro1"
xx.FaceId = 987
```

There are many properties of the objects that represent toolbar buttons that you can set in VBA to modify the control's appearance - for more information, see "Style

Property" and "FaceID" property in on-line Help, as well as the Help topics for other properties and methods of the **CommandBarButton** object.

To set a control to begin a group of controls (i.e. to be preceded by a line) just set the **BeginGroup** property of the object to True. Use **Controls**(*index*), where index is the caption or index number of a control, to return an object that represents the control.

Adding and Initialising Text Box, List Box and Combo Box Controls

You must use VBA for this purpose. Use the **Add** method of the **CommandBarControls** collection to add one of these components - you specify exactly which one you want to add using the *type* argument:

To add this control:	Specify this type:
Text box	msoControlEdit
List box	msoControlDropDown
Combo box	msoControlComboBox

Use the **Style** property of the component to indicate whether the caption should appear to the left of the box itself.

This example adds a combo box with the label "Colour" to a custom toolbar and assigns a macro named "GetColour" to it:

```
Set newCombo=CommandBars("Custom1").Controls.Add(Type:=msoControlComboBox)
With newCombo
   .AddItem "Red"
   .AddItem "Yellow"
   .AddItem "Blue"
   .AddItem "Green"
   .Style= msoComboNormal
   .OnAction="GetColour"
End With
```

While your VBA application is running, the procedure assigned to the **OnAction** property of the combo box control is called each time the user changes the control. In the procedure you can use the **ActionControl** property of the **CommandBars** object to find out which control was changed and to return the changed value. The **ListIndex** property will return the item that was entered into the combo box.

Deleting Toolbar Controls

In VBA, you use the **Delete** method to delete a custom toolbar or built-in toolbar control. You cannot delete a built-in toolbar.

```
CommandBars("Standard").Controls("Print").Delete
CommandBars("Custom Bar").Delete
```

Restoring Built-in ToolBar Controls

You can restore built-in (but not custom) toolbar controls you have deleted. Use the **Reset** method, e.g.

```
CommandBars("Standard").Reset
```

Run-Time Modifications to Toolbars

You can program the toolbars you create at design time to respond dynamically to changing conditions at run time. If a particular control is an inappropriate choice in certain contexts, you can remove it or disable it. If a control has two possible states, you can make the control appear pushed down to show it is turned on or appear flat to show that it is turned off. Obviously, to make run-time changes you must use VBA.

Displaying or Hiding Toolbars and Toolbar Controls

A toolbar takes up screen space that could otherwise be used to display data and you might choose to hide any non-essential toolbars and/or controls. Visibility is determined by the **Visible** property. The following procedure toggles the visibility of the toolbar every time the user clicks the menu item:

```
Sub ViewToolBar()
   With CommandBars("Worksheet Menu Bar").Controls("View").Controls("x")
      If .State = msoButtonUp Then
         .State = msoButtonDown
         CommandBars("TargetToolBar").Visible = True
      Else
         .State = msoButtonUp
         CommandBars("targetToolBar").Visible = False
      End If
   End With
End Sub
```

Restoring a Built-in Toolbar

The **Reset** method is used for this purpose - the following example resets all the toolbars to their default state and deletes all custom toolbars:

```
For Each cb in CommandBars
    If cb.BuiltIn Then
      cb.Reset
    Else
      cb.Delete
    End If
Next
```

NB Note that not only does **Reset** restore the built-in toolbars it also returns them to their original state, so any custom buttons you may have added will be lost!

Enabling or Disabling Toolbar Controls

You may want to control the availability of a toolbar control while your application is running. If you disable a control it beeps when clicked and does not run its associates procedure. Use the **Enabled** property as follows:

```
CommandBars("Standard").Controls(3).Enabled = False
```

Indicating the State of Toolbar Buttons

If appropriate, you can make a button appear to be pushed down - use the **State** property and toggle its value between **msoButtonDown** and **msoButtonUp**.

Modifying Text Box, List Box and Combo Box Controls

Run-time changes to these controls can be made, using the **Text** property and the **AddItem** and **RemoveItem** (with an index number) methods as necessary.

Menu Item and Toolbar Control IDs

Each Office application contains a unique set of menu bars and toolbars, and a unique set of available menu items and toolbar controls. Note that only a sub-set of these actually appears on any application's built-in menu and tool bars.

Although the functionality associated with each built-in item belongs to a specific Office application, the caption, button image, size and other default properties are stored in one shared resource. You use ID numbers to find specific menu items and toolbar controls in this resource.

Even though this shared resource contains information about the menu items and toolbar controls in all Office applications, you can only add items whose functionality is available in the target application. These is no restriction on using the images of the toolbar controls, however.

To determine the IDs of the built-in menu items and toolbar controls in a specific Office application, you can do any of the following:

- In a module, write code to assign a menu item or toolbar control that already appears on a menu or toolbar to an object variable, and then use debugging tools to inspect the object's ID value.
- Run the following procedure to create a text document that lists the Ids and captions of all the built-in commands in that application:

```
Sub OutputIDs()
  Const maxid = 4000
  Open "c:\ids.txt" for Output as #1
  'create a temporary command bar with every available item
  Set cbr = CommandBars.Add("temporary", msoBarTop, False, True)
  For i = 1 to maxid
    On Error Resume Next
    Cbr.Controls.Add Id:=i
  Next
  On Error Goto 0
  'Write the ID and caption of each control to the output file
  For Each btn in cbr.Controls
    Write #1,btn.Id, btn.Caption
  Next
  'Delete the command Bar and close the output file
  cbr.Delete
  Close #1
End Sub
```

- Run the following procedure in one of the Office applications to create a set of custom toolbars that contain as many buttons as there are valid **FaceId** property values in Office 97/2000. Each button's image and ToolTip text is set to one of those values. You can cross-reference the ID of a built-in command to the **FaceId** property value of a button on one of these toolbars, and vice versa.

```
Sub MakeAllFaceIDs()
  'make 14 toolbars, with 300 faces each
  'note that maxid is greater than last valid ID, so error will
  'occur when first invalid ID is used
  Const maxId = 3900
```

```
  On Error GoTo realMax
  For bars = 0 to 13
    firstId = bars * 300
    lastId = firstId + 299
    Set tb = CommandBars.Add
    For i = firstId to lastId
       Set btn = tb.Controls.Add
       btn.FaceId = i
       btn.TooltipText = "Faceid = " & i
    Next
    tb.Name = ("Faces " & CStr(firstId) & " to " & Cstr(lastId))
    tb.width = 591
    tb.Visible = True
  Next
  'Delete the button that caused the error and set toolbar name
  realMax:
  btn.Delete
  tb.Name = ("Faces " & CStr(firstId) & " to " & Cstr(i - 1))
  tb.width = 591
  tb.Visible = True
End Sub
```

Note: the Ids of the pop-up controls for built-in menus are in the range 30002 to 30426. Remember that these Ids return empty copies of the built-in menus.

10. WORD OBJECTS

We have so far concentrated on illustrating the use of VBA by means of Excel. Most of the techniques apply equally to Word, but there are obviously differences, related to the different object model if that application. In this section, we look at the features unique to Word.

In previous releases of Office, the macro language for Word was WordBasic, so it may be of interest for those of you who may be familiar with this language to discover the main differences between WordBasic and VBA as applied to Word.

Conceptual Differences between VBA and WordBasic

The primary difference between Visual Basic for Applications and WordBasic is that whereas the WordBasic language consists of a flat list of approximately 900 commands, Visual Basic consists of a hierarchy of objects, each of which exposes a specific set of methods and properties (similar to statements and functions in WordBasic). While most WordBasic commands can be run at any time, Visual Basic only exposes the methods and properties of the available objects at a given time.

Objects are the fundamental building block of Visual Basic; almost everything you do in Visual Basic involves modifying objects. Every element of Word - documents, paragraphs, fields, bookmarks, and so on - can be represented by an object in Visual Basic. Unlike commands in a flat list, there are objects that can only be accessed from other objects. For example, the **Font** object can be accessed from various objects including the **Style**, **Selection**, and **Find** object.

The programming task of applying bold formatting demonstrates the differences between the two programming languages. The following WordBasic instruction applies bold formatting to the selection.

```
Bold 1
```

The following example is the VBA equivalent for applying bold formatting to the selection.

```
Selection.Font.Bold = True
```

VBA doesn't include a **Bold** statement and function. Instead, there's a property named **Bold**. (A property is usually an attribute of an object, such as its size, its colour, or whether or not it's bold.) **Bold** is a property of the **Font** object. Likewise, **Font** is a property of the **Selection** object that returns a **Font** object. Following the object hierarchy, you can build the instruction to apply bold formatting to the selection.

The **Bold** property is a read/write Boolean property. This means that the **Bold** property can be set to **True** or **False** (on or off), or the current value can be returned. The following WordBasic instruction returns a value indicating whether bold formatting is applied to the selection.

