Thank you very much for attending the Excel VBA course. I hope that we managed to give you a few ideas on using VBA with Excel. You should find further information on all the topics that we covered in the course in this booklet.

Feel free to copy any of the content of this publication.

This is an Adobe Acrobat pdf document. To copy any content you have to select it first. Use the Text Select Tool as illustrated below. Or press V on the keyboard and then copy and paste as usual.

You can not print this document; ask for the printer friendly version.

Excel Macros

Visual Basic for Applications

Table of Contents

The Process

Macros usually start with a recording but recorded macros do not give you enough flexibility to control the whole process that you want to execute. Often you will need to introduce decision making and repetition into your macro code. This has to be done by typing-in control structures and assignment statements in the VBA language.

In this example there is a range of cells on the worksheet and where the cell value is greater than 500 is has to be formatted in bold and the cell value doubled. Conditional Formatting is of no use for this as it can not change the cell value. We must use a macro.

```
Sub Step1_Recording()
'
' Macro2 Macro
' Macro recorded by me
'
  Range("H4").Select
  Selection.Font.Bold = True
  ActiveCell.FormulaR1C1 = "=4.72*2"
End Sub
```

Here is the initial recording. It has shown us how to make the entry bold but has simply recorded the doubling of a specific value in a cell.

We have to double the value of any cell and we will have to type-in the relevant instruction.

```
Sub Step2_Abstraction()

  Selection.Font.Bold = True
  ActiveCell.Value = ActiveCell.Value * 2

End Sub
```

The cell selections and comments have been removed and we have entered an assignment statement to double the cell value.

```
Sub Step3_DecisionMaking()

  If ActiveCell.Value > 500 Then
    ActiveCell.Font.Bold = True
    ActiveCell.Value = ActiveCell.Value * 2
  End If

End Sub
```

Now we introduce the logical decision making structure using the If-Then-End If keywords.

```
Sub Step4_Looping()

  For Each cell In Range("A1").CurrentRegion
    If cell.Value > 500 Then
      cell.Font.Bold = True
      cell.Value = cell.Value * 2
    End If
  Next

End Sub
```

Next, we construct a collection loop to address each cell in turn in a specified area. The loop will visit each cell in the continuous area of cells associated with cell A1.

The decision structure is enclosed in the loop.

```
Sub Step5_ErrorProof()

  On Error Resume Next

  For Each cell In Range("A1").CurrentRegion
    If cell.Value > 500 Then
      cell.Font.Bold = True
      cell.Value = cell.Value * 2
    End If
  Next

End Sub
```

The slightest error will cause a macro to crash so we either have to think of all the possible situations where our macro could fail and test for them in our code. Or we decide that the only errors that we could encounter would be so petty that they are not worth considering and enter the statement that ignores all errors:

On Error Resume Next

Terminology

You are using the Microsoft Visual Basic for Applications (VBA) language to automate the manipulation of the Microsoft Excel application.

You need to know about how to address or *access* the various parts or *objects* of the Excel application and how these objects are organised in the *object model*.

You control the flow of this process using the *control structures* of VBA.

In the world of Excel, you describe this type of process as a macro; short for macroinstruction. In the world of Visual Basic you describe it as a *procedure*, a set of sequential instructions to complete a single process.

Procedures are stored in *Modules*. Modules are stored in Workbooks. The collection comprising of Worksheets, Modules and their containing Workbook file is called a *Project*.

The Basics of VBA

Data storage

There are no cells in a module, so when you are working and you need to store some information you need to use the computer's memory. These slices of memory are called *variables*; you use an identifier in your code and assign values to it.

It might be necessary to Declare your variables before you can use them. See Variable Declaration

Subroutine Calls and passing values

Complex processes need to be broken down into separate procedures. Then you need to have the procedures interact with each other. Procedures can *Call* other procedures, the *flow of control* goes to the *subroutine* and then returns to the *caller*. Data stored in variables is shared between the procedures by *passing*.

```
Sub Main()

      'Assign a value to a variable.
      x = 500

      'Change the variable's value.
      x = x + 10

      'Subroutine call, passing the x variable.
      Call MyOtherSub(x)

      'MsgBox function and concatenation operator.
      MsgBox "The value of x is " & x

End Sub
```

```
Sub MyOtherSub(x)

      'Assign a value to the variable.
      x = "a text value."

      'Return to calling procedure-no code required.

End Sub
```

Control Structures

Control structures are required for decision-making and repetition or *looping*.

Decision making

Decision-making structures are If-Then-Else and Case Statements. If-Then-Else has two syntax structures, a Case Statement only one.

If-Then-Else

In-Line Form

```
If conditional_test Then True_ statement Else False_ statement
```

Only one True or False statement is available. Else is optional. The structure is contained on one logical line. A logical line can be broken into more than one physical line by using line-continuation. See Line Continuation

Block Form

```
If conditional_test Then
      True_ statement
      True_ statement
ElseIf conditional_test Then
      True_ statement
ElseIf conditional_test Then
      True_ statement
Else
      False_ statement
End If
```

Multiple True or False statements. The ElseIf and Else clauses are optional. The structure is contained on multiple lines. Use either the Block form or the In-line form; do not try to combine them or you will cause a Compile error.

Examples

```
Sub InLineIfForm()

    x = 50

    If x > 100 Then MsgBox "Big" Else MsgBox "Small"

End Sub
```

```
Sub BlockIfForm()

      x = 100

      If x >= 250 Then
            msg = "Large."
      ElseIf x >= 50 And x < 250 Then
            msg = "Medium."
      Else
            msg = "Small."
      End If

      MsgBox "At " & x & ", x is " & msg

End Sub
```

Case Statements

Comparing a single test expression against multiple possible values. Each case test consists of a test and an outcome to that test. The outcome statements may be multiple lines and may also be omitted.

```
SelectCase TestExpression
      Case 5 'TestExpression is equal to 5.
            Statements
            Statements
      Case Is > 25 'TestExpression is greater than 25.
            Statements
      Case 10 To 12 'TestExpression is between 10 and 12.
            Statements
      Case 4,7,9 'TestExpression is 4,7 or 9.
            Statements
      Case Else 'TestExpression is anything else.
            Statements
End Select
```

Case statements are usually more concise and readable than the equivalent If-Then-Else structure. As with any decision structure, there is only one outcome; make the tests in the correct order i.e. is x greater than 10? Followed by is x greater than 5? Not the reverse.

In the following example, the value of the variable x determines the value of the variable y. If x is 250 or more then y is "Large", if x is from 50 to 249 then y is "Medium" and for any other value y is "Small":

```
Sub CaseStatement()

      x = 100

      Select Case x
            Case Is >= 250
                  y = "Large."
            Case 50 To 249
                  y = "Medium."
            Case Else
                  y = "Small."
      End Select

      MsgBox "At " & x & ", x is " & y

End Sub
```

CHOOSE and SWITCH

CHOOSE and SWITCH are VBA functions, rather than *keywords* and their decision-making process is often neglected. Although they are not as flexible as If-Then-Else or Case Statements they are invaluable when the decision making process is based on substitution or the evaluation of numeric values.

Choose function

Selects and returns a value from a list of arguments.

Choose(index, choice1, choice2, etc.)

Where *index* is a numeric expression that results in a value between 1 and the number of available choices. Choose returns a value from the list of choices based on the value of index. If index is 1, Choose returns the first choice in the list; if index is 2, it returns the second choice, and so on. If index is not a whole number, it is rounded to the nearest whole number before being evaluated.

Example

The message box displays the second item in the list:

```
Sub Main()
      x = 2
      y = Choose(x, "Tom", "Dick", "Harry")
      MsgBox y
End Sub
```

Switch Function

Evaluates a list of pairs of expressions and values and returns the value associated with the first expression in the list that is True.

Switch(expr1, value1, expr2, value2, etc.)

The expressions are evaluated from left to right but can be entered in any order.

Example

The message box displays "STG", the value associated with the x="UK" expression:

```
Sub Main()
      x = "UK"
      y = Switch(x = "UK", "STG", x = "USA", "USD", x = "DEN", "DKK")
      MsgBox y
End Sub
```

Decision making code is a matter of personal taste and judgement. Generally speaking, If-Then-Else is the most flexible, Case Statements are best where you are testing one expression over many different conditions, the CHOOSE function is best for processing sets of numbers and SWITCH is best for substitution.

In the following example, all four methods are demonstrated. An organisation has a financial year that starts in April and we need to take the current calendar month value and convert it into the current accounting month value; April is 1 etc. The x variable stores the current month as returned by the Month and Date functions and we have to calculate the value of the MonthNo variable:

```
x = Month(Date)

'If the date is 4 or more; deduct 3, otherwise add 9.
If x >= 4 Then
  MonthNo = x - 3
Else
  MonthNo = x + 9
End If

'When the date is from 4 to 12, deduct 3. When it is from 1 to 3, add 9.
Select Case x
  Case 4 To 12
    MonthNo = x - 3
  Case 1 To 3
    MonthNo = x + 9
End Select

'Pick the value from the list.
MonthNo = Choose(x, 10, 11, 12, 1, 2, 3, 4, 5, 6, 7, 8, 9)

'Match the pair in the list.
MonthNo = Switch(x = 1, 10, x = 2, 11, x = 3, 12, x = 4, 1, _
         x = 5, 2, x = 6, 3, x = 7, 4, x = 8, 5, x = 9, 6, _
         x = 10, 7, x = 11, 8, x = 12, 9)
```

Looping

When a process has to be repeated it is best to use a loop structure to make sections of instructions repeat rather than have multiple sets of duplicated instructions.

Conditional Loops

Repetition while a certain condition is satisfied or until a certain condition is satisfied.

Check for the condition before running the loop:

```
Do While condition
      Statements
Loop
```

Execute the commands once before checking the condition:

```
Do
      Statements
Loop While condition
```

Use the keywords Until or While to define the condition, placing them either at the top or at the end of the Do…Loop.

```
Sub DoLoops1()

      x = 10

      Do Until x > 40
            x = x + 10
            MsgBox x
      Loop

End Sub
```

```
Sub DoLoops2()

      x = 10

      Do
            x = x + 10
            MsgBox x
      Loop While x < 40

End Sub
```

You can conditionally break out of a Do...Loop using Exit Do.

Save your file before testing the code. It is very easy to get stuck in a conditional loop. You must try to terminate the procedure if you are stuck. Press the ESCAPE key. If this fails, try CTRL and BREAK together. It's bad news after this, CTRL+ALT+DELETE.

There is another Conditional Loop that is often seen, While...Wend. It is an equivalent structure to Do While…Loop, which supersedes it.

The BASIC language was developed in the early 1960's and contains many older or legacy structures. They are still supported but are rarely used.

Counter Loops

Iterating a loop for a specific number of repetitions:

```
Sub ForNextCounterLoop1()

    For i = 1 To 5
            MsgBox "The counter value is " & i
    Next

End Sub
```

```
Sub ForNextCounterLoop2()

    For i = 100 To 10 Step -10
            MsgBox "The counter value is " & i
    Next

End Sub
```

Implementing the structure on Excel objects, a loop to protect every worksheet in the workbook:

```
Sub ForNextCounterLoop3()

    'The Count property of the Worksheets Collection Object returns the
    'required stop value.

    For i = 1 To Worksheets.Count
            'Worksheets returned by using their index values.
            Worksheets(i).Protect
    Next i

End Sub
```

You can conditionally break out of a For...Next loop using Exit For. Loops can contain other loops, this is called *nesting*. There is no need to restate the loop counter variable after the Next keyword; usually it is only used to identify the ends of nested loops:

```
For i=1 To 10 'Exterior loop.
    Statements
    For j=1 To 5 'Interior loop.
      Statements
    Next j
Next i
```

Collection Loops

For iterating a collection; either a collection of objects in Excel or a collection in memory:

```
For Each Element In Collection
      Statements
Next
```

Where *Element* represents one of the items in *Collection*. Element is a variable. The collection is either a defined Excel Collection Object or is a container reference. There is no need to explicitly reference each element; it is implicit to the collection and the variable is used to represent each element on each iteration of the loop.

In the first example the Collection is the Worksheets Collection; the loop goes through each *member* of the Collection. In the second example the Collection is defined as a range of cells; a range contains cells so the loop goes to each one in turn. In neither case do you have to do make the object reference in the code, the loop does the referencing for you. The Worksheets Collection is a defined *Collection Object* in Excel, whereas in the second example the range reference is a *container*, a reference to a set of like objects.

```
Sub ForEachCollectionLoop1()

      'Unprotect each Worksheet in the Workbook.
    For Each Wsht In Worksheets
            Wsht.Unprotect
      Next
End Sub
```

```
Sub ForEachCollectionLoop2()

      'Double the cell value if it contains a number,
      'otherwise clear the cell.
      For Each cell In Range("A1:G50")
            If IsNumeric(cell) Then
                  cell.Value = cell.Value * 2
            Else
                  cell.Clear
            End If
     Next
End Sub
```

Understanding the Excel Object Model

Review of theory: Objects, Methods and Properties

Excel is an Object Model, a hierarchical arrangement of references where the higher-level object, the *Parent* object, contains the lower level object, the *Child* object.

To return the name of the current workbook file:

```
x = ActiveSheet.Parent.Name
```

Objects are either singular or *Collection* objects. Collections are sets of like objects. There is a Worksheets Collection object and it has certain Properties, like its Count property, which is the number of Worksheets in the Collection. Each Worksheet is a member of the Worksheets Collection but it is also an individual Worksheet object and has, in turn, its own particular Properties, like its Name property

To calculate the number of Pivot Tables on the worksheet:

```
x = ActiveSheet.PivotTables.Count
```

Objects have associated *Methods* and *Properties*. Methods are actions that they can perform. Properties are their particular attributes. Most Properties are variable properties and you can change them by specifying a new value. Every statement in VBA code that manipulates a part of Excel must take the following form:

`Object.Property` or `Object.Method`

You must start with the Object reference. The object reference can either be specific or non-specific.

Non-specific Object Reference

Assign the red Fill colour to every cell on the active worksheet:

```
ActiveSheet.Cells.Interior.ColorIndex = 3
```

or

```
Cells.Interior.ColorIndex = 3
```

Specific Object Reference

Assign the red Fill colour to every cell on Sheet2:

```
Worksheets("Sheet2").Cells.Interior.ColorIndex = 3
```

The object reference only has to be in context, you only need a worksheet reference if the cell is on a worksheet that is not the active worksheet. You do not need a workbook reference unless you are manipulating a workbook other than the active workbook.

Object.Property assignment statements contain an equals sign

```
Worksheets(2).[A1:D20].Interior.ColorIndex = 3
```

Object.Method statements are rather different as they can accept additional required or optional arguments. The following statement copies A1:A50 to the Clipboard using the Copy method.

```
Range("A1:A50").Copy
```

The Copy method has an optional argument, Destination. Using this you can specify where you want to directly copy the cells, avoiding the Clipboard. There is a space character required between the Method and the argument value.

```
Range("A1:A50").Copy Destination:= Range("A100")
```

You can leave out the argument descriptor and just give the value but there must always be a space character between the Method and the value.

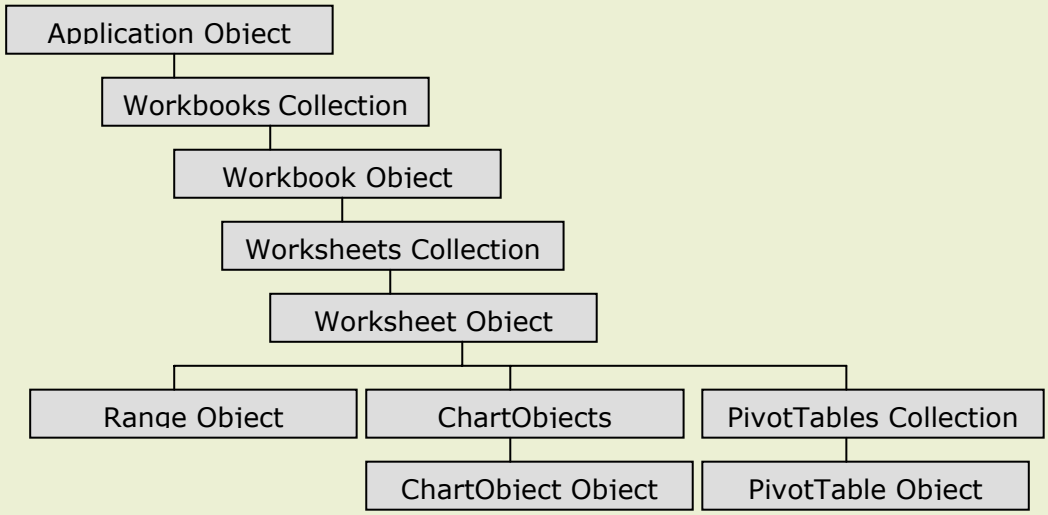
```
Range("A1:A50").Copy Range("A100")
```

For a fuller discussion on argument specification see By Name, By Order

The Excel Object Model

The full Excel Object Model has over 200 objects and is too detailed to show on one page. However you tend to only use certain objects on a regular basis and the following diagram shows the relationship between the most commonly used objects.