```
x = Bold()
```

The following example is the VBA equivalent for returning the bold formatting status from the selection.

```
x = Selection.Font.Bold
```

The Visual Basic thought process

To perform a task in VBA, you need to determine the appropriate object. For example, if you want to apply character formatting found in the **Font** dialog box, use the **Font** object. Then you need to determine how to "drill down" through the Word object hierarchy from the **Application** object to the **Font** object, through the objects that contain the **Font** object you want to modify. After you have determined the path to your object (for example, **Selection.Font**), use the Object Browser, Help, or the

features such as Auto List Members in the VBA Editor to determine what properties and methods can be applied to the object. For more information about drilling down to objects using properties and methods, see Understanding objects, properties, and methods.

Properties and methods are often available to multiple objects in the Word object hierarchy. For example, the following instruction applies bold formatting to the entire document.

```
ActiveDocument.Content.Bold = True
```

Also, objects themselves often exists in more than one place in the object hierarchy. For an illustration of the Word object model, see Microsoft Word Objects.

If you know the WordBasic command for the task you want to accomplish in Word 97/2000, see *Visual Basic Equivalents for WordBasic Commands* in on-line help.

The Selection and Range objects

Most WordBasic commands modify the selection. For example, the **Bold** command formats the selection with bold formatting. The **InsertField** command inserts a field at the insertion point. Anytime you want to work with the selection in VBA, you use the **Selection** property to return the **Selection** object. The selection can be a block of text or just the insertion point. The following VBA example inserts text and a new paragraph after the selection.

```
Selection.InsertAfter Text:="Hello World"
Selection.InsertParagraphAfter
```

In addition to working with the selection, you can define and work with various ranges of text in a document. A **Range** object refers to a contiguous area in a document with a starting character position and ending character position. Similar to the way bookmarks are used in a document, **Range** objects are used in VBA to identify portions of a document. However, unlike a bookmark, a **Range** object is invisible to the user unless the **Range** has been selected using the **Select** method. For example, you can use VBA to apply bold formatting anywhere in the document without changing the selection. The following example applies bold formatting to the first 10 characters in the active document.

```
ActiveDocument.Range(Start:=0, End:=10).Bold = True
```

The following example applies bold formatting to the first paragraph.

```
ActiveDocument.Paragraphs(1).Range.Bold = True
```

Both of these example change the formatting in the active document without changing the selection. For more information on the **Range** object see *Working with Range objects* in on-line help.

Word Objects

In this section we will look at the objects specific to Word. Although there will be a certain amount of overlap with material presented earlier in this booklet, it is worth providing an initial overview of the object model:

Objects and Collections in Word

In this case, an object represents an element of Word, such as a document, a paragraph, a bookmark, or a single character. A collection is an object that contains several other objects, usually of the same type; for example, all the bookmark objects in a document are contained in a single collection object. Using properties and methods, you can modify a single object or an entire collection of objects.

What is a property?

A property is an attribute of an object or an aspect of its behaviour. For example, properties of a document include its name, its content, and its save status, as well as whether change tracking is turned on. To change the characteristics of an object, you change the values of its properties. To set the value of a property, follow the reference to an object with a period, the property name, an equal sign, and the new property value. The following example turns on change tracking in the document named "MyDoc.doc."

```
Documents("MyDoc.doc").TrackRevisions = True
```

In this example, Documents refers to the collection of open documents, and the name "MyDoc.doc" identifies a single document in the collection. The **TrackRevisions** property is set for that single document.

Some properties cannot be set. The Help topic for a property indicates whether that property can be set (read-write) or can only be read (read-only). You can return information about an object by returning the value of one of its properties. The following example returns the name of the active document.

```
docName = ActiveDocument.Name
```

In this example, **ActiveDocument** refers to the document in the active window in Word. The name of that document is assigned to the variable docName.

Note: The Help topic for each property indicates whether you can set that property (read-write), only read the property (read-only), or only write the property (write-only). Also the Object Browser in the VBA Editor displays the read-write status at the bottom of the browser window when the property is selected.

What is a method?

A method is an action that an object can perform. For example, just as a document can be printed, the Document object has a **PrintOut** method. Methods often have arguments that qualify how the action is performed. The following example prints the first three pages of the active document.

```
ActiveDocument.PrintOut From:=1, To:=3
```

In most cases, methods are actions and properties are qualities. Using a method causes something to happen to an object, while using a property returns information about the object or it causes a quality about the object to change.

Returning an object

Most objects are returned via a single object from the collection. For example, the **Documents** collection contains the open Word documents. You use the **Documents** property of the **Application** object (the object at the top of the Word object hierarchy) to return the **Documents** collection. After you've accessed the collection, you can return a single object by using an index value in parentheses (this is similar to how you work with arrays). The index value is usually a number or a name. For more information, see *Returning an Object from a Collection* in on-line help.

The following example uses the **Documents** property to access the **Document** collection. The index number is used to return the first document in the **Documents** collection. The **Close** method is then applied to the **Document** object to close the first document in the **Documents** collection.

```
Documents(1).Close
```

The following example uses a name (specified as a string) to identify a **Document** object within the **Documents** collection.

```
Documents("Sales.doc").Close
```

Collection objects often have methods and properties that you can use to modify the entire collection of objects. The **Documents** object has a **Save** method that saves all the documents in the collection. The following example saves the open documents by applying the **Save** method.

```
Documents.Save
```

The **Document** object also has a **Save** method available for saving a single document. The following example saves the document named Report.doc.

```
Documents("Report.doc").Save
```

To return an object that is further down in the Word object hierarchy, you must "drill down" to it by using properties and methods to return objects. To see how this is done, open the VBA Editor and click **Object Browser** on the **View** menu. Click **Application** in the **Classes** list on the left, then click **ActiveDocument** from the list of members on the right. The text at bottom of the **Object Browser** indicates that **ActiveDocument** is a read-only property that returns a **Document** object. Click **Document** at the bottom of the **Object Browser**; the **Document** object is automatically selected in the **Classes** list, and the **Members** list displays the members of the **Document** object. Scroll through the list of members until you find **Close**. Click the **Close** method. The text at the bottom of the **Object Browser** window shows the syntax for the method. For more information about the method, press F1 or click the Help button to jump to the **Close** method Help topic.

Given this information, you can write the following instruction to close the active document.

```
ActiveDocument.Close SaveChanges:=wdSaveChanges
```

The following example maximises the active document window.

```
ActiveWindow.WindowState = wdWindowStateMaximize
```

The **ActiveWindow** property returns a **Window** object that represents the active window. The **WindowState** property is set to the **maximize** constant (**wdWindowStateMaximize**).

The following example creates a new document and displays the **Save As** dialog box so that a name can be provided for the document.

```
Documents.Add.Save
```

The **Documents** property returns the **Documents** collection. The **Add** method creates a new document and returns a **Document** object. The **Save** method is then applied to the **Document** object.

As you can see, you use methods or properties to drill down to an object. That is, you return an object by applying a method or property to an object above it in the object hierarchy. After you return the object you want, you can apply the methods and control the properties of that object. To review the hierarchy of objects, *see Microsoft Word Objects* in on-line help.

Working with the Application Object

When you start a Word session you automatically create an **Application** object. You can use this object to control many of the top level properties of Word. Properties of the **Application** object also provide access to lower level objects, such as the **Documents** collection.

Working with the Document Object

When you open or create a file in Word, you create a **Document** object. You use properties and methods of the **Document** object to open, create, save, activate and close files.

To open a document, you make use of the **Documents** collection as follows:

```
Documents.Open FileName:="C:\docs\xxx.doc"
```

You can set a variable to point to any open document in this way:

```
Set thisDoc = Documents("xxx.doc")
```

If you want to use a numerical index, you specify the position of the document in the **Documents** collection, e.g.

```
Set thisDoc = Documents(1)
```

.. if it is the first.

Alternatively, you can use the **ActiveDocument** property to refer to the document with the focus. The following example displays the name of the active document, provided there is one:

```
If Documents.Count > 0 Then
      MsgBox ActiveDocument.Name
Else
      MsgBox "No documents are open"
End if
```

Opening Documents

To open an existing document, use the **Open** method as seen above. It is important to specify the correct path however, or Word will give an error. The following example looks in the default documents directory for the specified file:

```
DefaultDir=Options.DefaultFilePath(wdDocumentsPath)
With Application.FileSearch
  .FileName="Test.doc"
  .LookIn = defaultDir
  .Execute
  If .FoundFile.Count = 1 Then
      Documents.Open FileName:=defaultDir & "\test.doc"
  Else
      Msgbox "Test.doc not found"
  End If
End With
```

Instead of hard-coding the *FileName* argument of the **Open** method, you may want a user to select a file to open. Use the **Dialogs** property with the **wdDialogFileOpen** constant to return a reference to the built-in FileOpen dialog box. The **Show** method displays and executes actions performed in the **Open** dialog:

```
Dialogs(wdDialogFileOpen).Show
```

Creating and Saving Documents

Use the **Add** method of the **Documents** collection - e.g. `Documents.Add`

You can also define a variable to refer to this new document :

```
Set newdoc = Documents.Add
```

To save a document for the first time, use the **SaveAs** method, otherwise use **Save**.

```
Documents("xxx.doc").Save
```

If you use the **Save** method on an unsaved document, the **SaveAs** dialog will appear and prompt you for a filename. To save all open documents, apply the **Save** method

to the **Documents** collection. You can also avoid a prompt for a filename by means of:

```
Documents.Save NoPrompt:=True
```

Activating a Document

To make a different document the active document, apply the **Activate** method to a **Document** object.

```
Set Doc1 = Documents.Open("c:\docs\xx1.doc")
Set Doc2 = Documents.Open("c:\docs\xx2.doc")
Doc2.Activate
```

Printing a Document

Apply the **Printout** method to a **Document** object - e.g. `ActiveDocument.PrintOut`

To set print options you would normally set via File/Print menu, use the arguments of the **PrintOut** method. For options set via the Tools/Options menu (Print tab) you use properties of the **Options** object:

```
Options.PrintHiddenText = True
ActiveDocument.PrintOut Range:=wdPrintFromTo, From:="1", To:="3"
```

Closing Documents

Apply the **Close** method to a **Document** object - e.g. `Documents("xx.doc").Close`

If there are changes in the document (the document is "dirty") Word displays a message asking if you want to save the changes. You can choose whether or not to display this prompt by using the **wdDoNotSaveChanges** or **wdSaveChanges** constant with the *SaveChanges* argument:

```
Documents("xx.doc").Close SaveChanges:=wdDoNotSaveChanges
```

To close all open documents apply **Close** to the **Documents** collection:

```
Documents.Close SaveChanges:=wdDoNotSaveChanges
```

Accessing Objects in a Document

From the **Document** object you have access to properties and methods that return objects contained within. For example, you can access the **Tables** collection:

```
MsgBox ActiveDocument.Tables.Count & " tables in this document"
Set myTable = Documents("xx.doc").Tables(1)
```

Adding Objects to a Document

You can add object using the **Add** method with a collection objects accessed from the **Document** object. For example, the following code adds a 3x3 table at the location specified by the **Range** variable `myRange` :

```
ActiveDocument.Tables.Add Range:=myRange, NumRows:=3, NumColumns:=3
```

For a list of collection objects that support the **Add** method, see "Add Method" in on-line help.

Working With the Range Object

A common task when using Visual Basic is to specify an area in a document and then do something with it, such as insert text or apply formatting. For example, you may want to write a macro that locates a word or phrase within a portion of a document. You can use a **Range** object to represent the portion of the document can be represented by a **Range** object. After the **Range** object is identified, methods and properties of the **Range** object can be applied in order to modify the contents of the range.

The **Range** object represents a contiguous area in a document. Each **Range** object is defined by a starting and ending character position. Similar to the way bookmarks are used in a document, **Range** objects are used in VBA procedures to identify specific portions of a document, however, unlike a bookmark, a **Range** object only exists while the procedure that defined it is running.

Note: **Range** objects are independent of the selection. That is, you can define and manipulate a range without changing the selection. You can also define multiple ranges in a document, while there can be only one selection per pane.

The **Start**, **End** and **StoryType** properties uniquely identify a Range object. The Start and End properties return or set the starting and ending character positions of the Range object. The character position at the beginning of the document is zero, the position after the first character is one, and so on. There are 11 different story types represented by the **WdStoryType** constants of the **StoryType** property.

Using the Range Object Instead of the Selection Object

The macro recorder will often create a macro that uses the **Selection** property to manipulate the **Selection** object, however you can usually accomplish the same task with fewer instructions using one or more **Range** objects. The following example was recorded using the macro recorder - it applies bold formatting to the first two words in the document:

```
Selection.Homekey Unit:=wdStory
Selection.MoveRight Unit:=wdWord, Count:=2, Extend:=wdExtend
Selection.Font.Bold = wdToggle
```

The following code does this using the Range object:

```
ActiveDocument.Range(Start:=0, _
End:=ActiveDocument.Words(2).End).Bold = True
```

The following example applies bold formatting to the first two words in the document and then inserts a new paragraph:

```
Selection.Homekey Unit:=wdStory
Selection.MoveRight Unit:=wdWord, Count:=2, Extend:=wdExtend
Selection.Font.Bold = wdToggle
Selection.MoveRight Unit:=wdCharacter, Count:=1
Selection.TypeParagraph
```

and here is the equivalent using the Range object:

```
Set myRange=ActiveDocument.Range(Start:=0, _
End:=ActiveDocument.Words(2).End)
MyRange.Bold=True
MyRange.InsertParagraphAfter
```

Both of the preceding examples change the formatting in the active document without changing the selection. In most cases, **Range** objects are preferred over **Selection** objects for the following reasons:

- You can define and use multiple **Range** objects, whereas you can only have one **Selection** object per document window.

- Manipulating **Range** objects doesn't change the selected text
- Manipulating **Range** objects is faster than working with a selection

Using the Range Method to Return a Range Object

Use the **Range** method to return a **Range** object defined by the given starting and ending character positions. This returns a **Range** object located in the main story. The following example returns a Range object that refers to the first 10 characters in the active document:

```
Set myRange = ActiveDocument.Range(Start:=0, End:=10)
```

MyRange refers to the first ten characters in the active document. You can see that the **Range** object has been created when you apply a property or method to the **Range** object stored in the MyRange variable. The following example applies bold formatting to the first ten characters in the active document.

```
Set myRange = ActiveDocument.Range(Start:=0, End:=10)
myRange.Bold = True
```

When you need to refer to a **Range** object multiple times, you can use the **Set** statement to set a variable equal to the **Range** object. However, if you only need to perform a single action on a **Range** object, there's no need to store the object in a variable. The same results can be achieved using just one instruction that identifies the range and changes the **Bold** property.

```
ActiveDocument.Range(Start:=0, End:=10).Bold = True
```

Like a bookmark, a range can span a group of characters or mark a location in a document. The **Range** object in the following example has the same starting and ending points. The range does not include any text. The following example inserts text at the beginning of the active document.

```
Set myRange = ActiveDocument.Range(Start:=0, End:=0)
myRange.InsertBefore "Hello "
```

You can define the beginning and end points of a range using the character position numbers as shown above, or use the **Start** and **End** properties with objects such as **Selection**, **Bookmark**, or **Range**. The following example creates a **Range** object beginning at the start of the second paragraph and ending after the third paragraph.

```
Set myDoc = ActiveDocument
Set myRange = myDoc.Range(Start:=myDoc.Paragraphs(2).Range.Start, _
    End:=myDoc.Paragraphs(3).Range.End)
```

For additional information and examples, see the *Range* method in on-line Help.

Using the Range Property to Return a Range Object

Use the **Range** property to return a **Range** object defined by the beginning and end of another object. The **Range** property applies to many objects (for example, Paragraph, Bookmark, and Cell). The following example returns a **Range** object that refers to the first paragraph in the active document.

```
Set aRange = ActiveDocument.Paragraphs(1).Range
```

The following example returns a **Range** object that refers to the second through fourth paragraphs in the active document

```
Set aRange = ActiveDocument.Range(Start:=
ActiveDocument.Paragraphs(2).Range.Start, _
    End:=ActiveDocument.Paragraphs(4).Range.End)
```

The **Range** property appears on multiple objects, such as **Paragraph**, **Bookmark**, and **Cell**, and is used to return a **Range** object. After you have a **Range** object, you can use any of its properties or methods to modify the range. The following example copies the first paragraph in the active document.

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

If you need to apply numerous properties or methods to the same Range object, you can use the With…End With structure. The following example formats the text in the first paragraph of the active document.

```
Set myRange = ActiveDocument.Paragraphs(1).Range
With myRange
    .Bold = True
    .ParagraphFormat.Alignment = wdAlignParagraphCenter
    .Font.Name = "Arial"
End With
```

Modifying Part of a Document

VBA includes these objects you use to modify the corresponding document elements:

This expression ..	Returns this object ..
Words(*index*),	Range
Characters(*index*	Range
Sentences(*index*),	Range
Paragraphs(*index*)	Paragraph
Sections(*index*).	Section

When you use these properties without an index, a collection with the same name is returned - for example, the **Paragraphs** property returns the **Paragraphs** collection. However, if you identify an item within a collection by index, the object in the second column of the preceding table is returned - for example Words(1) returns a **Range** object. You can use any of the range properties or methods to modify a **Range** object, as in this example, which copies the first word in the selection to the clipboard:

```
Selection.Words(1).Copy
```

Other examples include -

- copying the first paragraph in the active document to the clipboard

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

- setting the case of the first word in the active document

```
ActiveDocument.Words(1).case = wdUpperCase
```

- setting the bottom margin of the first selected section to 0.5 inches.