Search for "Microsoft Excel Objects" in VBA Help to see the full diagram.



Object references: Cells, Sheets and Workbooks

The macro recorder will show you what your object references are but it will not show you the variety of different expressions that can be used to access common Excel objects.

Non-specific Object References

`Selection`	The current selection
`ActiveCell`	The current active cell
`ActiveSheet`	The current worksheet
`ActiveWorkbook`	The current workbook
`ThisWorkbook`	Workbook containing the procedure

Specific Object References, various styles

`Range("A1")`	Cell A1
`Range("A1:F50")`	Range A1:F50
`[A1]`	Cell A1
`[A1:F50]`	Range A1:F50
`ActiveCell.Range("A2")`	The cell below the active cell
`Cells(1)`	Cell A1

`Range(Cells(1,1),Cells(50,6))`	Range A1:F50
`Range("NamedRange").Cells(1,1)`	The first cell in the named range
`Range("A:A")`	Column A
`[A:A]`	Column A
`Columns(1)`	Column A
`Range("5:5")`	Row 5
`[5:5]`	Row 5
`Rows(5)`	Row 5
`Sheets("Sheet1")`	The Sheet called Sheet1
`Worksheets("Sheet1")`	The Worksheet called Sheet1
`Sheets(2)`	The second Sheet in the Workbook
`Worksheets(3)`	The third Worksheet in the Workbook
`Worksheets("Sheet1").Range("A1")`	Cell A1 on Sheet1
`[Sheet1].[A1]`	Cell A1 on Sheet1
`ActiveSheet.Next`	The sheet after the active sheet
`Workbooks("Basic")`	The Workbook file, Basic.xls

Square brackets

The full object reference to the worksheet cell A1 is *Range("A1")*. If you are typing-in cell references rather than recording, it is easier to use the shortcut notation using square brackets, [A1]. You can use the same style of referencing on other objects as well, such as worksheets but there are a number of rules and restrictions.

It is usually best to restrict the square bracket notation to cell references only, where it is entirely definitive and reliable.

With...End With

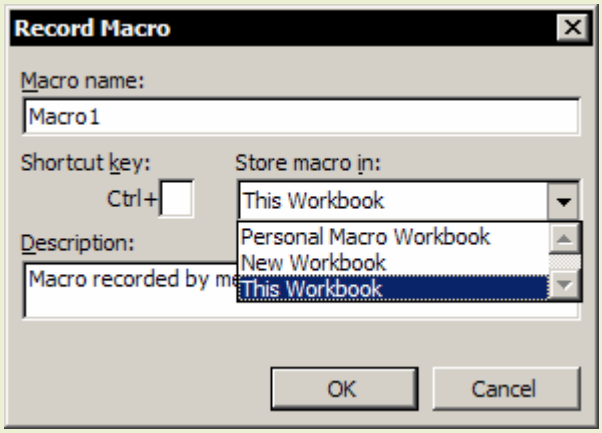
The With statement is used so the object reference can be made and then retained so that multiple actions may be carried out without having to repeat the same object reference in each statement.

You can keep the With reference open for as long as you like in the same procedure, just pointing to it using the dot operator. Every With requires an End With. You can have multiple With pointers. When you are reading code that uses multiple With pointers, the rule is simple; the dot points to the nearest With.

```
With Object
      .Property
      With .Child Object
            .Method
            .Method
      End With
End With
```

Recording and Editing

Recording a macro



Turn on the macro recorder by choosing, *Tools*, *Macro*, *Record New Macro* in Excel's worksheet environment. Choose where you want to store the module, fill in the Name and Shortcut key boxes. Turn off the recorder using the *Stop Recorder* Toolbar when you have finished.

If you want to set the shortcut key after the recording, choose *Tools*, *Macro*, *Macros* then select your macro from the list and click the *Options* button. The shortcut key

assignment has to be an alphabetical character. Your shortcut key overrides any Excel shortcut keys. If you want to rename the recorded macro, go to the module and change the Sub name. To see the recording in the VB Editor, choose *Tools*, *Macro*, *Macros* then select your macro from the list and click the *Edit* button.

Do not try to turn off the Macro Recorder by clicking the Close Box on the Stop Recording Toolbar. This hides the Toolbar and leaves the Macro Recorder still turned on. On your next recording, the Toolbar will not be visible. Display the Toolbar whilst you are recording with *View*, *Toolbars* to cure this problem.

Relative and Absolute recordings

The Relative Reference tool on the Stop Recording Toolbar governs the style of recording made when you select worksheet cells in your recording; an *Absolute* or a *Relative* recording. It is rather difficult to see what type of recording you are doing as the ToolTip always reads "Relative Reference" regardless of the state of the Control.

You get an Absolute reference recorded when the toolface is not pressed-in, so when you click on cell B5 the recording returned is:

```
Range("B5").Select
```

When the tool is pressed-in the recording is Relative, so when you click the cell below the active cell the recording returned is:

```
ActiveCell.Offset(1, 0).Range("A1").Select
```

So, you record specific cell selection using the Select method of the Range object and the Range property to specify the cell. Or you can record relative cell movement and selection using the Offset property and the Range property of the Range object.

For cell movement the Range("A1") expression is redundant and can be removed. For relative cell selection this Range property is more useful, the following recording means, starting one cell down from the active cell, select an area three columns wide by four rows deep. In other words, treat the offset from the active cell as position A1.

```
ActiveCell.Offset(1, 0).Range("A1:C4").Select
```

Move the active cell two cells to the right:

```
ActiveCell.Offset(0,2).Select
```

Extend the current selection two cells over to the right starting from the active cell:

```
ActiveCell.Range("A1:C1").Select
```

Excel takes Row Major order when using numeric references. Cell reference B20 is cell 20,2 in numeric Row-Column order. These are called R1C1 references.

Offset values are either positive or negative numbers. Positive values are Down and to the Right. Negative values are Up and to the Left. Always in Row-Column order.

Personal Macro Workbook

When you elect to record into your Personal macro workbook you create or use a hidden workbook file, Personal.xls that is stored in the XLSTART folder, the start up folder for Excel. Thus, macros in this workbook are available as soon as you start up Excel. You can always Copy and Paste VBA code from other modules to Personal.

Personal.xls is a standard workbook but with a hidden interface. Sometimes you need to Unhide it. Use the *Window*, *Unhide* command in Excel's worksheet environment. You need to save the macros stored in Personal.xls. You can not select a hidden workbook and therefore can not use File, Save. If you Unhide it and save it then it is saved as a visible workbook.

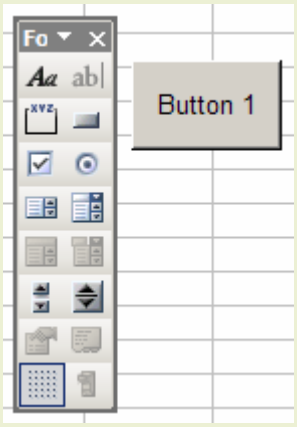
To save Personal.xls, either save it from the VB Editor or close down the Excel application and respond to the save files prompt. As an alternative to using Personal, see Creating an Add-In

Macro Buttons

You need an easy way of triggering your procedures from an Excel worksheet and Macro buttons are one of the most popular choices. You can either use the *Button* tool on the *Forms* Toolbar or the *CommandButton* tool on the *Control Toolbox* Toolbar.

The Button Tool is the old method inherited from earlier versions of Excel. The CommandButton Tool is the more modern method but takes longer to do.

The Button Tool

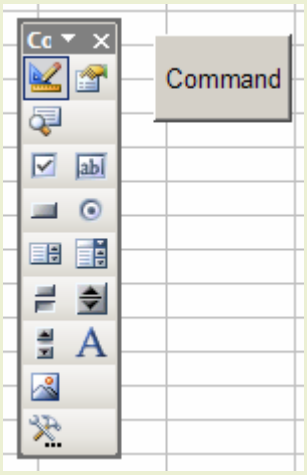


This is the good old-fashioned way of creating macro buttons and is usually the easiest.

1. View the Forms Toolbar.
2. Click the Button tool and draw a Button shape on the worksheet cells.
3. Choose the relevant macro from the list from the Assign Macro dialog.
4. Deselect the Macro Button

Right-click the Button to adjust its properties. This Button is a non-printing object by default.

The CommandButton Tool



The effect is the same but there is more to do.

1. View the Control Toolbox Toolbar.
2. Click the CommandButton tool and draw a Button shape on the worksheet cells.
3. Click the Properties button to adjust the Button's properties.
4. Click the View Code tool and fill-in the code for the button's Click event.
5. Click the Exit Design Mode tool to activate the Button object.

This Button is a printing object by default.

When you fill-in the click event code for the Command Button it is quite in order to enter a subroutine call to an existing procedure:

```
Private Sub CommandButton1_Click()
    Call MyMacro
End Sub
```

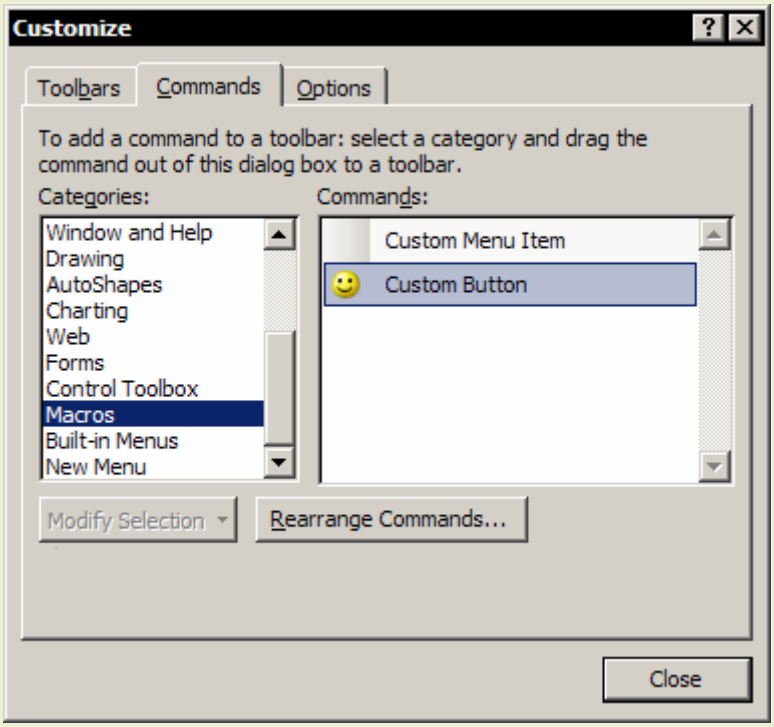
Command Bars

The alternative to the macro button is to assign your macro to a Command Object in Excel's menu structure or Toolbars. You can place a shortcut to run a macro or design a tool for a toolbar or even create your own command structure of menus and shortcuts. You have open access to the Excel command bar collection when you open the Customize dialog box. Choose *Tools, Customize* or right-click a visible toolbar.

Go to the *Commands* tab of the dialog and locate *Macros* in the *Categories* list.

Select one of the shortcuts and drag it onto a visible toolbar or menubar. Keep the *Customize* dialog open and then point to the shortcut that you have just dropped onto the toolbar and right-click.

Choose *Assign Macro* and having assigned your macro, right-click again to set any of the other properties; *Change Button Image* or *Edit Button Image* and indulge your creative urges. In a menu the ampersand character (&) before a letter nominates it as the accelerator key. Make sure you choose a unique letter for the menu.



If you want to have a toolbar that is stored with the workbook file, go to the *Toolbars* section of *Customize* and click *New* to make a new toolbar.

Populate the toolbar with shortcuts from the *Commands* section and then return to *Toolbars* and click *Attach* and attach the toolbar to the workbook.

Now you can send a copy of the workbook file and the toolbar is available to the recipient.

Editing and optimising recorded code

Recorded code is written by a program and can be rather unwieldy and difficult to read. Feel free to simplify your recordings and reduce them to the essentials.

Here is the original relative recording of entering XYZ down a column:

```
Sub RecordedXYZ()
      ActiveCell.FormulaR1C1 = "X"
      ActiveCell.Offset(1, 0).Range("A1").Select
      ActiveCell.FormulaR1C1 = "Y"
      ActiveCell.Offset(1, 0).Range("A1").Select
      ActiveCell.FormulaR1C1 = "Z"
End Sub
```

The recorder always uses the FormulaR1C1 property for data entry; here the Value property, the entry in the cell, is probably more appropriate. But you can leave out the property entirely as Value is the default property for a cell. The offsets have been simplified and the selections entirely removed.

```
Sub OptimisedXYZ()
      ActiveCell.Offset(0, 0) = "X"
      ActiveCell.Offset(1, 0) = "Y"
      ActiveCell.Offset(2, 0) = "Z"
End Sub
```

And here is the final version with meaningful comments and indentation:

```
'Enter XYZ down the column.
With ActiveCell
      .Offset(0, 0) = "X"
      .Offset(1, 0) = "Y"
      .Offset(2, 0) = "Z"
End With
```

There is no right way of writing code so allow your solution to follow your own thought process. There are two distinct styles: *Concrete* where the process follows the physical world, selecting cells and moving around, and *Abstract* which is a simpler style using numbers and indices. The recorded example is in the concrete style; type-in an entry, move down one cell, type-in another entry etc. The optimised version more abstract; write X into the current cell, Y in the cell below and Z into the cell below that.

Toggles

A Toggle is a statement that switches from one state to another and the standard construction can be applied to any Property that accepts a True / False value.

The following recorded statement turns off the display of Headings.

```
ActiveWindow.DisplayHeadings = False
```

After editing the statement now toggles the display of Headings.

```
ActiveWindow.DisplayHeadings = Not ActiveWindow.DisplayHeadings
```

Removing Selection statements

The most common and effective optimisation process is to remove the Selection statements from recorded macros. They are entirely unnecessary.

Recorded:

```
Columns("E:E").Select
Selection.Columns.AutoFit
Selection.Style = "Comma"
```

Optimised:

```
With Columns("E:E")
      .Columns.AutoFit
      .Style = "Comma"
End With
```

Common Tasks in Excel Macros

Printing

Here is an extract from a recorded macro to print a single range of cells in landscape. Unbelievable! Don't let this sort of recording put you off using the Macro Recorder, it is an invaluable tool. The Macro Recorder is not selective; it has recorded the state of every control in the Page Setup dialog. You just need to delete the unnecessary statements.

```
Sub Macro1()

      Range("C3:E8").Select
      ActiveSheet.PageSetup.PrintArea = "$C$3:$E$8"
      With ActiveSheet.PageSetup
            .PrintTitleRows = ""
            .PrintTitleColumns = ""
      End With
      ActiveSheet.PageSetup.PrintArea = "$C$3:$E$8"
      With ActiveSheet.PageSetup
            .PrintHeadings = False
            .PrintGridlines = False
            .PrintComments = xlPrintNoComments
            .PrintQuality = 600
            .CenterHorizontally = False
            .CenterVertically = False
            .Orientation = xlLandscape
            .Draft = False
            .PaperSize = xlPaperLetter
            .FirstPageNumber = xlAutomatic
            .Order = xlDownThenOver
```

```
            .BlackAndWhite = False
            .Zoom = 100
      End With
      ActiveWindow.SelectedSheets.PrintOut Copies:=1, Collate:=True

End Sub
```

The code for printing macros can be quite dramatically reduced.