```
Selection.Sections(1).PageSetup.BottomMargin = InchesToPoints(0.5)
```

- double-spacing the text in the active document. The **Content** property returns a **Range** object that represents the main document story.

```
ActiveDocument.Content.ParagraphFormat.Space2
```

Modifying a Group of Elements

To modify a range of text that consists of a group of document elements, you can create a **Range** object that includes them. Using the **Start** and **End** properties with a **Range** object, you can create a new **Range** object that refers to a group of document elements. The following example creates a **Range** object that refers to the first three words in the active document and changes their font name to Arial.

```
Set Doc = ActiveDocument
Set myRange = Doc.Range(Start:=Doc.Words(1).Start, End:=Doc.Words(3).End)
MyRange.Font.Name = "Arial"
```

The following example creates a **Range** object beginning at the start of paragraph 2 and ending after paragraph 3. The **ParagraphFormat** property is then used to access paragraph formatting properties:

```
Set Doc = ActiveDocument
Set myRange = Doc.Range(Start:Doc.Paragraphs(2).Range.Start, _
   End:=Doc.Paragraphs(3).Range.End)
With myRange.ParagraphFormat
  .Space1
  .SpaceAfter = 6
  .SpaceBefore = 6
End With
```

Returning or Setting the Text in a Range

Use the **Text** property to return or set the contents of a **Range** object. This returns the first word in the active document:

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

This changes the first word in the active document to "Hello"

```
ActiveDocument.Words(1).Text = "Hello"
```

Use the **InsertAfter** or **InsertBefore** methods to insert text before or after a range. The following example inserts text at the beginning of the second paragraph in the active document:

```
ActiveDocument.Paragraphs(2).Range.InsertBefore Text:="In the beginning .."
```

After using the **InsertAfter** or **InsertBefore** methods the range expands to include the new text. If you do not want to do this, use the **Collapse** method afterwards:

```
With ActiveDocument.Paragraphs(2).Range
    .InsertBefore Text:="Hello"
    .Collapse Direction:=wdCollapseStart
End With
```

Formatting the Text in a Range

Use the **Font** property to get character -formatting properties and methods, and the **ParagraphFormat** property to get to paragraph-formatting properties and methods. For example:

```
With ActiveDocument.Paragraphs(1).Range.Font
    .Name = "Times New Roman"
    .Size = 14
    .AllCaps = True
End With
With ActiveDocument.Paragraphs(1).Range.ParagraphFormat
    .LeftIndent = InchesToPoints(0.5)
    .Space1
End With
```

Redefining a Range Object

Use the SetRange method to redefine an existing **Range** object. The following example defines myRange to the current selection. The **SetRange** method redefines myRange so that it refers to current selection plus the next ten characters.

```
Set myRange = Selection.Range
myRange.SetRange Start:=myRange.Start, End:=myRange.End + 10
```

You can also redefine a **Range** object by changing the values of **Start** and **End**, or by using the **MoveStart** or **MoveEnd** methods. For example:

```
Set myRange = Selection.Range
MyRange.End = myRange.End + 10

Set myRange = ActiveDocument.Paragraphs(2)
MyRange.MoveEnd Unit:=wdParagraph, Count:=1
```

Looping Through a Range of Paragraphs

There are two main ways of doing this:

Using the For Each .. Next Statement

This example loops through the first 5 paragraphs in the active document, adding text before each of them:

```
Set myDoc = ActiveDocument
Set myRange = myDoc.Range(Start:=myDoc.Paragraphs(1).Range.Start, _
  End:=myDoc.Paragraphs(5).Range.End)
For Each para in myRange.Paragraphs
    Para.Range.InsertBefore "Question: " & vbTab
Next para
```

Using the Next Property or Method

The **Next** method redefines a range to refer to the next item or object in the class:

```
Set myRange = myRange.Paragraphs(1).Next.Range
```

Working With Stories

A story is a document area that contains a range of text distinct from other areas of text in a document. For example, if a document includes body text, footnotes, and headers, it contains a main text story, footnotes story, and headers story.

There are 11 different types of stories that can be part of a document, corresponding to the following **WdStoryType** constants: **wdCommentsStory**, **wdEndnotesStory**, **wdEvenPagesFooterStory**, **wdEvenPagesHeaderStory**, **wdFirstPageFooterStory**, **wdFirstPageHeaderStory**, **wdFootnotesStory**, **wdMainTextStory**, **wdPrimaryFooterStory**, **wdPrimaryHeaderStory**, and **wdTextFrameStory**. The **StoryRanges** collection contains the first story for each story type available in a document. Use the **NextStoryRange** method to return subsequent stories.

Use the **StoryType** property to return the story type for the specified range, selection or bookmark. The following example closes the footnote pane in the active window if the selection is contained in the footnote story:

```
ActiveWindow.View.Type = wdNormalView
If Selection.StoryType = wdFootNotesStory Then _
   ActiveWindow.ActivePane.Close
```

Working With the Selection Object

When you work on a document in Word, you usually select text and then perform an action, such as formatting the text or typing text. In VBA, it is usually not necessary to select text before modifying the text. Instead, you create a **Range** object that refers to a specific portion of the document. However, when you want your code to respond to or change the selection, you can do so with the **Selection** object.

The **Select** method activates an object. For example, the following instruction selects the first word in the active document.

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

For more Select method examples, see the **Select** method or "Selecting text in a document" in Online Help.

The **Selection** property returns a **Selection** object that represents the active selection in a document window pane. There can only be one **Selection** object per document window pane and only one **Selection** object can be active. For example, the following example changes the paragraph formatting of the paragraphs in the selection.

```
Selection.Paragraphs.LeftIndent = InchesToPoints(0.5)
```

The following example inserts the word "Hello" after the selection.

```
Selection.InsertAfter Text:="Hello"
```

The following example applies bold formatting to the selected text.

```
Selection.Font.Bold = True
```

The macro recorder will often create a macro that uses the **Selection** property. The following macro, created using the macro recorder, applies bold formatting to the first two words in the document.

```
Selection.HomeKey Unit:=wdStory
Selection.MoveRight Unit:=wdWord, Count:=2, Extend:=wdExtend
Selection.Font.Bold = wdToggle
```

The following example accomplishes the same task without using the Selection property.

```
ActiveDocument.Range(Start:=0,
End:=ActiveDocument.Words(2).End).Bold = True
```

Moving and Extending the Selection

The **Selection** object also includes various methods you can use to expand or move an existing selection. For example, the **MoveDown** method has an **Extend** argument that you can set to **wdExtend**. The following example selects the next three paragraphs in the active window.

```
With Selection
    .StartOf Unit:=wdParagraph, Extend:=wdMove
    .MoveDown Unit:=wdParagraph, Count:=3, Extend:=wdExtend
End With
```

Or

```
Selection.MoveDown Unit:=wdParagraph, Count:=1, Extend:=wdMove
```

The next example extends the selection by moving the end position to the end of the paragraph:

```
Selection.MoveEnd Unit:=wdParagraph, Count:=1
```

Because there can be only one selection in a document window or pane, you can also move the selection by selecting another object. You can also move the selection by

using the **GoToNext**, **GoToPrevious**, or **GoTo** method. The next example moves the selection to the fourth line in the document:

```
Selection.GoTo What:=wdGoToLine, Which:=wdGoToAbsolute, Count:=4
```

Properties and Methods of the Selection Object

This section highlights some commonly used methods and properties of the **Selection** object.

Returning or Setting the Text in the Selection

Use the **Text** property:

```
strText = Selection.Text
Selection.Text = "Hello World"
Selection.InsertBefore Text:="This is some text ..."
```

Formatting the Selected Text

Use the **Font** property:

```
With Selection.Font
      .Name = "Times New Roman"
      .Size = 14
End With
```

Returning a Range Object

If a method or property is available from the **Range** object but not from the **Selection** object, use the **Range** property :-

```
Selection.Range.CheckSpelling
```

Returning Information About the Selection

Use the **Information** property. There are 35 different constants that you can use to return different types of information about the selection. If the selection is in a table, for instance, the following example displays the number of rows and columns in the table:

```
If Selection.Information(wdWithinTable) = True Then
  MsgBox "Columns = " & Selection.Information(wdMaximumNumberOfColumns) _
    & vbCr & "Rows = " & selection.Information(wdMaximumNumberOfRows)
End If
```

The table below gives a complete list of the constants you can use with the **Information** property:

Constant	Function
wdActiveEndAdjustedPageNumber	Returns the number of the page that contains the active end of the specified selection or range. If you set a starting page number or make other manual adjustments, returns the adjusted page number (unlike wdActiveEndPageNumber)
wdActiveEndPageNumber	Returns the number of the page that contains the active end of the specified selection or range, counting from the beginning of the document. Any manual adjustments to page numbering are disregarded (unlike wdActiveEndAdjustedPageNumber).
wdActiveEndSectionNumber	Returns the number of the section that contains the active end of the specified selection or range.
wdAtEndOfRowMarker	Returns True if the specified selection or range is at the end-of-row mark in a table.
wdCapsLock	Returns True if Caps Lock is in effect.
wdEndOfRangeColumnNumber	Returns the table column number that contains the end of the specified selection or range.

wdEndOfRangeRowNumber	Returns the table row number that contains the end of the specified selection or range.
wdFirstCharacterColumnNumber	Returns the character position of the first character in the specified selection or range. If the selection or range is collapsed, the character number immediately to the right of the range or selection is returned (this is the same as the character column number displayed in the status bar after "Col")
wdFirstCharacterLineNumber	Returns the line number of the first character in the selection. If the Pagination property is False or the Draft property is True, returns –1.
wdFrameIsSelected	Returns True if the selection or range is an entire frame or text box
wdHeaderFooterType	Returns a value that indicates the type of header or footer that contains the specified selection or range, as shown in the following table: **Value** **Type of header or footer** –1 None (the selection or range isn't in a header or footer) 0 (zero) Even page header 1 Odd page header (or the only header, if there aren't odd and even headers) 2 Even page footer 3 Odd page footer (or the only footer, if there aren't odd and even footers) 4 First page header 5 First page footer
wdHorizontalPositionRelativeToPage	Returns the horizontal position of the specified selection or range; this is the distance from the left edge of the selection or range to the left edge of the page, in twips (20 twips = 1 point, 72 points = 1 inch). If the selection or range isn't within the screen area, returns –1
wdHorizontalPositionRelativeToTextBoundary	Returns the horizontal position of the specified selection or range, relative to the left edge of the nearest text boundary enclosing it, in twips (20 twips = 1 point, 72 points = 1 inch). If the selection or range isn't within the screen area, returns –1
wdInClipboard	Returns True if the specified selection or range is on the Macintosh Clipboard
wdInCommentPane	Returns True if the specified selection or range is in a comment pane
wdInEndnote	Returns True if the specified selection or range is in an endnote area in page layout view or in the endnote pane in normal view
wdInFootnote	Returns True if the specified selection or range is in a footnote area in page layout view or in the footnote pane in normal view
wdInFootnoteEndnotePane	Returns True if the specified selection or range is in the footnote or endnote pane in normal view or in a footnote or endnote area in page layout view. For more information, see the descriptions of wdInFootnote and wdInEndnote in the preceding paragraphs
wdInHeaderFooter	Returns True if the selection or range is in the header or footer pane or in a header or footer in page layout view
wdInMasterDocument	Returns True if the selection or range is in a master document (that is, a document that contains at least one subdocument)
wdInWordMail	Returns a value that indicates the WordMail location of the selection or range, as shown in the following table **Value** **WordMail Location** 0(zero) The selection or range isn't in a WordMail message. 1 The selection or range is in a WordMail send note. 2 The selection or range is in a WordMail read note.
wdMaximumNumberOfColumns	Returns the greatest number of table columns within any row in the selection or range.
wdMaximumNumberOfRows	Returns the greatest number of table rows within the table in the specified selection or range
wdNumberOfPagesInDocument	Returns the number of pages in the document associated with the selection or range.
wdNumLock	Returns True if NumLock is in effect
wdOverType	Returns True if overtype mode is in effect. The Overtype property can be used to change the state of overtype mode
wdReferenceOfType	Returns a value that indicates where the selection is in relation to a

	footnote, endnote, or comment reference, as shown in the following table **Value** **Description** – 1 The selection or range includes but isn't limited to a footnote, endnote, or comment reference. 0 (zero) The selection or range isn't before a footnote, endnote, or comment reference. 1 The selection or range is before a footnote reference. 2 The selection or range is before an endnote reference. 3 The selection or range is before a comment reference.
wdRevisionMarking	Returns True if change tracking is in effect
wdSelectionMode	Returns a value that indicates the current selection mode, as shown in the following table **Value** **Selection mode** 0 (zero) Normal selection 1 Extended selection ("EXT" appears on the status bar) 2 Column selection. ("COL" appears on the status bar)
wdStartOfRangeColumnNumber	Returns the table column number that contains the beginning of the selection or range
wdStartOfRangeRowNumber	Returns the table row number that contains the beginning of the selection or range.
wdVerticalPositionRelativeToPage	Returns the vertical position of the selection or range; this is the distance from the top edge of the selection to the top edge of the page, in twips (20 twips = 1 point, 72 points = 1 inch). If the selection isn't visible in the document window, returns – 1
wdVerticalPositionRelativeToTextBoundary	Returns the vertical position of the selection or range, relative to the top edge of the nearest text boundary enclosing it, in twips (20 twips = 1 point, 72 points = 1 inch). This is useful for determining the position of the insertion point within a frame or table cell. If the selection isn't visible, returns – 1
wdWithInTable	Returns True if the selection is in a table
wdZoomPercentage	Returns the current percentage of magnification as set by the Percentage property

Determining Whether Text is Selected

Use the **Type** property to set or return the way you want the selection to be indicated in your document. For instance you can use the **wdSelectionBlock** constant to determine whether a block of text is selected. The following example selects the paragraph that contains the insertion point if the selection is an insertion point:

```
If Selection.Type = wdSelectionIP Then
    Selection.Paragraphs(1).Range.Select
End If
```

Working With the Find and Replacement Objects

Finding and replacing is exposed by the **Find** and **Replacement** objects. The **Find** object is available from the **Selection** and **Range** object. The find action differs slightly depending upon whether you access the **Find** object from the **Selection** or **Range** object.

Finding text and selecting it

If the **Find** object is accessed from the **Selection** object, the selection is changed when the find criterion is found. The following example selects the next occurrence of the word "Hello." If the end of the document is reached before the word "Hello" is found, the search is stopped.

```
With Selection.Find
    .Forward = True
    .Wrap = wdFindStop
    .Text = "Hello"
```

```
    .Execute
End With
```

The **Find** object includes properties that relate to the options in the **Find and Replace** dialog box (choose **Find** from the **Edit** menu). You can set the individual properties of the **Find** object or use arguments with the **Execute** method as shown in the following example.

```
Selection.Find.Execute FindText:="Hello", Forward:=True, Wrap:=wdFindStop
```

Finding text without changing the selection

If the **Find** object is accessed from a **Range** object, the selection is not changed but the **Range** is redefined when the find criterion is found. The following example locates the first occurrence of the word "blue" in the active document. If the find operation is successful, the range is redefined and bold formatting is applied to the word "blue."

```
With ActiveDocument.Content.Find
    .Text = "blue"
    .Forward = True
    .Execute
    If .Found = True Then .Parent.Bold = True
End With
```

The following example performs the same result as the previous example using arguments of the **Execute** method.

```
Set myRange = ActiveDocument.Content
myRange.Find.Execute FindText:="blue", Forward:=True
If myRange.Find.Found = True Then myRange.Bold = True
```

Using the Replacement object

The **Replacement** object represents the replace criteria for a find and replace operation. The properties and methods of the **Replacement** object correspond to the options in the **Find and Replace** dialog box (**Edit** menu).

The **Replacement** object is available from the **Find** object. The following example replaces all occurrences of the word "hi" with "hello." The selection changes when the find criterion is found because the **Find** object is accessed from the **Selection** object.

```
With Selection.Find
    .ClearFormatting
    .Text = "hi"
    .Replacement.ClearFormatting
    .Replacement.Text = "hello"
    .Execute Replace:=wdReplaceAll, Forward:=True, Wrap:=wdFindContinue
End With
```

The following example removes bold formatting in the active document. The **Bold** property is **True** for the **Find** object and **False** for the **Replacement** object. In order to find and replace formatting, set the find and replace text to empty strings ("") and set the **Format** argument of the **Execute** method to True. The selection remains unchanged because the **Find** object is accessed from a **Range** object (the **Content** property returns a **Range** object).

```
With ActiveDocument.Content.Find
    .ClearFormatting
    .Font.Bold = True
    With .Replacement
        .ClearFormatting
        .Font.Bold = False
    End With
    .Execute FindText:="", ReplaceWith:="", Format:=True,
Replace:=wdReplaceAll
```

```
End With
```

Working with Table, Column, Row and Cell Objects

The Word object model includes and object for tables as well as objects for the various elements of a table. Use the **Tables** property of the **Document**, **Range** or **Selection** object to return the **Tables** collection. The following example converts the first table in the selection to text:

```
If Selection.Tables.Count > 0 Then
    Selection.Tables(1).ConvertToText Separator:=wdSeparateByTabs
End If
```

Use the **Cells** property with the **Column**, **Range**, **Row** or **Selection** object to return the **Cells** collection. E.g.

```
Set myCell = ActiveDocument.Tables(1).Cell(Row:=1, Column:=1)
```

Inserting text into a table cell

The following example inserts text into the first cell of the first table in the active document. The **Cell** method returns a single **Cell** object. The **Range** property returns a **Range** object. The **Delete** method is used to delete the existing text and the **InsertAfter** method inserts the "Cell 1,1" text.

```
If ActiveDocument.Tables.Count >= 1 Then
    With ActiveDocument.Tables(1).Cell(Row:=1, Column:=1).Range
        .Delete
        .InsertAfter Text:="Cell 1,1"
    End With
End If
```

Creating a table, inserting text, and applying formatting

The following example inserts a 4 column, 3 row table at the beginning of the document. The **For Each**...**Next** structure is used to step through each cell in the table. Within the **For Each**...**Next** structure, the **InsertAfter** method is used to add text to the table cells (Cell 1, Cell 2, and so on).

```
Set oDoc = ActiveDocument
Set oTable = oDoc.Tables.Add(Range:=oDoc.Range(Start:=0, End:=0), _
NumRows:=3, NumColumns:=4)
iCount = 1
For Each oCell In oTable.Range.Cells
    oCell.Range.InsertAfter "Cell " & iCount
    iCount = iCount + 1
Next oCell
oTable.AutoFormat Format:=wdTableFormatColorful2, _
    ApplyBorders:=True, ApplyFont:=True, ApplyColor:=True
```

Returning text from a table cell without returning the end of cell marker

The following 2 examples return and display the contents of each cell in the first row of the first document table.

```
Set oTable = ActiveDocument.Tables(1)
For Each aCell In oTable.Rows(1).Cells
    Set myRange = ActiveDocument.Range(Start:=aCell.Range.Start, _
        End:=aCell.Range.End - 1)
    MsgBox myRange.Text
Next aCell
```

```
Set oTable = ActiveDocument.Tables(1)
```

```
For Each aCell In oTable.Rows(1).Cells
    Set myRange = aCell.Range
    myRange.MoveEnd Unit:=wdCharacter, Count:=-1
    MsgBox myRange.Text
Next aCell
```

Converting existing text to a table

The following example inserts tab-delimited text at the beginning of the active document and then converts the text to a table.

```
Set oRange1 = ActiveDocument.Range(Start:=0, End:=0)
oRange1.InsertBefore "one" & vbTab & "two" & vbTab & "three" & vbCr
Set oTable1 = oRange1.ConvertToTable(Separator:=Chr(9), NumRows:=1, _
NumColumns:=3)
```

Returning the contents of each table cell

The following example defines an array equal to the number of cells in the first document table (assuming Option Base 1). The **For Each**...**Next** structure is used to return the contents of each table cell and assign the text to the corresponding array element.

```
If ActiveDocument.Tables.Count >= 1 Then
    Set oTable = ActiveDocument.Tables(1)
    iNumCells = oTable.Range.Cells.Count
    ReDim aCells(iNumCells)
    i = 1
    For Each oCell In oTable.Range.Cells
        Set myRange = oCell.Range
        myRange.MoveEnd Unit:=wdCharacter, Count:=-1
        aCells(i) = myRange.Text
        i = i + 1
    Next oCell
End If
```

Copying all tables in the active document into a new document

This example copies the tables from the current document into a new document.

```
If ActiveDocument.Tables.Count >= 1 Then
    Set oDoc1 = ActiveDocument
    Set MyRange = Documents.Add.Range(Start:=0, End:=0)
    For Each oTable In oDoc1.Tables
        oTable.Range.Copy
        With MyRange
            .Paste
            .Collapse Direction:=wdCollapseEnd
            .InsertParagraphAfter
            .Collapse Direction:=wdCollapseEnd
        End With
    Next
End If
```

Working With Other Common Objects

Using the HeaderFooter Object

The **HeaderFooter** object can represent either a header or a footer.

Use **Headers**(index) or **Footers**(index), where index is one of the **WdHeaderFooterIndex** constants (**wdHeaderFooterEvenPages**, **wdHeaderFooterFirstPage**, or **wdHeaderFooterPrimary**), to return a single **HeaderFooter** object. The following example changes the text of both the primary header and the primary footer the first section of the active document.

```
With ActiveDocument.Sections(1)
    .Headers(wdHeaderFooterPrimary).Range.Text = "Header text"
    .Footers(wdHeaderFooterPrimary).Range.Text = "Footer text"
End With
```

You can also return a single **HeaderFooter** object by using the **HeaderFooter** property with a **Selection** object.

Note: You cannot add **HeaderFooter** objects to the **HeaderFooters** collection.

Use the **DifferentFirstPageHeaderFooter** property with the **PageSetup** object to specify a different first page. The following example inserts text into the first page footer in the active document.

```
With ActiveDocument
  .PageSetup.DifferentFirstPageHeaderFooter = True
  .Sections(1).Footers(wdHeaderFooterFirstPage).Range.InsertBefore _
      "Written by Joe Smith"
End With
```

Use the **OddAndEvenPagesHeaderFooter** property with the **PageSetup** object to specify different odd and even page headers and footers. If the **OddAndEvenPagesHeaderFooter** property is True, you can return an odd header or footer by using **wdHeaderFooterPrimary**, and you can return an even header or footer by using **wdHeaderFooterEvenPages**.

Use the **Add** method with the **PageNumbers** object to add a page number to a header or footer. The following example adds page numbers to the primary footer the first section of the active document.

```
With ActiveDocument.Sections(1)
    .Footers(wdHeaderFooterPrimary).PageNumbers.Add
End With
```

Using FormFields Objects

You can create a Word online form that includes check boxes, text boxes and drop-down list boxes. The corresponding VBA objects are **CheckBox**, **TextInput** and **DropDown**. All these objects can be returned from any **FormField** object in the **FormFields** collection.

Using the FormFields Collection

Use the **FormFields** property to return the **FormFields** collection. The following example counts the number of text box form fields in the active document.

```
For Each aField In ActiveDocument.FormFields
    If aField.Type = wdFieldFormTextInput Then count = count + 1
Next aField
MsgBox "There are " & count & " text boxes in this document"
```

Use the **Add** method with the **FormFields** object to add a form field. The following example adds a check box at the beginning of the active document and then selects the check box.

```
Set ffield = ActiveDocument.FormFields.Add( _
    Range:=ActiveDocument.Range(Start:=0,End:=0),
Type:=wdFieldFormCheckBox)
ffield.CheckBox.Value = True
```

Use **FormFields**(index), where index is a bookmark name or index number, to return a single **FormField** object. The following example sets the result of the Text1 form field to "xxx."

```
ActiveDocument.FormFields("Text1").Result = "xxx"
```

The index number represents the position of the form field in the selection, range, or document. The following example displays the name of the first form field in the selection.

```
If Selection.FormFields.Count >= 1 Then
    MsgBox Selection.FormFields(1).Name
End If
```

Use the Add method with the **FormFields** object to add a form field. The following example adds a check box at the beginning of the active document and then selects the check box.

```
Set ffield = ActiveDocument.FormFields.Add( _
  Range:=ActiveDocument.Range(Start:=0, End:=0), Type:=wdFieldFormCheckBox)
ffield.CheckBox.Value = True
```

Use the **CheckBox**, **DropDown**, and **TextInput** properties with the **FormField** object to return the **CheckDown**, **DropDown**, and **TextInput** objects. The following example selects the check box named "Check1."

```
ActiveDocument.FormFields("Check1").CheckBox.Value = True
```

Modifying Word Commands

You can modify most Word commands by turning them into macros. For example, you can modify the **Open** command on the **File** menu so that instead of displaying a list of Word document files (in Windows, files ending with the .DOC file name extension), Word displays every file in the current folder.

To display the list of built-in Word commands in the **Macro** dialog box, you select **Word Commands** in the **Macros In** box. Every menu command and every command available on a toolbar or through shortcut keys is listed. Menu commands begin with the menu name associated with the command. For example, the **Save** command on the **File** menu is listed as **FileSave**.

You can replace a Word command with a macro by giving a macro the same name as a Word command. For example, if you create a macro named "FileSave," Word runs the macro when you choose **Save** from the **File** menu, click the **Save** toolbar button, or press the **FileSave** shortcut key combination.

This example takes you through the steps needed to modify the **FileSave** command.

➢ **To modify a Word command**

1 On the **Tools** menu, point to **Macro**, and then click **Macros**.

2 In the **Macros In** box, select **Word Commands**.

3 In the **Macro Name** box, select the Word command to modify (e.g **FileSave)**.

4 In the **Macros In** box, select a template or document location to store the macro. For example, select **Normal.dot** (**Global Template**) to create a global macro (the

FileSave command will be automatically modified for all documents).

5 Click the **Create** button.

The VBA editor opens with module displayed that contains a new procedure whose name is the same as the command you clicked. If you clicked the **FileSave** commans, the FileSave macro appears as shown below:

```
Sub FileSave()
    ' Saves the active document or template
    ActiveDocument.Save
End Sub
```

You can add additional instructions or remove the existing ActiveDocument.Save instruction. Now every time the **FileSave** command runs, your **FileSave** macro runs instead of the word command. To restore the original **FileSave** functionality, you need to rename or delete your **FileSave** macro.

Note: You can also replace a Word command by creating a code module named after a Word command (for example, **FileSave**) with a subroutine named Main.

Working With Events

An event is an action that is recognised by an object (such as opening a document or quitting an application) and for which you can write code to provide a response.

Document Events

The Document object supports three events: **Close**, **New** and **Open**. You write procedures to respond to these events in the class module named "ThisDocument." Use the following steps to create an event procedure.

1 Under your **Normal** project or document project in the **Project Explorer** window, double-click **ThisDocument**. (In Folder view, **ThisDocument** is located in the Microsoft Word Objects folder.)

2 Select **Document** from the **Object** drop-down list box.

3 Select an event from the **Procedure** drop-down list box. An empty subroutine is added to the class module.

4 Add the Visual Basic instructions you want to run when the event occurs.

The following example shows a **New** event procedure in the **Normal** project that will run when a new document based on the **Normal** template is created.