This is all you need for printing:

```
Sub ConcisePrintMacro()

      'Print a range of cells.
      Range("A1:G250").PrintOut

      'Print the used range of the active worksheet.
      ActiveSheet.PrintOut

End Sub
```

Printing and Page Setup settings are like this:

```
Sub PageSetupSettings()

      'The PageSetUp object is a child of the worksheet,
      'not the range.

      With ActiveSheet.PageSetup
            .CenterFooter = "My Report"
            .RightFooter = "by Anon E. Mouse"
            .Orientation = xlLandscape
            .FitToPagesWide = 1
            .FitToPagesTall = 1
      End With

      Range("A1:G250").PrintOut

End Sub
```

You sometimes need to print out a named range of cells. To make Page Setup settings you need to identify the worksheet that owns the range. Use the Parent property of the range rather than making an explicit reference to the worksheet. The Parent property of an object points back up the containment hierarchy to identify the object above.

```
Sub IdentifyParentSheetOfNamedRange()

      Dim MyRange As Range
      Dim MySheet As Worksheet

      Set MyRange = Range("DataArea")
      Set MySheet = Worksheets(MyRange.Parent.Name)

      With MySheet.PageSetup
          .CenterFooter = "My Report"
          .RightFooter = "by Anon E. Mouse"
          .Orientation = xlLandscape
          .FitToPagesWide = 1
          .FitToPagesTall = 1
      End With

      MyRange.PrintOut

End Sub
```

Copying

This again, is a reduction of recorded code.

```
Sub RecordedCopyAndPaste()

      Range("C4:E11").Select
      Selection.Copy
      Sheets("Sheet2").Select
      Range("D7").Select
      ActiveSheet.Paste
      Application.CutCopyMode = False

End Sub
```

Try these instead:

```
Range("C4:E11").Copy    'No Selections required.
Range("C4:E11").Cut

Range("C4:E11").Paste   'This will fail, Paste is not supported.

Range("C4:E11").PasteSpecial 'But Paste Special is.

Range("C4:E11").Copy Destination:= Range("G10") 'One line of code.

[C4:E11].Copy [G10] 'Same as above, easier typing.

[Sheet1].[C4:E11].Copy [Sheet2].[G10] 'From Sheet to Sheet.

[B1] = [A1] 'An assignment statement; this copies the cell display…

[A1].Copy [B1] 'whereas this copies the formula.
```

Measuring areas and lists

Measure the dimensions of the current block of consecutive data:

```
a = ActiveCell.CurrentRegion.Rows.Count
b = ActiveCell.CurrentRegion.Columns.Count
```

Identify the coordinates of this range:

```
c = ActiveCell.CurrentRegion.Address
```

Measure the dimensions of the area containing data on a worksheet:

```
d = ActiveSheet.UsedRange.Rows.Count
e = ActiveSheet.UsedRange.Columns.Count
```

Identify the first used row of the worksheet:

```
f = ActiveSheet.UsedRange.Row
```

Identify the last used row of the worksheet:

```
g = Cells.SpecialCells(xlCellTypeLastCell).Row
```

Identify the next free row starting from A1:

```
h = Range("A1").End(xlDown).Row + 1
```

To select the block of cells containing the active cell:

```
ActiveCell.CurrentRegion.Select
```

Identify the first row and column in the block containing the active cell:

```
i = ActiveCell.CurrentRegion.Row
j = ActiveCell.CurrentRegion.Column
```

Select from cell C3 to the top of the current region:

```
Range("C3").End(xlUp).Select
```

Select from cell C3 to the last cell on the right in the current region:

```
Range("C3").End(xlToRight).Select
```

If your macros incorporate extensive moving and selecting you might consider creating a Move object to make your macros easier to create. See Creating a Move object

Locating data on a worksheet

Use the Special Cells Method to locate cells on a worksheet that have particular characteristics. Record this using *Edit*, *Goto*, *Special*.

The following procedure clears every cell in the workbook that contains a constant numeric value, leaving the text and the formulas intact.

```
Sub DeleteNumbers()
      Dim wksSheet      As Worksheet
      Dim rngNumbers    As Range

      On Error Resume Next

      For Each wksSheet In Worksheets

            'Identify numeric cells.
            Set rngNumbers = _
                  wksSheet.Cells.SpecialCells(xlCellTypeConstants, 1)

            'Delete cell values.
            rngNumbers.Clear

      Next

End Sub
```

Without using the SpecialCells Method the procedure would have been far harder to write requiring a loop to examine each worksheet cell and a conditional test to see whether the cell contained a number that was not a formula, as follows:

```
For Each Cell In ActiveSheet.UsedRange
      If Cell.HasFormula = False And IsNumeric(Cell) = True Then
            Cell.Clear
      End If
Next
```

Note that SpecialCells is a Method of the Range Object therefore the following line of code would fail:

```
ActiveSheet.SpecialCells(xlCellTypeConstants, 1).Select
```

You must return the Range Object:

```
ActiveSheet.Cells.SpecialCells(xlCellTypeConstants, 1).Select
```

or

```
ActiveSheet.UsedRange.SpecialCells(xlCellTypeConstants, 1).Select
```

Manipulating cells

The Cells property can replace A1 references or offsets to manipulate cells.

The Cells property returns the Range Object, every cell on the entire worksheet:

```
Cells.NumberFormat = "General"
```

Manipulating cells with R1C1 coordinates, using a loop counter to make cell references:

```
Sub CopyValues()
      For r = 2 To 100
            Select Case Cells(r, 1)
                  Case 1
                        Cells(r, 2).Copy Cells(r, 3)
                  Case 2
                        Cells(r, 2).Copy Cells(r, 4)
                  Case 3
                        Cells(r, 2).Copy Cells(r, 5)
                  Case 4
                        Cells(r, 2).Copy Cells(r, 6)
                  Case Else
                        Cells(r, 2).Copy Cells(r, 7)
            End Select
      Next
End Sub
```

This style of code is entirely abstract but very concise and direct. Notice how easy it is to get the idea of going down to the next row on a worksheet by using the incrementing counter variable of a For…Next loop rather than the clumsy Offset, Select statements.

You can display R1C1 references on an Excel worksheet by choosing *Tools*, *Options*, *General* Tab, *Settings*, *R1C1 Reference Style*.

Application Settings

Here are some useful Application Property settings that can speed up execution time. As the Application Object (Excel itself) is the top-level object you could enter these without using the Application object reference.

Switching between automatic and manual recalculation:

```
Application.Calculation = xlCalculationAutomatic
Application.Calculation = xlCalculationManual
```

Turning on/off the screen display:

```
Application.ScreenUpdating = True
```

or without the object reference:

```
ScreenUpdating = True
```

Suppress the display of confirmation messages:

```
Application.DisplayAlerts = False
```

Disable the ESC key:

```
Application.EnableCancelKey = xlDisabled
```

Block all input from the keyboard and mouse except for interactive elements displayed by the procedure:

```
Application.Interactive = False
```

Most of the above will need to be reset to their normal states at the end of the procedure. Be particularly careful with the Interactive property. Make absolutely sure that you set its value to True before the end of the procedure otherwise Excel will not accept any user input after the macro has been executed.

Using the Visual Basic Editor

The editor has several different user windows, if the one you need is not visible then open it using the VBE View menu. Rearrange and resize the windows as you wish.

Code Window

Press F7. This is where you view the code, the actual instructions contained in the Module. The Code Window has two views, Procedure View and Full Module View. Procedure View shows only one procedure at a time, Full Module View lists all the procedures in the module, separated by ruler lines. To change the view, use the buttons situated at the lower left-hand corner of the window.

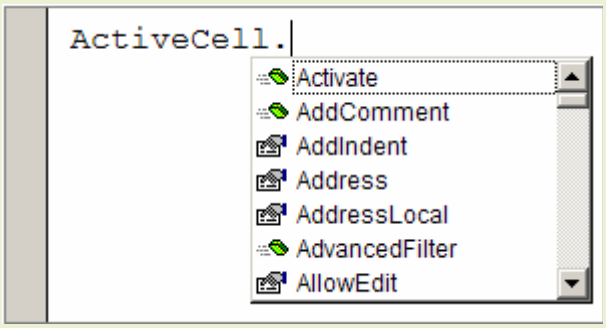
At the top of the window are two drop-down lists, the one on the right-hand side is the Procedure List. Use this to navigate from one procedure to another. Or use the keyboard shortcuts Ctrl+PgUp/Ctrl+PgDn. The left-hand list is the Object list.

You will also notice the standard code colours: Blue for Keywords, Green for Comments and Black for everything else. Try not to change the colours unless the Blue is indistinct from the Black on your monitor or if you suffer from Red/Green colour vision problems. Change the colours using *Tools*, *Options*, *Editor Format* Tab.

Context Help

To look up the relevant page in the documentation, click an expression in the Code Window and press the F1 key.

Complete Word



One of the most useful features of the Code Window is Complete Word. These are drop-down Autolists that enable statement completion showing those Objects, Methods, Properties and Events that are available in context. The lists appear as soon as you start typing. To accept an item from the list and stay on the same line to continue your statement, press TAB.

The lists significantly reduce the number of typing errors. To start up the lists without typing an initial expression, press Ctrl+Space. Or right-click the relevant line and choose Complete Word.

Commenting/ Uncommenting

The apostrophe is the Remark character, *remarks* or *comments* are entirely ignored when the code is run. Comments are used for explanation and annotation of the process code. Comments can be entered at any position in the Module. There is no end comment character; everything following the apostrophe is a comment.

Every procedure should have at least one comment. Code is updated and revised periodically during its lifetime. It is very difficult trying to interpret uncommented code.

You can add a comment at the end of the code line; you do not need to start a new line.

```
Selection.NumberFormat = "0.00" 'Set number format
```

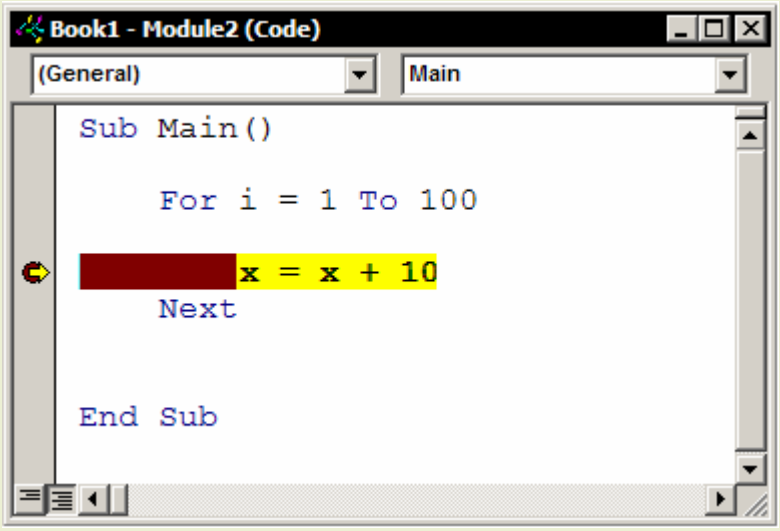
Commenting out is a technique where sections of code are temporarily disabled for testing purposes only to be reinstated once the testing is completed. It is extremely tedious to comment out each separate line. You will find the *Comment Block* and *Uncomment Block* Tools on the *Edit* Toolbar.

Running and Stepping Into statements

You either Run your code at normal speed or you Step Into it one statement at a time in Break Mode. There are many variations on the theme of Stepping, look at the Debug

menu. The fundamental shortcuts are F5 for Run and F8 for Step Into, click the body of the procedure first to set the context. The Run Tool is on the Standard Toolbar. The Step Into Tool is on the Debug Toolbar.

Breakpoints and Break Mode



A Breakpoint is a line of code that you set as being the point at which Excel switches from Run Time to Break Mode.

It is helpful to set Breakpoints when you do not want to Step through the entire Procedure, just trace a few commands.

Press F9 to set the Breakpoint, press F5 to run to the Breakpoint and then press F8 to Step through the code.

The easiest way to set or remove Breakpoints is to click on the left-hand grey margin of the code window. Break Mode is when you can see the Yellow indicators, Reset to return to Design Mode. The Reset Tool is on the Standard Toolbar.

Errors

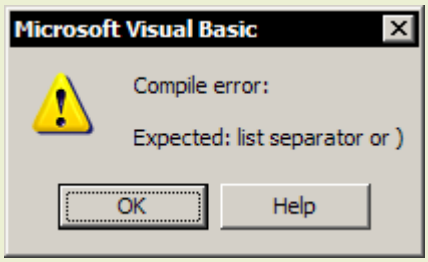
Unless you can record all your macros you will always get some kind of error as you develop your code. There are three types of error, *Logical errors*, *Syntax errors* and *Run-Time errors*. Choose *Debug*, *Compile* the Project to check for errors.

A Logical error is where the code does not fail but does not do what you wanted. You will always get an error message for the other types of errors. Syntax errors are coloured red. Run-Time errors do not arise as you type-in your code, only when you run the procedure. Always Debug a Run-Time error. The Debug Button switches the Module to Break Mode and identifies the statement that caused the error. It does not correct the statement. The entire Module is compiled when you run a procedure, the Run-Time error is not necessarily in the current procedure. Reset when you have fixed the error.

Syntax Errors

Syntax error, clearly there is something wrong:

```
Selection.SpecialCells(xlCellTypeVisible select
```



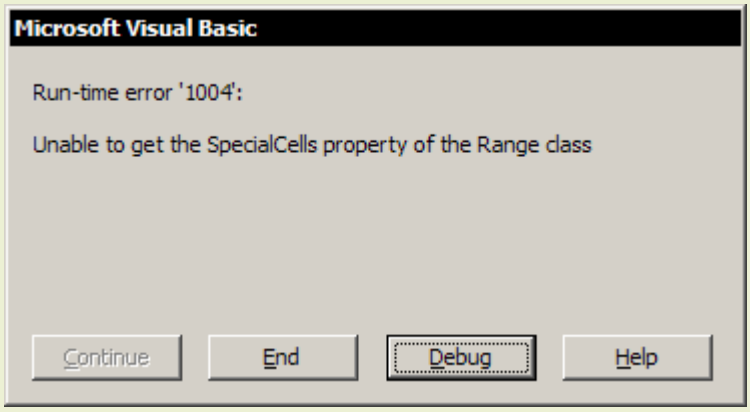
A syntax error is usually a minor error in typing or construction; comma missing, brackets not closed etc. Syntax errors rarely cause serious problems.

Syntax errors are coloured red.

Run-Time errors

Run-Time error, there is something wrong but it is not obvious:

```
Selection.SpecialCells(x1CellTypeVisible).Select
```



Did you spot the error? It should read xlCellTypeVisible, not x1CellTypeVisible, a lower case alphabetical el, not a number 1. The Courier font is notoriously indistinct for these two characters and this is a classic Excel Run-Time error. So many Run-Time errors are just typos; try to avoid them by using the Complete Word lists as much as possible.

Line Continuation

Some statements are rather lengthy and difficult to read on one line. Do not press enter to wrap the text; this just produces a syntax error. To continue the same logical line onto the next physical line, introduce a line continuation character into your code.

Use the following sequence of keystrokes for a line continuation character; Spacebar, Underscore, Enter. It is a sequence, not a key combination. You can have as many line-continuations as you require. Second and subsequent lines can be tabbed.

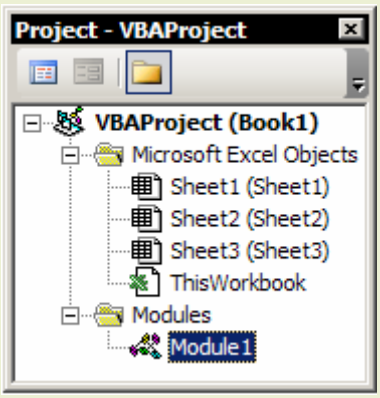
Statements like this can be rather difficult to read:

```
ActiveWorksheet.Cells.SpecialCells(xlCellTypeVisible).Select
```

Statements are much easier to read with line continuation characters:

```
ActiveWorksheet.Cells. _
            SpecialCells(xlCellTypeVisible). _
            Select
```

Project Explorer Window



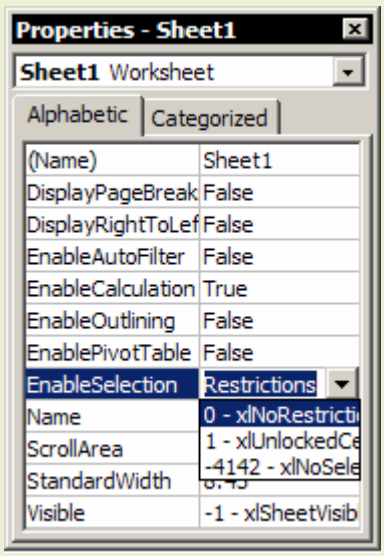
Press Ctrl+R. This window exposes the objects of each open Project. If you want to change the name of an object, select it in this window and enter a new name in the Properties window. To delete an object, using the delete key has no effect, right click the object and *Remove* it. You can Drag and Drop a Module from one Project to another.