```
Private Sub Document_New()
    MsgBox "New document was created"
End Sub
```

The following example shows a **Close** event procedure in a document project that runs only when that document is closed.

```
Private Sub Document_Close()
    MsgBox "Closing the document"
End Sub
```

Unlike auto macros, event procedures in the **Normal** template don't have a global scope. For example, event procedures in the **Normal** template only occur if the attached template is the **Normal** template.

If an auto macro exists in a document and the attached template, only the auto macro stored in the document will execute. If an event procedure for a **Document** event exists in a document and its attached template, both event procedures will run.

ActiveX Control Events

Word implements the **LostFocus** and **GotFocus** events for ActiveX controls in a Word document. The other events listed in the Procedure drop-down list box in are documented in Microsoft Forms Help

The following example shows a **LostFocus** event procedure that runs when the focus is moved away from **CheckBox1**. The macro displays the state of **CheckBox1** using the **Value** property (**True** for selected and **False** for clear).

```
Private Sub CheckBox1_LostFocus()
    MsgBox CheckBox1.Value
End Sub
```

Application Events

Application events occur when the user quits the application or the focus is shifted to another document. Unlike document and ActiveX control events, however, the **Application** object events are not enabled by default. To create an event handler for an event of the **Application** object, you need to complete the following three steps:

1 Declare an object variable in a class module to respond to the events.

Before you can write procedures for the events of the **Application** object, you must create a new class module and declare an object of type **Application** with events. For example, assume that a new class module is created and called **EventClassModule**. The new class module contains the following code.

```
Public WithEvents App As Word.Application
```

2 Write the specific event procedures.

After the new object has been declared with events, it appears in the **Object** drop-down list box in the class module, and you can write event procedures for the new object. (When you select the new object in the **Object** box, the valid events for that object are listed in the **Procedure** drop-down list box.) Select an event from the **Procedure** drop-down list box; an empty procedure is added to the class module.

```
Private Sub App_DocumentChange()
End Sub
```

3 Initialize the declared object from another module.

Before the procedure will run, you must connect the declared object in the class module (App in this example) with the **Application** object. You can do this with the following code from any module (EventClassModule is the name of the new class module you created for this purpose).

```
Dim X As New EventClassModule
Sub Register_Event_Handler()
    Set X.App = Word.Application
End Sub
```

Run the Register_Event_Handler procedure. After the procedure is run, the App object in the class module points to the Word **Application** object, and the event procedures in the class module will run when the events occur.

After you have enabled events for the **Application** object, you can create event procedures for the events described in the following table:

Event	Description
DocumentChange	Occurs when a new document is created, when an existing document is opened or when another document is made the active document.
Quit	Occurs when the user quits Word.

Using Auto Macros

By giving a macro a special name, you can run it automatically when you perform an operation such as starting Word or opening a document. Word recognises the following names as automatic macros, or "auto" macros.

Macro name	When it runs
AutoExec	When you start Word or load a global template
AutoNew	Each time you create a new document
AutoOpen	Each time you open an existing document
AutoClose	Each time you close a document
AutoExit	When you quit Word or unload a global template

Auto macros in code modules are recognised if either of the following conditions are true.

- The module is named after the auto macro (for example, AutoExec) and it contains a procedure named "Main."
- A procedure in any module is named after the auto macro.

Just like other macros, auto macros can be stored in the **Normal** template, another template, or a document. The only exception is the AutoExec macro, which will not run automatically unless it is stored in the **Normal** template or a global template stored in the folder specified as the **Startup** folder.

In the case of a naming conflict (multiple auto macros with the same name), Word runs the auto macro stored in the closest context. For example, if you create an **AutoClose** macro in a document and the attached template, only the auto macro stored in the document will execute. If you create an **AutoNew** macro in the normal template, the macro will run if a macro named **AutoNew** doesn't exist in the document or the attached template.

Note:

You can hold down the SHIFT key to prevent auto macros from running. For example, if you create a new document based on a template that contains an **AutoNew** macro, you can prevent the **AutoNew** macro from running by holding down SHIFT when you click OK in the **New** dialog box (**File** menu) and continuing to hold down SHIFT until the new document is displayed. In a macro that might trigger an auto macro, you can use the following instruction to prevent auto macros from running.

```
WordBasic.DisableAutoMacros
```

11. INTERACTING WITH OTHER APPLICATIONS

In addition to working with Word or Excel data and functions you may want your application to exchange data with other Office applications, or a program you have written yourself. There are several ways to communicate with other applications, including OLE automation, dynamic data exchange DDE and dynamic-link libraries (DLLs).

Using Automation

Automation (previously known as OLE Automation) allows you to retrieve, edit and export data by referencing another application's objects, properties and methods. Objects that can be returned from outside the application are called *Automation objects*. An application that exposes its Automation objects to other applications is called an *server application*. An application that can access and manipulate Automation objects is called an *automation controller.*

To exchange data with another application by using Automation while working in Excel, say, you first create a reference to the application you want to communicate with (the server). You then add, change or delete information using the server's objects, properties and methods. When you have finished using the server, you close it from within the controller application.

In more detail, the process involves three steps:

1. The first step towards making, say, Word available to another application for Automation is to create a reference to the Word type library. This is done in the VBA Editor by selecting **References** from the **Tools** menu, and selecting the check box next to **Microsoft Word 8.0 Object Library**.
2. The second step is to declare an object variable that will refer to the Word **Application** object, as in this example:

```
Dim appWD as Word.Application.8
```

3. The third step is to use the VBA **CreateObject** or **GetObject** function with the Word OLE Programmatic Identifier (Word.Application.8 or Word.Document.8), as shown below (setting the visibility to **True** allows you to see the instance of Word):

```
Set appWD = CreateObject("Word.Application.8")
AppWd.Visible = True
```

The **CreateObject** function returns a Word **Application** object and assigns it to appWd. Using the objects, properties and methods of the Word **Application** object, you can control Word. The following example creates a new Word document:

```
AppWd.Documents.Add
```

The **CreateObject** function starts a Word session that Automation will not close when the object variable that references the **Application** object expires. Setting the object reference to the **Nothing** keyword will not close Word either - you must use the **Quit** method. The following Excel example inserts data from cells A1:B10 on Sheet1 into a new Word document and then arranges the data in a table. The example uses the **Quit** method to close the new instance of Word if the **CreateObject** function was used. If the **GetObject** function returns error 429 (no instance of Word running) the example uses **CreateObject** to start a new Word session.

```
Dim appWd as Word.Application
Err.Number= 0
On Error GoTo notloaded
Set appWd = GetObject(, "Word.Application.8")
```

```
Notloaded:
If Err.Number = 429 Then
      Set appWd = CreateObject("Word.Application.8")
TheError = Err.Number
End If
AppWd.Visible = True
Wirth appWd
      Set myDoc = .Documents.Add
      With .Selection
            For Each c in Worksheets("Sheet1").Range("A1:B10")
                  .InsertAfter Text:=c.Value
                  Count = Count + 1
                  If Count Mod 2 = 0 Then
                        .InsertAfter Text:=vbCr
                  Else
                        .InsertAfter Text:=vbTab
                  End If
            Next c
            .Range.ConvertToTable Separator:=wdSeparateByTabs
            .Tables(1).AutoFormat Format:=wdTableFormatClassic1
      End With
      MyDoc.SaveAs FileName:="C:\Temp.doc"
End With
If theError = 429 the appWd.Quit
Set appWd = Nothing
```

Automating Another Application from Word

To exchange data with another application by using Automation from Word, you must first set a reference to the other application's type library in the **References** dialog box (**Tools** menu). Then the objects, properties and methods of the other application will appear in the Object Browser, and the syntax will be automatically checked at compile time. You can also get context-sensitive help on these objects, properties and methods.

Next, declare object variables that will refer to the objects in the other application as specific types. The following example declares a variable that will point to the Excel **Application** object:

```
Dim xlObj as Excel.Application.8
```

You obtain a reference to the Automation object by using **CreateObject** or **GetObject** as before, and you then have access to the objects, properties and methods of the other application. The following Word examples determines if Excel is currently running and if so uses **GetObject** to create a reference to the Excel **Application** object. If not, **CreateObject** is used. The example then sends the selected text to cell A1 of Sheet1 in the active workbook. Use the **Set** statement with the **Nothing** keyword to clear the Automation object variable after the task has been completed.