To insert a module into the project without using the recorder, either use the Insert menu or right click the relevant project. Do not double-click one of the worksheet objects, this gives you entirely the wrong type of module, an Object Module! You will have nothing but trouble if you use one of these to contain General code. You want a General module in the Modules collection.

Object Modules look identical to General Modules but their inadvertent use can cause errors that are hard to detect. For example, a simple statement like, *Range("A1").Select* entered in the Sheet1 Object Module would only work on Sheet1. It would cause a Run-Time error if the code were run on any other sheet in the workbook.

Properties Window

Press F4. The Properties Window is where you set the variable properties of objects. Not of much use for developing code in General modules. It is extensively used when designing graphical User Forms.

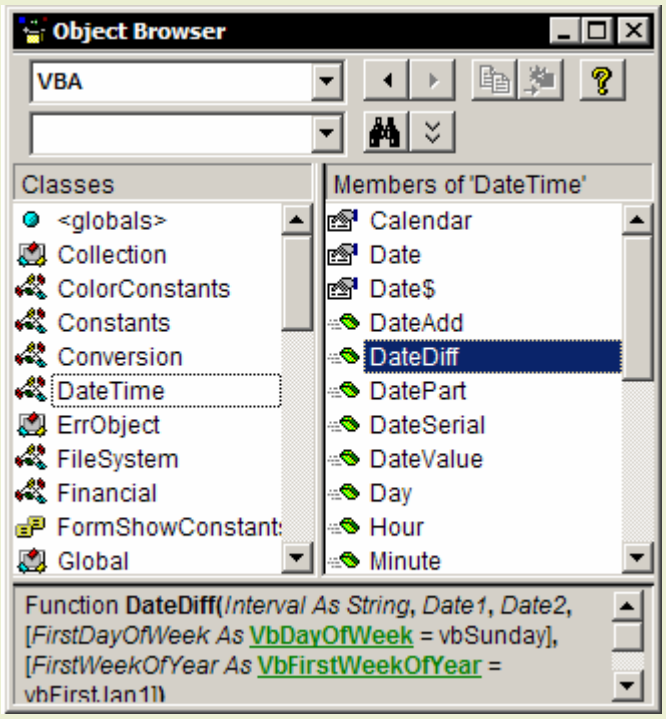


The Properties Window can be irritating for the first few occasions when you use it. It always displays the variable properties of the active selection and you may find yourself looking down the listings in the property pages and being unable to find the property that you are looking for. Check the active selection, it is so easy to change the selection and not realise it.

Most of the property page values can be selected from drop down lists but sometimes they have to be typed-in. To register a typed value, either press ENTER or click another cell on the property page. Do not click outside the window as this usually just changes the selection.

Note that there are two tab sections which classify all the properties; Alphabetic and Categorised.

Object Browser



Press F2. If the macro recorder is the phrasebook for VBA then the Object Browser is the dictionary.

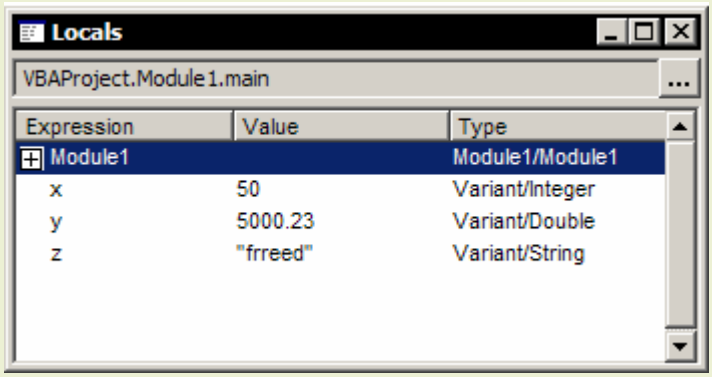
All references are listed here. Choose the top drop-down list to reference the relevant library, Excel or VBA?

If you have an idea of the name of something that you are trying to look up, enter an expression into the Search box below to perform a freeform search of the database.

If you just want to see a full listing of what is available then choose an item from the Classes list on the left hand side and examine the Members list on the right hand side.

It can take some time to find what you are looking for in the Object Browser but there is no alternative if you do not know its name.

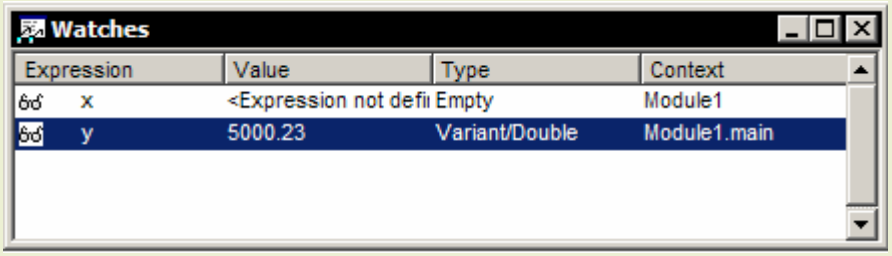
Locals Window



This window is used for viewing the current values of all the variables currently in Scope.

Step Into your code and see the exact state of any variable at any point in the procedure.

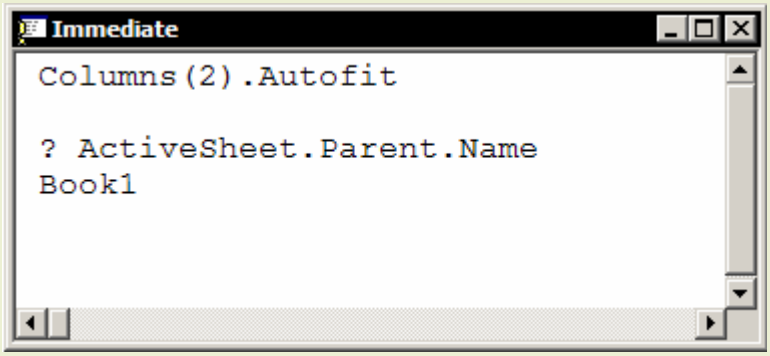
Watch Window



The Watch Window is similar to the Locals Window but is used to view the current values of only certain, nominated variables.

You need to specify the Watch expressions using Add Watch on the Debug menu.

Immediate Window



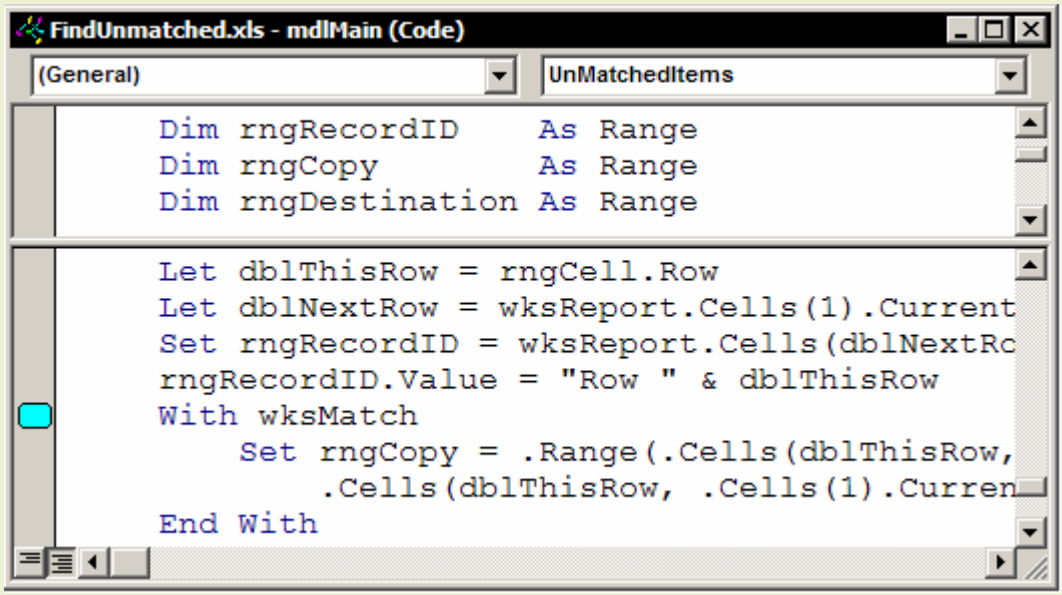
Press CTRL+G. The Immediate Window is used for immediate execution of a single expression. Type the expression into the Window and press enter to execute it.

If the expression returns a value then it needs to print the result to the Window. In this case you should precede the expression with a question mark.

Use this Window to experiment with statements. You can write a log to the Immediate Window by including Debug.Print statements in your procedure.

Splits and Bookmarks

It can be rather difficult to navigate your way through all the lines of code in a long and complicated set of procedures. Use the window split bar to display your declarations section as you write the code, you are far more likely to remember to declare your variables if you can see them. *Bookmarks* are blue indicators that can be used to mark positions in the code; this makes the process of returning to a specific point in a procedure far easier than having to scroll through multiple lines.



Set a bookmark by choosing *Edit*, *Bookmarks*, *Toggle Bookmark*. Then choose *Next Bookmark* and *Previous Bookmark* to navigate. Bookmark shortcuts are on the *Edit* Toolbar.

VBA Memory Variables and Constants

The role of Option Explicit

You can use implicit variables in VBA by just typing-in an identifier and assigning a value to it. However you will not be able to do this if the Option Explicit statement is present.

Option Explicit forces you to declare your variables before you can use them. It is used to improve the execution speed and precision of the code. You can delete Option Explicit if you wish and continue with implicit variables. Otherwise you must declare.

To include the Option Explicit statement on all future modules:

1. Choose *Tools*, *Options*, *Editor* Tab.
2. Check the *Require Variable Declaration* checkbox.

Variable Declaration

The Dim statement is used to declare variables either in a single line or listing form. Explicitly declared variables are available in the Complete Word lists. Dim is short for Dimension (which makes no particular sense unless the variable is an array, a variable that can have more than one dimension). You can place the Dim statement anywhere in the procedure, so long as you declare the variable before you use it. It is a convention to list declaration statements at the start of the procedure.

```
Dim x, y, z
```

or

```
Dim x
Dim y
Dim z
```

```
Option Explicit

Sub Main()
    Dim x, y, z

      x = 50
      y = 100
      z = Application.Average(x, y)

      MsgBox z
      'View variable values in the Locals Window.
End Sub
```

Data Types

You can also declare the Type of data you intend to store within a Variable or Constant. This will ensure you use only the memory required to hold the data and validate the data. It will also cause problems if you do it incorrectly.

If you do not specify a data type, the Variant data type is assigned by default.

The Data Type is declared in the same statement as the variable or constant itself.

```
Dim MyVar   As String
Const MyNum As Integer = 5
```

You can also use one declaration statement for several variables:

```
Dim MyVar As String, MyNum As Integer
```

However, when using a single declaration statement, you must declare the Data Type for each variable. In the following example, only one variable has a defined Data Type, the other is Variant.

```
Dim MyVar, MyNum As Integer
```

Summary of Data Types

Type	Size	Stores
Boolean	2 bytes	True or False values.
Byte	1 Byte per character	Unsigned whole numbers 0 to 25
Integer	2 bytes	Whole numbers ± 32,768
Long (Integer)	4 bytes	Whole numbers ± 2,147,483,648
Single	4 bytes	Numbers ± 1.401298E-45 to 3.402823E38
Double	8 bytes	Numbers ± 1.79769313486232E308 to 4.94065645841247E-324
Currency	8 bytes	Numbers ± 922,337,203,685,477.5808
Decimal	14 bytes	± 79,228,162,514,264,337,593,543,950,335 with no decimal point. ± 7.9228162514264337593543950335 with up to 28 decimal places.
Date	8 bytes	Date values ranging from January 1, 100 to December 31, 9999.
Object	4 bytes	Any Object Reference.
String	1 byte per character	Text data.
Variant	Varies	Anything, Variant is a chameleon data type where any value is stored. Variant is the default data type.

If you do declare the Data Type, make sure that you do so correctly. It is quite easy to determine what your Data Type should be so long as you allow the VB compiler to do it for you. See How to determine the Data Type below.

In the following example you can see the sort of problems caused when the Data Type is declared incorrectly. Three variables are declared, all as Integers. The variable x causes an Overflow error; the value is too large to be stored. The variable y does not fail but is stored as 2, not 1.5; integers are whole numbers. The variable z causes a Type mismatch error; the value to be stored is a String, a text value.

```
'Declaration:
Dim x As Integer, y As Integer, z As Integer

'Initialisation:
x = 50000
y = 1.5

MsgBox y

z = "Fred"
```

How to determine the Data Type

1. Write your code and initialise your variables to the sort of values that you will be storing in them.
2. Open the Locals Window. *View*, *Locals*.
3. Press F8 and Step Into the code to execute your initialisation statements.
4. Look at the third column in the Locals Window, headed Type.
5. You will see that your variables were declared as Variants and then internally coerced to a specific Data Type.
6. Declare the Data Type using these coercion data types.

Variable Scope and Lifetime

The Scope and Lifetime of a Variable or Constant is its visibility to other procedures and how long its value lasts. There are three levels of Scope: Public Module, Private Module and Procedure.

The Scope is set by the nature and position of the Declaration statement. The Module's Declarations section is at the top of the Module before the first procedure. It is very bad practice to declare two variables with the same identifier at different levels of scope.

Public Module Scope

A variable / constant with a Public Scope can be utilised by any procedure in any of the modules within that Project. Use a Public statement in the Declarations section.

```
Public MyVariable
```

Private Module Scope

A variable / constant with Module level scope can be used by all procedures within that particular Module. Use a Dim statement in the Declarations section.

Procedure Scope

A variable / constant with Procedure level scope is not available to any other procedure within the Module unless it is passed in a subroutine call. Use a Dim statement in the Sub.

```
'Dim before the Sub. Module level.
Dim MyVariable

Sub Main()

    'Dim after the Sub. Procedure level.
    Dim MyOtherVariable

   MyVariable = 50
   MyOtherVariable = 100

End Sub
```

```
Sub Main2()

   'Only MyVariable is in Scope.
    ActiveCell.Value = MyVariable

End Sub
```

Public and Private

Procedures can also be declared as Public or Private, they are Public by default. A Public procedure is accessible to all other procedures in all modules in the Project, whereas a Private procedure is accessible only to other procedures in the module where it is declared.

It is good practice to declare subroutine procedures as Private. This will clean up the clutter of macro names in the Macros box and provide only a single entry point into a set of related procedures.

```
Private Sub MyMacro()
```

When to use Set

Storing the reference to an Excel object in a variable is quite a different concept to storing values. Here you are creating an *alias* or a *shortcut* or a *pointer*. Various terms

are used for this process. VBA calls this an *Object Variable*. You must use the Set keyword to initialise an Object Variable. Do not use Set for any other purpose.

```
Sub ObjectVariables()

      Dim x As Integer
      Dim c As Range

      'Here you could use the Let keyword,
      Let x = 250
      'But it makes no difference either way.
      x = 250

      'Here you must initialise with the Set
      'keyword. The variable, c now can be used
      'in the code as a substitute for ActiveCell.

      Set c = ActiveCell

      'Without the use of Set this line of code would fail.
      c.Offset(1,0).Select

End Sub
```

The use of the Let keyword in assignment statements is a matter of personal style. Some authors like to use it as it explicitly shows that the statement is a variable assignment statement. However, there is a fundamental difference between Let and Set.

```
Set x = Range("A1:D25")
```

This statement creates the Object Variable, x that can then be used as an alias for the cell range. Any actions carried out on x are immediately reflected back to the cells.

```
Let x = Range("A1:D25")
```

This statement creates the Array Variable, x which stores the current values of the cells in memory. Any actions carried out on x do not affect the cells. The cell values and the variable values are entirely separate entities.

Declaring the Data Type of Object Variables

There is no specific requirement that you do explicitly declare the Data Type of an object variable, you can just leave it out. However, if you do then you have a choice, either use the generic type, Object or be more precise and identify the Class of the Object. (You do not need to know about Classes when using Excel VBA but you often see the term being used. For further information see Classes)

```
Sub ObjectDataTypes()
      Dim PrimoSheet    As Worksheet
      Dim SecundoSheet  As Object
      Dim MyRange       As Range

    'Here we can be specific, this must be a Worksheet.
    Set PrimoSheet = Worksheets(1)

    'Sheets(2) might not be a worksheet, use generic.
     Set SecundoSheet = Sheets(2)

    'This could only ever be a Range.
     Set MyRange = [A1:Z500]

End Sub
```

When you have finished using the Object Variable in your code but the procedure is going to continue to execute statements you can release the memory allocated to the variable and destroy the object variable by setting its value to Nothing.

```
Set MyObjVar = Nothing
```

Use of Constants

A Constant is a value in a procedure that does not change. Constants are similar to Variables; the key difference is that the values of variables can change during execution, whereas the values of constants are fixed.

Unlike variables, constants are both declared and initialised in one statement.

```
Sub ConstantsVsVariables()

      'Declaration.
      Dim USD As Currency
      'Initialisation.
      USD = 1.80

      Const USD As Currency = 1.80
      Const PAYMENT_TERMS = 30

End Sub
```

Data Type Conversion Functions

CBool	Boolean
CByte	Byte
CCur	Currency
CDate	Date
CDbl	Double
CDec	Decimal
CInt	Integer
CLng	Long
CSng	Single
CStr	String
CVar	Variant

Sometimes it is not possible to explicitly set the data type of a variable as the value and data type of the variable is unknown at the point of declaration.

Declare the variable as type variant and when the value has been acquired and validated then you can convert the variable to the correct data type using a conversion function.

For example:

```
Dim USD As Variant

USD = CCur(USD)
```

Naming Conventions

A one or three character lower case prefix is commonly added to variables as a document convention. Variable identifiers are then readily recognised in the code and are easier to enter; explicitly declared variables are available in the Complete Word drop-down lists.

```
Sub NamingConventions()

    Dim strProductName  As String
    Dim intCounter      As Integer
    Dim rngRange        As Range

End Sub
```

Should I declare my variables?

All the pundits scream, "Yes!" But be realistic, if your procedure is short and uses only one or two variables then you have little to gain, what are the chances of mistyping x and y? But what if the procedure contains a counter loop that iterates hundreds of times? In the long run code with explicitly declared variables of the correct data type will execute faster than code with implicit variables and it will be much easier to debug and maintain. But it takes longer to write and there are pitfalls for the unwary.

As a general rule of thumb, it is usually best to keep Option Explicit in the declarations section. Use it when appropriate and delete it when it is not. Always properly declare and

type your variables for long and difficult procedures, you will end up doing so in any case once you run into a few problems! For short macros though, it is barely worth the effort.

See also Using Arrays to store sets of data
See also User Defined Data Type
See also Enumerations

Functions

Function procedures accept, manipulate and then return values. They can be used in conjunction with Sub procedures to perform utility tasks in your code and perform in a similar manner to subroutine calls.

More commonly in Excel, Function procedures are used to bundle complex calculations into a central procedure or to design user-defined functions. You do not run Function procedures; they are called. In VBA code a function is called in the same way as a VBA function. In worksheet cells a function is called in the same way as an Excel function

In your code you can call existing functions from VBA, see Calling VBA Functions and also Excel Worksheet Functions, see Calling Excel Functions

You can also write Function Procedures in VBA to interact with your Sub procedures, see, Creating Function Procedures or as User Defined Functions for use in Worksheet formula expressions, see Creating a custom function for Excel

Calling VBA functions

Use the Object Browser to see a list of all of the VBA functions. VBA functions have a simple syntax structure as follows:

```
FunctionName(Arguments)
```

Unlike Excel worksheet functions, the parentheses are only required if there is an argument value, for example to return the current date and time:

In Excel, =NOW() In VBA, `Now`

The Format function

Expression	Format Code	Transformation
-5000	"#,##0.00_);(#,##0.00)"	(5,000.00)
5000000	"0,,.0 million"	5.0 million
Month(Date)	"00"	12
Month(Date)	"MMMM"	December
Date	"DDDD"	Thursday
34 / 5000	"0%"	0.7%

This is the Function that has a thousand uses, "if only I had known about it six months ago…" Use the Format Function to transform any numeric value. Although they differ in detail the fundamental number format codes for Excel and VBA are identical. To find the relevant code values look up Custom Number Formats in Excel Help or the Format function in VBA Help.

```
Format(Expression,"Format Code")
```

Calling Excel Worksheet Functions

Excel functions are members of the WorksheetFunction Collection. You can call any Worksheet function in the module but you must identify it as being exclusively an Excel function by including the Application object reference otherwise the call will fail.

The shortcut is to just access the Application object.

```
x = Application.Average( y, z)
```

The full reference is more efficient and shows all the functions in the Complete Word list.

```
x = Application.WorksheetFunction.Average( y, z)
```

Creating a Function procedure

Write a Function procedure in a module using exactly the same methods as a Sub procedure. For example, the FileExists function illustrated is a utility procedure to validate file names; which can be called by any other procedure.

```
Sub FileOpeningRoutine()
      Dim sFileName As String

      sFileName = "Basic.xls"

      'Open the file if the file exists.
      If FileExists(sFileName) Then
            Workbooks.Open FileName:=sFileName
      End If
End Sub
```

```
Function FileExists(sFileName As String) As Boolean
      'Accepts : File Name as String.
      'Returns : TRUE if the file name is good.

      If Dir(sFileName) = "" Then
            FileExists = False
      Else
            FileExists = True
      End If
End Function
```

The two-line Accepts and Returns comments are the standard document convention for all Function procedures.

Creating a Custom Function for Excel

A user-defined function is an excellent method of centralising a specialised or complex calculation so that it can easily be entered into worksheet cells. You use the functions in formula expressions as you do with normal Excel functions, filling-in the arguments with cell references. The function returns the manipulated values to the cell.

```
Function VAT(Number)
      'Accepts : Value from a worksheet cell.
      'Returns : VAT @ 17.5%.

      Const RATE = 0.175
      VAT = Number * RATE
End Function
```

Excel formulas have to use awkward linear conditional statements whereas VBA has superior structures. It is far more efficient to use a simple expression in your worksheet cells to call a complex calculation than it is to have multiple copies of a complex formula.

```
Function RATIO(First_Number, Second_Number)
    'Accepts : Values from two worksheet cells.
    'Returns : Simple ratio to 0 decimal places.

    x = First_Number
    y = Second_Number

    If x = y Then
            RATIO = "Parity"
    ElseIf x > y Then
            RATIO = Round(x/y, 0) & ":1"
    Else
            RATIO = "1:" & Round(y/x, 0)
    End If

End Function
```

To call the RATIO function you would enter the following expression into a worksheet cell, replacing the argument names with cell references:

=RATIO(First_Number, Second_Number)

However, to call the function from a cell in another workbook you would need an external reference to the workbook containing the function procedure:

=BookName!RATIO(First_Number, Second_Number)

Creating an Add-In

To make a Function or a Sub procedure global and visible to all workbooks make an Excel Add-In. This is a compiled version of the code in the file that is loaded automatically as the Excel application is opened. To make an Add-In:

Step 1 (Optional)

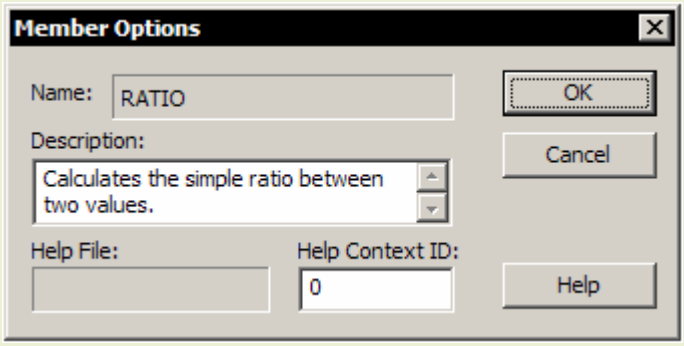
Document the Add-In. Should you skip this step when you create an Add-In only the Name of the Add-In file is shown in the Add-In Manager list and there is no Help documentation in the Paste Function dialog box.

Documenting the Add-In in the Add-In Manager listing:

Worksheet Menu: Attach Summary properties to the file. *File*, *Properties*, *Summary* Tab, fill-in *Title* and *Comments* (these are used for the Caption and Description text in the Add-In Manager list)

Documenting the function procedure in the Paste Function dialog:

Visual Basic Editor: Open the Object Browser window and choose VBAProject from the All Libraries drop-down list.



Examine either the listing held under the Globals object or the Module and you will see the function procedure listed as a method.

Select the method, right-click and choose Properties from the short-cut menu. Fill-in the Description box in the Member Options dialog.

Step 2

Create the Add-In file.

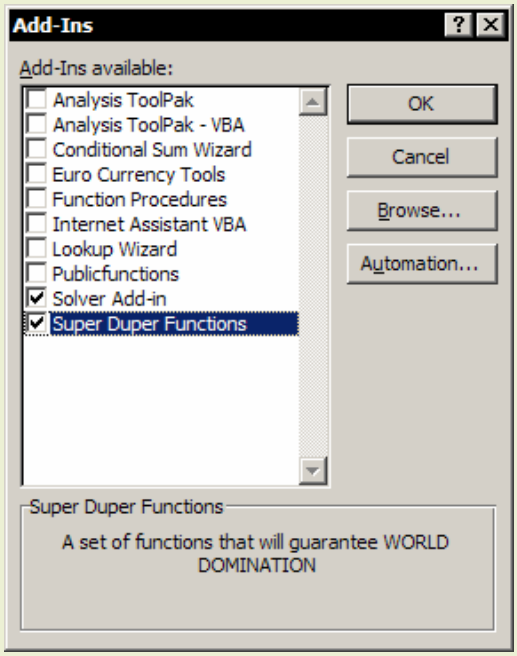
Worksheet Menu: *File*, *Save As*, *Save As Type* = Excel Add-In *.xla (at end of the list)

Step 3

Set the Add-In Manager to load the Add-In file as Excel starts up.

Worksheet Menu: *Tools*, *Add-Ins*, *Browse*. Select the .xla file from the file listings. Make sure that the check box is checked.

All the procedures in the current project are included in the Add-In. Add-Ins can contain Subs or Functions or both.



Protecting a Project

Even the code for an Add-In is available in the VB Editor so to prevent tampering, lock the Project with a password. Choose *Tools*, *VBA Project Properties*, *Protection* Tab.

Events

The role of event driven procedures

Worksheet and Workbook events trigger your code automatically when a specific event occurs, such as opening a file or recalculating a worksheet. The Event procedures already exist as code shells. All you need to do is find them and fill in the shell.

Using the event code shells

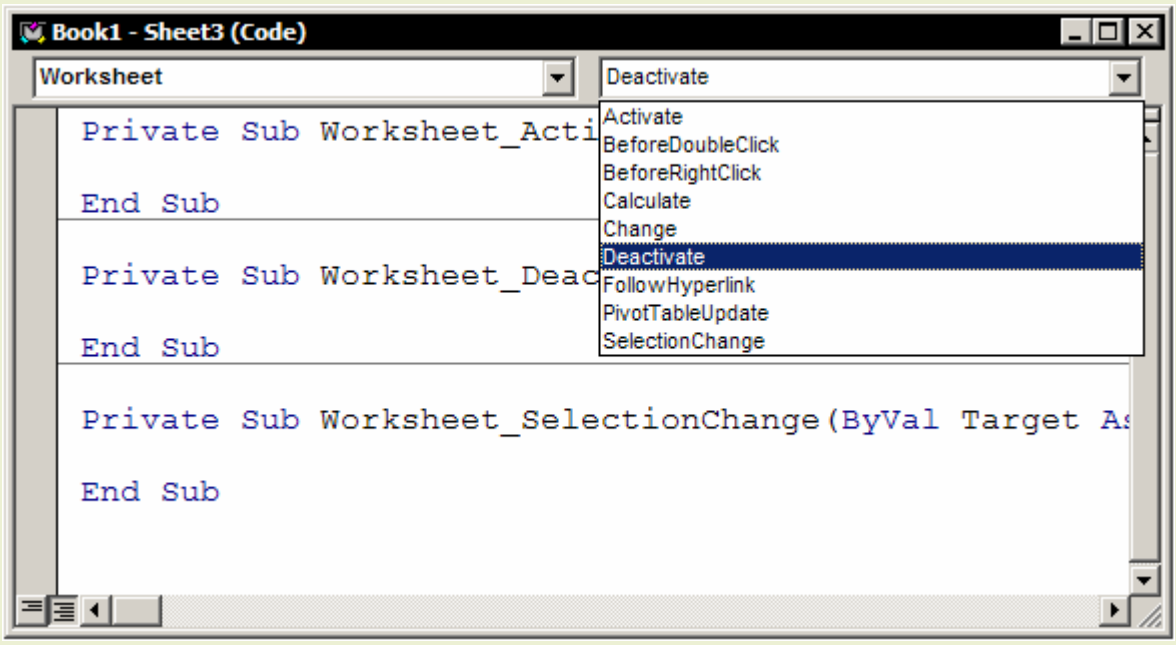
To use a Workbook Event:

1. Open the Module for ThisWorkbook in the Microsoft Excel Objects section of the Project Explorer Window. Click View Code or press F7 or double-click.
2. Select Workbook from the Object drop down list. (left-hand side)
3. Select the Event from the Procedure drop down list. (right-hand side)

To use a Worksheet Event:

1. Open the Module for the required worksheet in the Microsoft Excel Objects section of the Project Explorer.
2. Select Worksheet from the Object drop down list. (left-hand side)
3. Select the Event from the Procedure drop down list. (right-hand side)

Or right-click the worksheet tab in the normal Excel workspace and choose, *View Code*.



To disable the automatic execution of the Workbook_Open event, hold down the SHIFT key as you open the file.

Reserved Procedure Names

You can use the reserved procedure names, Auto_Open and Auto_Close as an alternative to using object events for automatic execution. The spelling of the reserved name must be precise and include the underscore. These names are used for procedures in General Modules. The object's Open event is precedent to an Auto_Open procedure. The following procedure displays the message box automatically when the file opens.

```
Sub Auto_Open

      MsgBox "Hello"

End Sub
```

On Methods

These are Methods of the Application object and have the same effect as events but are implemented in a different way. You need two procedures: one to schedule the event in the memory, the other is the procedure that is called when that event occurs.

OnKey Method

Runs a specified procedure when a particular key or key combination is pressed.

This example assigns My_Procedure to the key sequence CTRL+PLUS SIGN and assigns Other_Procedure to the key sequence SHIFT+CTRL+RIGHT ARROW.

```
Application.OnKey "^{+}", "My_Procedure"
Application.OnKey "+^{RIGHT}", "Other_Procedure"
```

This example returns SHIFT+CTRL+RIGHT ARROW to its normal meaning.

```
Application.OnKey "+^{RIGHT}"
```

This example disables the SHIFT+CTRL+RIGHT ARROW key sequence.

```
Application.OnKey "+^{RIGHT}", ""
```

OnTime Method

Schedules a procedure to be run at a specified time in the future (either at a specific time of day or after a specific amount of time has passed).