```
Dim xlObj As Excel.Application.8
If Tasks.Exists("Microsoft Excel") = True then
      Set xlObj = GetObject(, "Excel.Application.8")
Else
      Set xlObj = CreateObject("Excel.Application.8")
End If
XlObj.Visible = True
If xlObj.Workbooks.Count = 0 the xlObj.Workbooks.Add
XlObj.Worksheets("Sheet1").Range("A1").Value = Selection.Text
Set xlObj = Nothing
```

The following Word example creates a new PowerPoint presentation with the first text box including the name of the first active Word document and the second including the text from the first paragraph in the active document:

```
Dim pptObj as PowerPoint.Application.8
If Tasks.Exists("Microsoft PowerPoint") = True Then
      Set pptObj = GetObject(, "PowerPoint.Application.8")
Else
      Set pptObj = CreateObject("PowerPoint.Application.8")
End If
PptObj.Visible = True
Set pptPres = pptObj.Presentations.Add
Set aSlide = pptPres.Slides.Add(Index:=1, Layout:=ppLayoutText)
ASlide.Shapes(1).TextFrame.TextRange.Text = ActiveDocument.Name
ASlide.Shapes(2).TextFrame.TextRange.Text = _
   ActiveDocument.Paragraphs(1).Range.Text
Set pptObj = Nothing
```

Other possible declarations are given below:

```
Dim xlApp As Excel.Application
Dim xlBook As Excel.Workbook
Dim xlSheet As Excel.WorkSheet
Set xlApp = CreateObject("Excel.Application")
Set xlBook = xlApp.Workbooks.Add
Set xlSheet = xlBook.Worksheets(1)
```

Communicating with Embedded Word Objects

When you embed a Word document in an Excel worksheet, Excel controls the object, and Word controls everything inside the object.

A linked object is an object that contains a reference pointer to its application. Data associated with a linked object are not stored within the application that contains the object. If you change the data in a linked application, the data will change in the original application as well.

An embedded object contains a "snapshot" of the data existing at the time you embedded the object. Data associated with an embedded object are stored in the file in which the object is embedded. If you change the data in an embedded object, the data in the original application do not change.

An OLE container application can store embedded or linked objects provided by OLE object applications. An OLE object application is an application that exposes an OLE object.

Editing an Embedded Word Object

To edit a Word document embedded as an OLE object, you must activate it before you can refer to one of the top-level objects. The following example activates and edits a Word document, which is the first OLE object on sheet1:

```
Dim wordobj as Object
Worksheets("sheet1").OLEObjects(1).Verb
Set wordobj=Worksheets("sheet1").OLEObjects(1).Object
.Application.WordBasic
With wordobj
      .Insert "This is the first new line."
      .InsertPara
      .LineUp 1
      .EndOfLine 1
      .Bold
      .LineDown 1
```

```
      End With
```

Using the Verb method with no arguments, as above, is equivalent to using the Activate method. For more information, see the Verb method in Help.

Printing an Embedded Word Object

```
Dim wordobj as Object
Worksheets("sheet1").OLEObjects(1).Verb
Set wordobj=Worksheets("sheet1").OLEObjects(1).Object .Application.WordBasic
With wordobj
      .Insert "This is the first new line."
      .InsertPara
      .FilePrint
      .FileClose
End With
```

Sending Keystrokes

You can send keystrokes to other Windows applications using the **SendKeys** method.

The **SendKeys** method is processed when your system is idle or when the **DoEvents** method is called. If the **Wait** argument of the **SendKeys** method is **True**, Excel waits for the keys to be processed before returning control to the calling procedure; if **False** the procedure continues to run without waiting for the keys to be processed. The following example sends keystrokes to the Calculator that add numbers from 1 to 10 and then close the Calculator

```
Sub DemoSendKeys()
      returnvalue = Shell("calc.exe",1)
      AppActivate returnvalue
      For i=1 to 10
            SendKeys i & "{+}", True
      Next i
      SendKeys "=", True
      SendKeys "%{F4}", True
End Sub
```

Note Keystrokes are sent to the active application. If this isn't the one you want to receive the keystrokes, you need to activate it using the AppActivate statement. If the application you want to send keystrokes to isn't already running, start it using the Shell function.

To specify characters that aren't displayed when you press the key (such as ENTER or TAB), enclose the key code in braces - {} - and enclose the braces in straight double-quotation marks. To specify a key to be used in combination with SHIFT, CTRL or ALT, precede the braces with +, ^ or % respectively. The following example sends the key combination ALT+F4:

```
SendKeys "%{F4}", True
```

For a complete list of key codes, see "SendKeys" method in Help.

NB : you cannot send keystrokes that generate interrupts instead of character codes, such as CTRL+ALT+DEL, or PRINT SCREEN.

12. USING DLLs and THE WINDOWS API

A dynamic-link library (DLL) is a library of routines loaded into memory and linked to applications at run-time. DLLs are usually created in a programming language such as C, Delphi or Visual Basic, and contain procedures that you can call in your applications. You can call DLL functions and functions within the Windows Applications Programming Interface (API) from VBA.

Because DLL routines reside in files that are external to your application, you must let VBA know where it can find the routines you want to use, and what the arguments of each routine are. This information is provided with the **Declare** statement, placed in the declarations section of a module. Once you have declared a DLL or Windows API routine you can use it in your code like any other routine, although it must be emphasised that you must pay very careful to ensuring the correct number any type of arguments are passed. If you fail to do this you can crash your system!

There are two steps to using a DLL/API routine:

1. Tell VBA about the routine by using a **Declare** statement.
2. Make the actual call

You declare a function once only, but can call it a number of times from any procedure in that workbook.

Declaring a DLL Routine

Place declarations at the top of your code (this is not necessary but it makes your programs much easier to read this way!). For example:

```
Declare Function SetWindowText Lib "user32" Alias "SetWindowTextA" _
  (ByVal hwnd As Long, ByVal lpString As String) As Long

Declare Function GetSystemMetrics Lib "user32" (ByVal nIndex As Long) As Long
```

In 32-bit Microsoft Windows systems, you can use conditional compilation to write code that can run on either Win32 or Win16:

```
#If Win32 Then
      Declare Sub MessageBeep Lib "User32" (ByVal N As Long)
#Else
      Declare Sub MessageBeep Lib "User" (ByVal N As Integer)
#End If
```

These statements declare routines in the Windows API library contained in USER32.DLL. A full list of all Windows API functions and subroutines, containing the syntax of their calls, can be found in the help file "WIN32API.TXT" which is available for download from the unit webpages.

Also in the above directory are the following:

WIN32API.TXT Lists Visual Basic calls to the 32-bit Windows API

Calling DLL Routines

Once a routine has been declared, you a can call it just as you would a VBA statement or function. The following example uses the **GetSystemMetrics** Windows API routine to determine whether a mouse is installed.

The value 19 assigned to SM_MOUSEPRESENT is one of a number of values that can be passed to this function.

```
Declare Function GetSystemMetrics Lib "user32" ByVal nIndex As Long) As Long
Sub Main()
      SM_MOUSEPRESENT=19
      If GetSystemMetrics(SM_MOUSEPRESENT) Then MsgBox "Mouse Installed"
End Sub
```

Important: VBA can't verify that you are sending valid values to a DLL. If you pass incorrect values the routine may fail, which may cause unexpected behaviour or errors in your application or in the operating system. Take care when experimenting with DLLs and save your work often.

An example of a partly-useful API call is in the generation of the standard Windows "About" box. This is part of the code in the "BISECT.XLS" worksheet:

```
Declare Function ShellAbout Lib "shell32.dll" Alias "ShellAboutA" _
 (ByVal hwnd As Long, ByVal szApp As String, ByVal szOtherStuff As _
  String, ByVal hIcon As Long) As Long

Declare Function GetActiveWindow Lib "user32" Alias "GetActiveWindow" () _
  As Long

Sub About()
   Dim hWnd As Integer
   Dim windowname As String
   nl$ = Chr$(10) + Chr$(13)
   hWnd = GetActiveWindow()
   x = ShellAbout(hWnd, "Brilliant Code", nl$ + Chr$(169) + _
     " Jeff Waldock, July 9, 1998" + nl$, 0)
End Sub
```

Note that the declaration statements must each be one single line, not with the line continuation characters included here.

The **GetActiveWindow** function determines the "handle" hWnd of the currently active window. This is one of the parameters to the **ShellAbout** function; the others are obvious! Note also that parameters are often passed to the DLLs using the "ByVal" keyword. This indicates that the values are passed by value rather than by reference (i.e. address). It is important that you remember to do this!