This example runs My_Procedure 45 seconds from now:

```
Application.OnTime Now + TimeValue("00:00:45"), "My_Procedure"
```

This example runs my_Procedure at 5 P.M:

```
Application.OnTime TimeValue("17:00:00"), "My_Procedure"
```

This example cancels the OnTime setting from the previous example:

```
Application.OnTime EarliestTime:=TimeValue("17:00:00"), _
                   Procedure:="My_Procedure", Schedule:=False
```

The following procedures are stored in Personal.xls and guarantee that you will not forget to go home on time:

```
Sub Auto_Open()

    Application.OnTime TimeValue("17:30:00"), "HomeTime"

End Sub
```

```
Sub HomeTime()

      MsgBox "Get your coat! It's Home Time."

End Sub
```

User Interaction

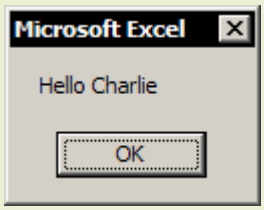
Message Box

The MsgBox function can used in either its Statement or Function forms.

Statement form

This is the simplest form, used for non-interactive messaging. You do not need parentheses around the arguments:

```
MsgBox "Hello Charlie"
```



The prompt text in the message does not wrap onto a new line in the box until the character count reaches 160; meanwhile the box just gets wider with the text on one line. Use any one of the following constant values to force a new line:

```
Chr(10),Chr(13), vbCrLf, vbCr, vbLf
```

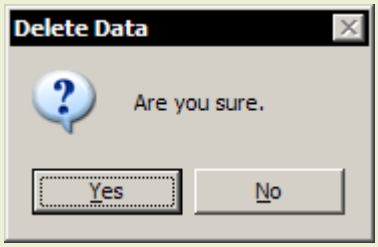
(This is quite a different idea to code line continuation using Spacebar, Underscore, Enter. This is for forcing new lines of text in Message Boxes and Input Boxes)

Forcing new lines in the prompt:

```
MsgBox "Hello Charlie," & vbCrLf & "have a nice day."
```

See overleaf for a discussion of the arguments accepted by the MsgBox function. See By Name, By Order for instructions on how to specify them.

Function Form



You must use the Function form when you are interacting with the user. You need to store their response. The Function form requires the arguments in parentheses; you are entering an assignment statement.

The message box returns a result based on which button as clicked then this returned result is evaluated.

```
Sub Main()

    Dim iAns As Integer

      iAns = MsgBox ("Are you sure.", _
                  vbYesNo + vbQuestion, Title:="Delete Data")

      If iAns = vbYes Then
            MsgBox "Data will now be deleted.", _
                              Buttons:=vbInformation, _
                              Title:="Delete Data"

      Else
            MsgBox "Data will not be deleted.", _
                              Buttons:=vbExclamation, _
                              Title:="Data Retained"
      End If

End Sub
```

MsgBox Buttons and Return values

The Buttons argument is optional and is a numeric expression that is the sum of the values specifying the number and type of buttons to display, the icon style to use, the identity of the default button, and the modality of the message box. The default value for the buttons argument is 0.

Buttons

Constant	Value	Description
vbOKOnly	0	OK button only
vbOKCancel	1	OK and Cancel buttons
vbAbortRetryIgnore	2	Abort, Retry, and Ignore buttons
vbYesNoCancel	3	Yes, No, and Cancel buttons
vbYesNo	4	Yes and No buttons
vbRetryCancel	5	Retry and Cancel buttons
vbCritical	16	Critical Message icon
vbQuestion	32	Warning Query icon
vbExclamation	48	Warning Message icon
vbInformation	64	Information Message icon
vbDefaultButton1	0	First button is default
vbDefaultButton2	256	Second button is default
vbDefaultButton3	512	Third button is default
vbDefaultButton4	768	Fourth button is default
vbApplicationModal	0	Application modal
vbSystemModal	4096	System modal

The values from 0 to 5 describe the number and type of buttons displayed.

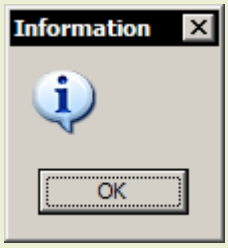
The second group; 16, 32, 48 and 64 describe the icon style.

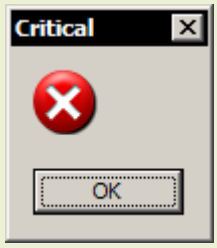
The third group; 0, 256, 512 and 768 determine which button is the default.

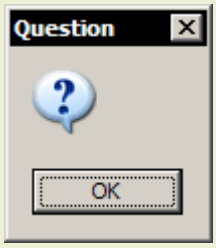
The fourth group; 0 and 4096 determine the modality of the message box.

Application modal; the user must respond to the message box before continuing work in the current application or *System modal*; all applications are suspended until the user responds to the message box.

When you are adding numbers to create a final value for the buttons argument you should use only one number from each group. You can use either the numbers or the constants. For example, to specify the display of a Yes and a No command button and a question mark icon the expression is either 4+32, or 36 or vbYesNo+vbQuestion.







Return Values

Constant	Value	Description
vbOK	1	OK
vbCancel	2	Cancel
vbAbort	3	Abort
vbRetry	4	Retry
vbIgnore	5	Ignore
vbYes	6	Yes
vbNo	7	No

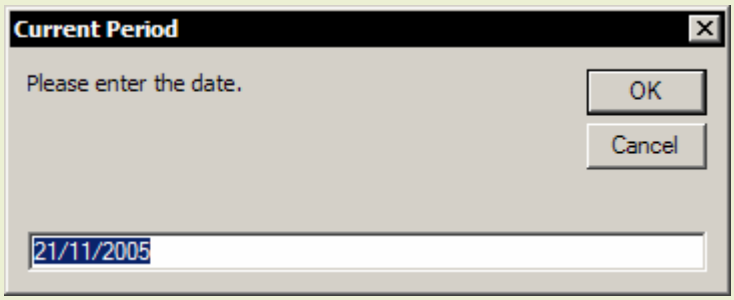
The return value is only generated when the MsgBox function is used in its function form and depends upon which command button was clicked when the message box was dismissed.

You can use either the value or the constant in your code to determine which button was clicked.

Input Boxes

You can use either the generic VBA Input Box function or the Excel Application object's Input Box Method. The InputBox Method allows for some entry validation using its optional Type argument and is the only one where you can point out of the box to select a range of cells on a worksheet. Invalid data entry into Excel's Input Box is handled by the Excel application.

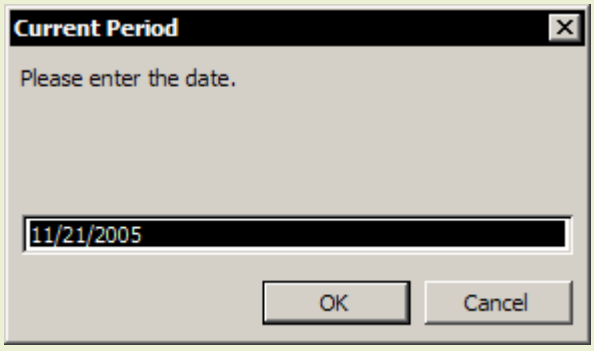
VBA Input Box Function



The generic VBA function does not have any facility for validating the user's input, this has to be done in the code. The result of the function can be directly assigned to a cell but it is usually better assigned to a variable so that it can be effectively evaluated.

```
Range("A1") = InputBox("Please enter the date.", _
                       Title:="Current Period", _
                       Default:=Date)
```

Excel Input Box Method



You will notice the difference between the two when you enter an invalid input. So long as you have completed the Type argument, Excel will handle any invalid input but you must test for the Cancel button in your code. The Cancel button for the Input Box function returns a zero-length string, test for "" in your code. The Cancel button for the Input Box method returns FALSE.

```
Range("A1") = Application.InputBox("Please enter the date.", _
                                   Title:="Current Period", _
                                   Default:=Date, _
                                   Type:=1)
```

0	Formula
1	Number
2	Text
4	True or False
8	Cell reference
16	Error value
64	An array of values

The Type argument specifies the return data type. It can be one or a sum of the values shown in the table.

Only the Excel InputBox Method allows you to point out of the box to return a range reference, in the example return data type 8 is specified and the input box will accept a range reference either by typing or dragging through the cells:

```
Sub ExcelInputBoxMethod()

    Set MyRange = Application.InputBox( _
                              Prompt:="Please select a range.", _
                              Title:="Colour me Red", _
                              Type:=8)

    MyRange.Interior.ColorIndex = 3

End Sub
```

When you assign a variable value using an Input Box never set the data type before the input has been received and validated. To avoid Type Mismatch errors, declare the variable as Type Variant and then use Type conversion functions after the input has been captured and validated.

In the following example the USD variable has to be of Type Currency. Had the initial declaration been As Currency then the code would produce a Type Mismatch error when the Input Box received invalid data and before the input could be evaluated in the loop.

```
Sub MisMatchErrors()
      Dim USD As Variant

      Do
            USD = InputBox("Enter the USD rate:")
      Loop Until IsNumeric(USD) = True

      USD = CCur(USD)

      MsgBox USD

End Sub
```

Most of the work involved in coding Input Boxes is in the validation of the received input. In the following example, we must specify the current month as a two-digit string.

```
Sub DataValidation()
      Dim vMonthNo As Variant

      vMonthNo = InputBox("Enter current month number.", _
                                    Title:="Month Number", _
                                    Default:=Month(Date))

      Select Case vMonthNo
            Case 1 To 12
                  vMonthNo = Round(vMonthNo, 0)
                  vMonthNo = Format(vMonthNo, "00")
                  MsgBox "Current month is " & vMonthNo
            Case Else
                  MsgBox "Action cancelled"
                  Exit Sub
      End Select

End Sub
```

If your Input Box is prompting for an entry into a worksheet cell then consider an easier alternative to writing a macro. See Review of Excel's User Interface features

Excel's Status Bar and Caption

Changing the value of the Application's StatusBar (bottom of the Window) or Caption (top of the Window) properties is ideal for non-modal messaging. The StatusBar is often used for progress messages.

This example forces the status bar to be visible as it sets the status bar text to "Updating data, please wait" while the File Links are updated, then it restores the original state.

```
With Application
      x = .DisplayStatusBar
      .DisplayStatusBar = True
      .StatusBar = "Updating data, please wait..."
      ThisWorkbook.UpdateLink Name:="C:\MyData.xls", _
                                                Type:=xlExcelLinks
      .StatusBar = False
      .DisplayStatusBar = x
End With
```

The Caption property is the text that appears in the title bar of the main Microsoft Excel window. If you do not set a name, or if you set the property to Empty, this property returns "Microsoft Excel".

```
Application.Caption = "The date today is " & Date
```

Menus and Toolbars

Simple Method

Consider an easier alternative to constructing a Menu or Toolbar by using code. It is simpler to open the Excel Customize dialog, go to the Toolbars section and click the New button. Now you have created a new Toolbar you can add Command Bar objects, see Command Bars and then Attach the Toolbar to the workbook. Finally, you use a suitable Event, such as the Worksheet Activate or Workbook Open events, to show the Toolbar as required. See Events

The following examples are event procedures that display the MyBar Toolbar docked at the top of the workspace window when the worksheet is activated and then hide it when a different worksheet is activated.

```
Private Sub Worksheet_Activate()
  With CommandBars("MyBar")
    .Visible = True
    .Position = msoBarTop
  End With
End Sub
```

```
Private Sub Worksheet_Deactivate()
  CommandBars("MyBar").Visible = False
End Sub
```

Using VBA code to construct menus

Menu bars and Toolbars are CommandBar objects. Menus are CommandBar Popup objects and items on the menus are CommandBar Button objects. The code for creating menu structures is rather dense but the process is straightforward, you are adding Controls to Command Bars and assigning macros to them.

Menu construction code is best implemented using Event procedures. The following example uses the Activate and Deactivate events of the ThisWorkbook object. The custom menu is displayed at the top of the worksheet window when the workbook is activated and all the current toolbars are hidden. When another workbook has the focus, the normal menu bar is reinstated and the Standard and Formatting toolbars are displayed.

The custom menu retains the Excel File and Window menus but substitutes a new structure into the body of the menu so that the main Excel menu looks like this:

Here are the event procedures. For the sake of convenience, the procedure for creating the new menu structure is stored in the separate module, 'mdlUserMenus'.

```
Private Sub Workbook_WindowActivate(ByVal Wn As Excel.Window)
    Dim cbMenuBar As CommandBar

    On Error Resume Next

    'Turn off display of visible Command Bars except for
    'the Worksheet Menu Bar.
```

```
    For Each cbMenuBar In Application.CommandBars
            If cbMenuBar.Visible And Not cbMenuBar.Index = 1 Then
                  cbMenuBar.Visible = False
            End If
    Next

    'Create local application menu bar.
    Call mdlUserMenus.SetUpMenu

End Sub
```

```
Private Sub Workbook_WindowDeactivate(ByVal Wn As Excel.Window)

    On Error Resume Next

    With Application
            .CommandBars.Item(1).Reset 'Worksheet Menu Bar.
            .CommandBars.Item(3).Visible = True 'Standard Toolbar.
            .CommandBars.Item(4).Visible = True 'Formatting Toolbar.
    End With

End Sub
```

Here is the procedure that creates the new menu structure. The Caption property sets the text that is displayed in the menu, an ampersand character (&) before a letter underlines it in the menu and sets it as the accelerator key. The OnAction property is where you nominate the procedure that the menu item calls when it is selected. The FaceID property is optional and is used when you require an icon displayed in the menu.

```
Public Sub SetUpMenu()
    Dim cbMenuBar      As CommandBar
    Dim cbElement      As CommandBarControl
    Dim cbChangeDate   As CommandBarControl
    Dim cbRecords      As CommandBarControl
    Dim cbCheckRecords As CommandBarControl
    Dim cbGetRecords   As CommandBarControl
    Dim cbRunReport    As CommandBarControl

    'Clear elements of Worksheet Menu Bar.
    Set cbMenuBar = Application.CommandBars.Item(1)
    With cbMenuBar
            'Force the menu to standard configuration.
            .Reset
            For Each cbElement In .Controls
                  Select Case cbElement.Caption
                        Case "&File", "&Window"'Do nothing.
                        Case Else
                              cbElement.Delete
                  End Select
            Next
            .Position = msoBarTop
    End With

    'Construct new menu items.
    Set cbChangeDate = _
                  cbMenuBar.Controls. _
                  Add(Type:=msoControlButton,Before:=2, Temporary:=True)
    With cbChangeDate
            .Style = msoButtonIconAndCaption
            .Caption = "Change &Date..."
            .FaceId = 125
            .OnAction = "DBOps01ChangeDate"
    End With
```

```
    Set cbRecords = _
      cbMenuBar.Controls. _
      Add(Type:=msoControlPopup, Before:=3, Temporary:=True)
    With cbRecords
            .Caption = "&Records"
    End With

    Set cbGetRecords = _
     cbRecords.Controls.Add(Type:=msoControlButton, Temporary:=True)
    With cbGetRecords
            .Caption = "&Get Records..."
            .FaceId = 2151
            .OnAction = "DBOps02GetRecords"
    End With

    Set cbCheckRecords = _
     cbRecords.Controls.Add(Type:=msoControlButton, Temporary:=True)
    With cbCheckRecords
            .Caption = "&Check Records"
            .FaceId = 141
            .OnAction = "DBOps07DefineData"
    End With

    Set cbRunReport = _
      cbMenuBar.Controls.Add(Type:=msoControlButton, _
      Before:=4, Temporary:=True)
    With cbRunReport
            .Style = msoButtonIconAndCaption
            .Caption = "Run Re&port..."
            .FaceId = 3271
            .OnAction = "DBOps04RunReport"
    End With

End Sub
```

Restoring the user's Toolbars

In the previous example we turned off all the toolbars and then reinstated just the Standard and Formatting toolbars. A better approach would have been to store the identity of the currently visible toolbars and then reinstate the original state of the user's toolbars. In the following example, we loop through the Command Bars Collection and store the Index value of each visible bar in the module level variable, 'm_iVisibleBars'.

The variable contains a dynamic array as clearly, it would not be possible to size the array until the number and identity of the visible toolbars had been retrieved.

```
'Store identity of visible command bars to enable reset.
ReDim m_iVisibleBars(0)
For Each cbMenuBar In Application.CommandBars
    If cbMenuBar.Visible And Not cbMenuBar.Index = 1 Then
            m_iVisibleBars(UBound(m_iVisibleBars)) = cbMenuBar.Index
            ReDim Preserve m_iVisibleBars(UBound(m_iVisibleBars) + 1)
    End If
Next

'Size the array to the data stored.
If Not UBound(m_iVisibleBars) = 0 Then
      ReDim Preserve m_iVisibleBars(UBound(m_iVisibleBars) - 1)
End If
```

At the appropriate time, it is then a simple matter to restore the original state of the user's toolbars by looping through the elements of the array.

See overleaf:

```
'Reinstate original Command Bars.
If Not UBound(m_iVisibleBars) = 0 Then
    For i = LBound(m_iVisibleBars) To UBound(m_iVisibleBars)
            Application.CommandBars(m_iVisibleBars(i)).Visible = True
    Next
End If
```

Calling Excel's built-in Dialogs

You can show any of the Excel application's intrinsic dialogs using the Dialogs property of the application object. The following statement displays the Excel File, Open dialog box.

```
Application.Dialogs(xlDialogOpen).Show
```

The Show Method returns False if the dialog was cancelled. The Show Method does have optional arguments to control some of the options in the dialog, they are documented in VBA Help; look for the Help topic "Built-In Dialog Box Argument Lists". All these arguments are positional, not named so you will have to use commas to denote the specific argument value.

The following procedure points to a specific directory, shows the File, Open dialog listing All Files and evaluates whether or not the dialog was cancelled.

```
Sub ShowExcelFileOpen()
      Dim ReturnValue As Boolean
      ChDir "C:\My Documents"
      ReturnValue = Application.Dialogs(xlDialogOpen).Show("*.*")
      If Not ReturnValue Then
        MsgBox "Cancelled"
      End If
End Sub
```

Review of Excel's User Interface features

Before deciding to use the user interaction resources of VBA make sure that you are not ignoring the application itself. It is, of course, much quicker and easier to use features in the application that you have paid for than it is to recreate them.

Task	Feature	Excel Menu
Changing the appearance of cells based on the data stored in them.	Conditional Formatting. User Defined Number Formats.	Format
Pop-up messages in cells.	Comments.	Insert
Prompting for and governing the input of data into cells. Showing a drop down list of choices in a worksheet cell.	Validation.	Data
Interactive graphical controls such as drop-down lists, check boxes etc. linked to cells.	Form Controls. ActiveX Controls.	Forms Toolbar. Control Toolbox.

User Forms

Display interactive dialogs in the Excel interface by including a User Form in your project. The programming of User Forms can be time-consuming as every action that the User Form performs has to be coded, the OK button does not do anything until you write the code contained in its click event.

You need to be familiar with User Form objects, there is no macro recorder here. The User Form object model is zero-based, the first item in a list is item 0. Excel is one-based. There are potential mismatch problems.

Designing the User Form

The general methodology for designing User Forms is as follows:

1. Insert a User Form into your Project.
2. Create the visual image by adding Controls to the Form.
3. Name the Controls and set their static properties.
4. Write the code in your General Module to show the User Form.
5. Fill in the event code shells in the User Form's object module.

Step by Step

If any of the interface elements mentioned below are not visible then choose them from the View Menu.

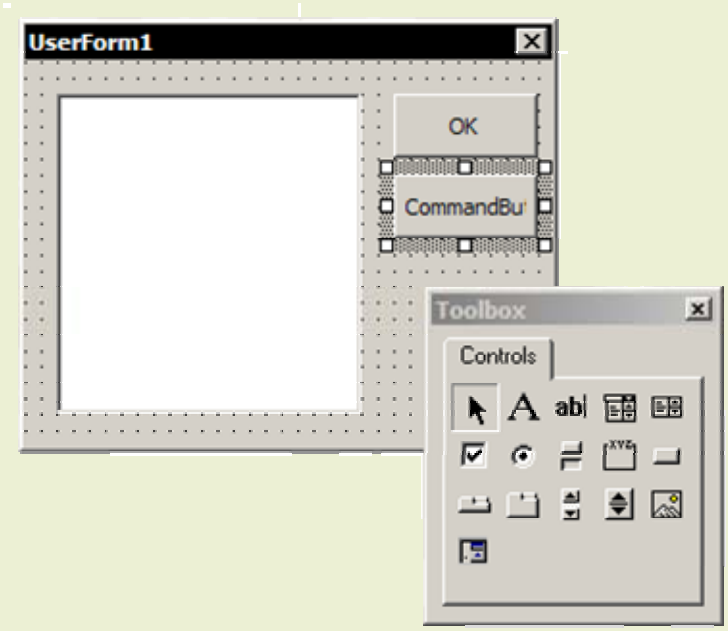
1. Select your project in the Project Explorer Window and use the Insert menu to insert a User Form.
2. Use the Toolbox to draw the required Controls on the User Form. Drag to resize the Form or its contained Controls as necessary. Use the usual Drag and Drop techniques to copy or move the Controls—drag to move, CTRL drag to copy.
3. Set the Name properties of your controls as soon as you have drawn them. It is important that you do this early on as it can prove impossible to register them later on and you are stuck with the default Names. Use the Properties Window to set any other properties that are static, such as Captions.

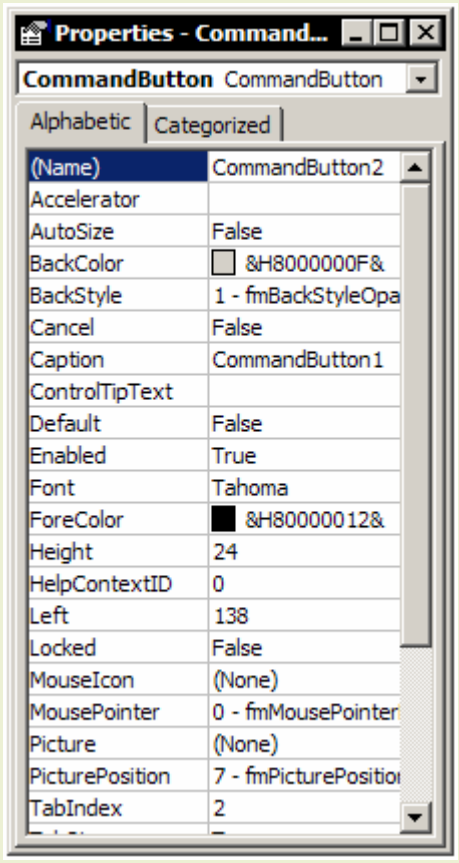
   Some Form properties are static and are done at Design Time, others are dynamic and will be changed as the user manipulates the Form. These are done in code at Run Time.
4. In your General Module (use the Insert menu to insert a Module if necessary) enter the code to show the User Form at the relevant point in your procedure.

   ```
   UserFormName.Show
   ```

   Use the Close box on the Form to close it; you will notice that clicking the OK button at this stage has no effect.
5. Now expose the User Form's Object Module and complete the Events for each Form element. Click the View Code tool or press F7 or double-click any one of the Controls.

You will find that during this design process that you will have numerous windows open in the VB Editor and you may find yourself getting confused and loosing track of what you were trying to do. Persevere, you do get used to it. But there is no magic wand, you have to get used to all the different views and windows. There is the window containing the User Form object which has two views; the code view and the object view, there is the Properties window and there is the Controls Toolbox.





Designing a User Form; a riot of windows and objects.

Draw the controls on the User Form and set their properties in the Properties window. Try to remember to name each control as you create it; discipline at this stage pays dividends when you come to write the code.

Completing the Form's Events

You will see two drop-down lists at the top of the Code Window for the User Form object module. The left-hand list is the *Object list*, the right-hand list is the *Procedure list*. Choose an object from the Object list and its associated Events are displayed in the Procedure list.

Before starting work on the code, consider what you want to do and how the Form should be interacting with its user. For example, use the Form's Initialize event to set default values or build lists before the Form is visible, use the Click event of a command button to close the form etc.

To place the User Form in memory, without displaying it:

```
Load UserFormName
```

To remove the User Form from memory:

```
Unload UserFormName
```

To display the User Form:

```
UserFormName.Show
```

To remove the User Form from the display, but not from memory:

```
UserFormName.Hide
```

Hide the Form when you intend users to switch in and out of the same Form repeatedly. Unload the User Forms as soon as you can, to release the memory. Once the User Form has been unloaded the values of its controls are no longer in scope.

When you access the User Form Object from your General module use the name of the object. In the User Form Object Module code you can use the keyword, Me.

It is important that the User Form is unloaded at the right time so that key decisions and selections made in the User Form are available for evaluation when your code needs to continue.

For example, you might want to return to the main process code in your General module and write the code that would be the outcome of choosing either the OK or Cancel

buttons in the User Form. None of the controls would be visible to the General module at this point if the Form had already been unloaded.

In the first example, the Click Events of the OK and Cancel buttons change the value of a Public variable and then unload the Form. The Public variable is still in scope after the Form has been destroyed and is therefore available for evaluation in the General module.

General Module Code

```
Public GlobalVar As Integer

Sub Main()
     GlobalVar = 1
     frmDemo.Show
     Select Case GlobalVar
           Case 1
                 'OK button
           Case 0
                 'Cancel button
     End Select
End Sub
```

User Form Object Module Code

```
Private Sub cmdOK_Click()
     GlobalVar = 1
     Unload Me
End Sub

Private Sub cmdCancel_Click()
     GlobalVar = 0
     Unload Me
End Sub
```

In the second example the User Form is only hidden, not unloaded by the OK and Cancel button Click Events. The Form remains in scope with its control values visible to the main process code in the General module. The relevant decisions based on its control values are made and then finally the Form is unloaded. Form Controls have a non-specific property, Tag which can be used to store a control value.

General Module Code

```
Sub Main()
     frmDemo.Show
     Select Case frmDemo.cmdOK.Tag
           Case True
                 'OK button
           Case False
                 'Cancel button
     End Select
     Unload frmDemo
End Sub
```

User Form Object Module Code

```
Private Sub cmdOK_Click()
  With Me
     .cmdOK.Tag = True
     .cmdCancel.Tag = False
     .Hide
  End With
End Sub

Private Sub cmdCancel_Click()
  With Me
     .cmdOK.Tag = False
     .cmdCancel.Tag = True
     .Hide
  End With
End Sub
```

But the code is still not completed, as we have not yet handled the situation where the user has closed the User Form by clicking the Form's Close Box instead of using the Cancel button. In this case, the form is unloaded but none of the code associated with the Cancel button is executed; as the Click event has not occurred. Here, we must use the Form's QueryClose event to specify the precise meaning of the Close Box.

Having to consider all the nuances of the User Form's events makes coding User Forms a chore but it is the only way to achieve a robust application.

A User Form in your Project means that you have more than one code module to deal with. It is good practice to follow the convention of organising your code so that the main process of execution is in the General Module and the code in the User Form Module is restricted to the manipulation of the Form.

Naming Conventions

It is awkward having to use the default object names when you are completing the event procedures for each control; is the OK button CommandButton1 or is it CommandButton2? Follow the published standard conventions for Control names, add the three-character lower case prefix to your names and you will never have any problems identifying your control objects in code.

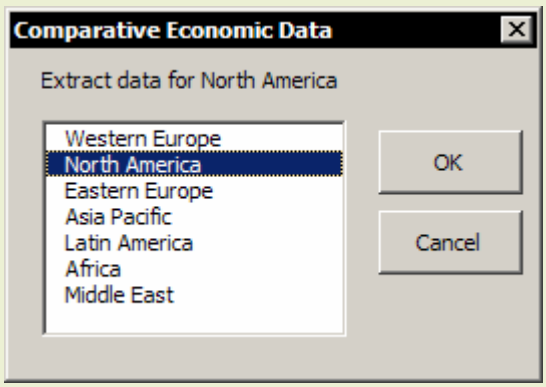
Remember to name your controls as they are created and before you run the Form, you may not be able to rename then retrospectively.

Object	Prefix
Check Box	chk
Combo Box	cbo
Command Button	cmd
Frame	fra
Label	lbl
List Box	lst
Option Button	opt
Text Box	txt
Toggle Button	tog
User Form	frm

Give the controls obvious names; remember that when you are writing the code for the controls you will not be able to see the User Form. For example, good names for the OK and Cancel buttons are 'cmdOK' and 'cmdCancel'. All the control names will be available in the Complete Word listings so the more organised and consistent the naming convention is the easier the code will be to write.

User Form Example Code

In the following example we show the User Form illustrated to the right. The items in the list correspond to Range Names in the workbook that store data. When you click an item in the list, the label in the Form has to change. When you click the OK button, the data is cleared from the target range and is replaced by data copied from the selected range. The same happens when you double click one of the items in the list. Nothing happens if you click the Cancel button or close the User Form.



You are informed if you clicked the OK button but did not choose an item in the list.

Code in the General Module

```
Option Explicit

'Global control variable visible to the User Form.
Public g_strRegionSelected As String
```

```
Public Sub ExtractRegionalData()
    Dim rngSource    As Range
    Dim rngDestination As Range
    Dim rngOldData   As Range

    'Initialise regional choice variable.
    g_strRegionSelected = ""

    'Show user form to determine region choice.
    frmRegion.Show

    Select Case g_strRegionSelected
            Case " " 'Action cancelled.
                  GoTo ExtractRegionalDataCLOSE
            Case "" 'No selection made.

                  MsgBox "You did not choose a Region.", _
                                    Buttons:=vbExclamation, _
                                    Title:="Data not extracted"
                  GoTo ExtractRegionalDataCLOSE

            Case Else 'Extract selected regional data.
```

```
                'Transform Region text into Name definition.
                g_strRegionSelected = Application. _
                            WorksheetFunction.Substitute _
                                  (g_strRegionSelected," ", "_")

                'Clear destination range.
                Set rngDestination = Range("Destination")
                With rngDestination
                      If .CurrentRegion.Rows.Count > 2 Then
                            Set rngOldData = _
                                        Range(rngDestination,                _
                                        Cells(.End(xlDown).Row, 7))
                            rngOldData.Clear
                      End If
                End With

                'Initialise source range.
                Set rngSource = Range(g_strRegionSelected)

                'Copy source to destination.
                rngSource.Copy rngDestination

    End Select

    ExtractRegionalDataCLOSE:
    Exit Sub

End Sub
```

Code in the User Form Object Module

```
Private Sub UserForm_Initialize()
      With Me
            'Initialise instructions text.
            .lblRegion.Caption = "Choose a Region:"
            'Initialise list items.
            With .lstRegion
                  .AddItem "Western Europe"
                  .AddItem "North America"
                  .AddItem "Eastern Europe"
                  .AddItem "Asia Pacific"
                  .AddItem "Latin America"
                  .AddItem "Africa"
                  .AddItem "Middle East"
            End With
      End With
End Sub
```

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

```
Private Sub cmdCancel_Click()
    g_strRegionSelected = " "
    Unload Me
End Sub
```

```
Private Sub lstRegion_Click()
      With Me
            'Which Region is selected.
            g_strRegionSelected = _
                  .lstRegion.List(.lstRegion.ListIndex)
            'Change list caption.
```

```
            .lblRegion.Caption = _
                  "Extract data for " & g_strRegionSelected
      End With
End Sub
```

```
  Private Sub lstRegion_DblClick(ByVal Cancel As _ MSForms.ReturnBoolean)
      With Me.lstRegion
            'Which Region is selected.
            g_strRegionSelected = _
                        .List(.ListIndex)
      End With
      Unload Me
End Sub
```

```
Private Sub UserForm_QueryClose(Cancel As Integer, _
             CloseMode As Integer)

         'The Form's Close Box mimics the Cancel Button.
         If CloseMode <> vbFormCode Then
               g_strRegionSelected = " "
         End If

End Sub
```

List Boxes

In the previous example the list in the List Box was populated from static values in the code using the AddItem method. This is not always appropriate and you may need to fill the list with values from worksheet cells. Use the RowSource property of the List Box to specify the cell values required but do not try to use an object reference; only an external formula reference is accepted.

If you are setting the property value in the Properties Window then the following style of reference should be used:

=Sheet1!A1:A12

If you are setting the property value in your code then the statement should be like this:

```
Me.NameOfListBox.RowSource = "=Sheet1!A1:A12"
```

The following example we are unable to specify the cell range for a list box definitively as the list is dynamic and constantly changing. Use the CurrentRegion property to find the list and then the Address property to reveal the cell references of the list.

```
Private Sub UserForm_Initialize()
    Dim SheetName       As String
    Dim SourceRange     As String

    SheetName = ActiveSheet.Name
    SourceRange = Range("A1").CurrentRegion.Address

    Me.lstDynamic.RowSource = _
                  "=" & SheetName & "!" & SourceRange

End Sub
```

Instancing a User Form

The size of your workbook file will increase dramatically if you include multiple User Forms. In this case, consider having just one base User Form and changing the Form and its controls using the Form's Initialize event. You show the same Form in various different guises by creating an *instance* (a copy) of the Form object using the New keyword.

Design the Form by drawing all the control objects required and then hide or reveal them or change their positions as necessary. The following example shows a succession of two User Forms, both are completely different but are the same base Form object, frmDemo.

```
Public g_sTypeOfForm As String

Public Sub InstancingUserForm()
    Dim MyForm As frmDemo

    'Create an instance of the base Form.
    g_sTypeOfForm = "Step 1"
    Set MyForm = New frmDemo
    MyForm.Show

    'Create another instance of the base Form.
    g_sTypeOfForm = "Step 2"
    Set MyForm = New frmDemo
    MyForm.Show

End Sub
```

Code in the User Form Object Module

```
Private Sub UserForm_Initialize()
    With Me
            'Hide all form controls.
            For Each Control In .Controls
                  Control.Visible = False
            Next
            'Initialise User Form controls.
            Select Case g_sTypeOfForm
                  Case "Step 1"
                        .Caption = "Step 1 of 2"
                        .Height = 180
                        .Width = 240
                        With .cmdButton1
                              .Caption = "Next"
                              .Left = 156
                              .Top = 24
                              .Visible = True
                        End With
                        With .cmdButton2
                              .Caption = "Cancel"
                              .Left = 156
                              .Top = 72
                              .Visible = True
                        End With
                        'Etc. Specify the controls for the Form.
                  Case "Step 2"
                  'Etc. Etc. Specify the controls for the other Form.
              End Select
    End With
End Sub
```

The code is quite long and repetitive but is easily generated by copying. Execution of the code is rapid; it is certainly no slower to build Form controls through code than it is to have them preset. The memory overhead of extra lines of code in a module is significantly less than that of multiple User Forms.

Using Me

You will have noticed from the examples the use of the keyword, Me to return the reference to the User Form object itself. This should only be used in the code contained

in the User Form module, it is out of scope in the General module. It can be omitted as the top level object in the User Form is, of course, the User Form itself.

For example, to return the reference to the User Form, frmDataEntry. In the General module, the reference would have to be explicit:

```
frmDataEntry.Show
```

However, in the Object module, the reference would either be explicit:

```
frmDataEntry.Caption = "Step 1 of 2"
```

Or use Me:

```
Me.Caption = "Step 1 of 2"
```

Or be entirely implicit:

```
Caption = "Step 1 of 2"
```

VBA Memory Arrays

Using Arrays to store sets of data

Variables that store more than one element of data are described as arrays. Arrays are usually lists or tables of related data. See also User Defined Data Type

Arrays have *Dimensions* that contain *Elements*. They are tables of data held in memory. Information stored in arrays is faster and easier to manipulate than information stored in worksheet cells.

Arrays can store any type of data and arrays can contain other arrays. Arrays are either of fixed dimension, see Dimensioned Arrays, or can be sized and resized at run time, see Dynamic Arrays. Arrays are a convenient and efficient alternative to storing data in worksheet cells. Arrays can easily be created and populated from data stored in a range of worksheet cells, see Using cell values in arrays

By default, VBA arrays are zero-based (the first item is 0). Excel is one-based (the first item is 1). This can cause problems but they are not serious so long as you are aware that potential mismatches can occur.

You can re-base the entire module (using Option Base 1) but be careful, different versions of Excel behave to base changes in different ways. If the base value is a problem, then it is usually best to one-base the arrays that you create.

Use the functions LBound and UBound to return the lower and upper boundaries of an array in preference to using constants.

Dimensioned Arrays

The Variant Array

The simplest array form is a variant array using the array function. The data type must always be of type Variant irrespective of the data stored in the array. In some versions of Excel variant arrays are always zero-based and do not comply with the module base.

```
Sub VariantArray()

      Dim vRanges As Variant
      Dim vRange As Variant

      'Store a list of range names.
      vRanges = Array ("Jan", "Feb", "Mar",
                       "Apl", "May", "Jun")

      'Loop to print the ranges.
      For i = LBound(vRanges) To UBound(vRanges)
            Range(vRanges(i)).PrintOut
      Next
```

```
      'or

      For Each vRange In vRanges
            Range(vRange).PrintOut
      Next

End Sub
```

Array Subscripts

Arrays are created when a variable is declared with a dimensional subscript value and can be single dimensioned or multi-dimensional. Arrays can have up to 60 dimensions. The data type is common to the entire array, although type Variant is acceptable. Arrays only need to be declared to the dimensions of the data that they will hold, beware of eating up memory by over-sizing your arrays.

This statement declares an array of ten elements:

```
Dim MyList(1 To 10)
```

This statement declares an array of one hundred elements, not twenty:

```
Dim MyList(1 To 10, 1 To 10)
```

Declaring and populating arrays:

```
Sub DimensionedArray()

      Dim sList(4)                  As String
      Dim sTable(1 To 5, 1 To 2)    As String

      'A zero based one-dimensional array of strings.
      sList(0) = "Jan"
      sList(1) = "Feb"
      sList(2) = "Mar"
      sList(3) = "Apr"
      sList(4) = "May"

      'A one based two-dimensional array of strings.
      sTable(1, 1) = "Jan"
      sTable(2, 1) = "Feb"
      sTable(3, 1) = "Mar"
      sTable(4, 1) = "Apr"
      sTable(5, 1) = "May"
      sTable(1, 2) = "January"
      sTable(2, 2) = "February"
      sTable(3, 2) = "March"
      sTable(4, 2) = "April"
      sTable(5, 2) = "May"

      'Return the 4th element of the 2nd dimension.
      MsgBox sTable(4, 2)

End Sub
```

Using Cell values in arrays

Arrays are easily created from cell values by direct assignment to a variable and are always one-based. The array is two-dimensional if the range is two-dimensional. The values from the cells are read into memory where they can be easily manipulated and written back when required. Of course, the array subscripts correspond precisely to the R1C1 coordinates of the range.

The following example creates and populates a one based, two-dimensional array from the range of cells; view the array elements in the Locals window.

```
Dim MyArray As Variant

MyArray = Sheets(1).Range("A1:B6")
```

Dynamic Arrays

A dimensioned array has to be declared using a constant value, however this constant value maybe unknown at the point of declaration. Use ReDim instead of Dim to create a dynamic array; one that can be re-sized at run time.

```
Sub DynamicArray()

      'Create an array of sheet names.
      Dim iNumShts      As Integer
      Dim i             As Integer

      'Calculate the number of sheets.
      iNumShts = Sheets.Count

      'Size the array.
      ReDim sSheetNames(1 To iNumShts) As String

      'Populate the array.
      For i = LBound(sSheetNames) To UBound(sSheetNames)
            sSheetNames(i) = Sheets(i).Name
      Next

      'Add another sheet.
      Sheets.Add

      'Resize the array.
      ReDim sSheetNames(1 To Sheets.Count) As String

      'Repopulate the array.
      For i = LBound(sSheetNames) To UBound(sSheetNames)
            sSheetNames(i) = Sheets(i).Name
      Next

End Sub
```

In the previous example you will have noticed that we had to repopulate the array after having resized it. ReDim resizes the array but clears the data already stored. Use ReDim Preserve when you want to resize an array but retain the data previously stored.

ReDim Preserve is particularly useful when you want to gather some information and store it in an array but do not know the extent of the data. In the following example a range of cells is being searched, we want to store the cell references of the cells containing a certain value.

As the data is found, it is stored in the array and then an extra element is added to the array ready for the next item of data. When the search is completed the array has one element too many; this is then removed.

Note the use of ReDim at the start of the procedure to initialise the array variable, this has to be done so that the UBound function can calculate the size of the array when the first element of data is stored.

```
Sub DynamicArrayOnTheFly()

      Dim vList As Variant
      Dim oCell As Range

      'Initialise the variable so that we can
      'use UBound later on.
      ReDim vList(0)
```

```
    'Loop through the cells.
    For Each oCell In Range("A1:D50")

        'Test for a value of 5.
        If oCell.Value = 5 Then

            'Store cell reference in array.
            vList(UBound(vList)) = oCell.Address

            'Add element to array ready for next item.
            ReDim Preserve vList(UBound(vList) + 1)

        End If

    Next

    'Remove empty element from array.
    ReDim Preserve vList(UBound(vList) - 1)

End Sub
```

VBA Error Handling

It is not always possible to test and debug a procedure to the extent that every possible error is allowed for. Some errors are impossible to test for; they have to be allowed to occur so that they can then be handled.

Use the On Error Statement to allow and plan for errors, building in commands that enable the procedure to continue in run time. Without an On Error statement, any run-time error that occurs is fatal and the procedure is terminated.

You will probably need to redirect the flow of control using the GoTo statement, this sends execution to a specific point, a line label, in the procedure. A line label is a text identifier and a colon. In the following example, notice how the procedure flows directly to the line label and ignores the intervening code.

```
Sub GoToLineLabels()

    GoTo MyLineLabel

    MsgBox "Hello Charlie"

    MyLineLabel:

    MsgBox "Goodbye Charlie"

End Sub
```

```
On Error GoTo linelabel
```

This statement redirects flow to a line label in the event of an error occurring:

```
On Error GoTo Error_Handler
    …

Error_Handler:
  Select Case Err.Number
        Case 55     '"File already open" error.
            Close #1
        Case Else
            GoTo Procedure_Exit
  End Select
```

This statement moves to the next statement in the procedure and ignores the error:

```
On Error Resume Next
```

This statement disables the current error handler in the procedure. If the procedure is a subroutine then the error is handled by the calling procedure:

```
On Error GoTo 0
```

You can set as many error statements as you require but only one is current.

```
Sub IgnoringAllErrors()

      'Code will break on all errors.

      On Error Resume Next

      'All errors are ignored.

      On Error GoTo 0

      'Code will break on all errors.

End Sub
```

To return to the statement at which the error occurred:

```
Resume
```

To return to the command after the one that caused the error:

```
Resume Next
```

To resume execution at a specific line label:

```
Resume LineLabel
```

7	Out of memory
11	Division by zero
18	User interrupt occurred
53	File not found
482	Printer error
521	Can't open Clipboard
735	Can't save file to TEMP directory
744	Search text not found
31036	Error saving to file

There is a range of trappable errors with defined values that you can use to evaluate the error. Here are a few examples, for the full listing see "Trappable Errors" in VBA Help.

Use the values of the trappable errors to test for and allow for their occurrence.

```
If Err.Number = 53 Then MsgBox "Bad File Name"
```

The Err Object can be used to give you specific details on the current error, using the following properties:

```
Err.Number
Err.Source
Err.Description
```

You will find that the Err object's Number property will reset under certain conditions, assign its current value to a variable in order to produce reliable validation code.

Here is a standard template for arranging error-handling code. Notice how the error handler is isolated from the main process by terminating the procedure prematurely using the Exit Sub statement. You only want the error handler code to execute if an error actually occurs.

```
Sub ErrorHandlerTemplate()
      Dim x As Integer

      On Error GoTo ErrorHandler

      'Cause an error.
      x = 50000

      'Isolate the error handler from main process.
      Exit Sub

ErrorHandler:
      MsgBox "An unexpected error occurred, " & Err.Description

      Resume Next

End Sub
```

It is quite in order to have the error handler call another procedure passing the current error values for evaluation. Many different procedures can then all use the same central error handler procedure.

```
ErrorHandler:
  Call CentralErrorHandler(Err.Description, Err.Number, Err.Source)
```

Excel Pivot Tables

Excel Pivot Tables are members of the PivotTables Collection which is contained by the worksheet object. Each Pivot Table contains a collection of PivotFields which are identified by the text in the header row of the source data. The Excel VBA documentation advises you to use the macro recorder for the manipulation of Pivot Tables as the object model is quite complicated and there are so many different elements to each table. It is very good advice. The following example is a simplification of a recorded macro where the pivot fields were rearranged. These macro recordings are fairly easy to interpret.

```
Sub ChangeSummaryReport()
   With Sheets("Analysis").PivotTables("TradeSummary")
      With .PivotFields("Product")
         .Orientation = xlColumnField
      End With
      With .PivotFields("Country")
         .Orientation = xlPageField
      End With
   End With
End Sub
```

Creating a Pivot Table report

It is in the creation of a PivotTable where the macro recordings can be difficult to interpret and control. This is a recording of creating a PivotTable:

```
ActiveWorkbook.PivotCaches.Add(SourceType:=xlDatabase, SourceData:= _
        "Sheet1!R1C1:R87C6").CreatePivotTable TableDestination:="", _
        TableName:="PivotTable1", DefaultVersion:=xlPivotTableVersion10
ActiveSheet.PivotTableWizard TableDestination:=ActiveSheet.Cells(3, 1)
ActiveSheet.Cells(3, 1).Select
ActiveSheet.PivotTables("PivotTable1").AddFields RowFields:="Country", _
        ColumnFields:="Month"
ActiveSheet.PivotTables("PivotTable1").PivotFields("Units").Orientation = _
        xlDataField
```

Ouch! We need to make some sense out of this if we are to control the creation of our reports. The source data contains columns containing Product, Country and Month information with Sales Units data that we want to analyse.

To create a new PivotTable we can use the Add and CreatePivotTable methods of the PivotCaches object:

```
ActiveWorkbook.PivotCaches.Add(SourceType:=xlDatabase, SourceData:= _
        "Sheet1!R1C1:R87C6").CreatePivotTable TableDestination:="", _
        TableName:="PivotTable1", DefaultVersion:=xlPivotTableVersion10
```

The *SourceData* is a range object containing the data for the report, the *TableDestination* is where the report is returned. The *TableName* and other arguments are optional.

For example, define the source data as being all the data from A1 on the active worksheet:

```
Set rngSource = ActiveSheet.Range("A1").CurrentRegion
```

The table destination is a new worksheet in the workbook, inserted after the active sheet:

```
Set wksSales = Worksheets.Add(After:=ActiveSheet)
```

And create the PivotTable, naming it as 'Sales Report':

```
ActiveWorkbook.PivotCaches.Add _
        (SourceType:=xlDatabase,SourceData:=rngSource) _
        .CreatePivotTable TableDestination:= wksSales.Range("A1"), _
        TableName:="Sales Report"
```

Create an object variable to refer to the pivot table report:

```
Set ptSales = wksSales.PivotTables("Sales Report")
```

Now, add the fields required. Every column in the source data range creates a member of the PivotFields collection as the Pivot cache contains all the source data. But to show a field in the report you have to use the AddFields method:

```
ptSales.AddFields RowFields:="Country", ColumnFields:="Month"
```

Specifing them as:

```
PageFields:= "Product"
RowFields:= "Country"
ColumnFields:= "Month"
```

To specify two or more fields with the same orientation it is like this:

```
ColumnFields:= Array("Month","Country")
```

Data Fields

To add a data field to the report you do not use the AddFields method, rather you set the Orientation property of an existing pivot table field to xlDataField (this field does not have to be one of those already added, it can be any of the pivot fields):

```
ptSales.PivotFields("Units").Orientation = xlDataField
```

However, it is not possible to predict the name of the new pivot field as Excel names it automatically depending on the default Summary function. If the default Summary function is Sum then it is called "Sum of Units", if the default function is Count then it is called "Count of Units". And, at this stage there is no way of finding out what the default Summary function is! Once you have named the field Excel will not change it again automatically but you need to make sure that you can specify the summary function correctly.

Either, refer to the field not as a member of the PivotFields collection (where it is contained but you do not know what it is called) but as a member of the DataFields collection. As you create a data field it becomes the first member of this collection, the next data field is the second member etc.

```
ptSales.PivotFields("Units").Orientation = xlDataField
ptSales.DataFields(1).Function = xlSum
ptSales.DataFields(1).Name = "Total Sales"
```

Or, set all the relevant properties as you create each data field, like this:

```
With ptSales.PivotFields("Units")
    .Orientation = xlDataField
    .Caption = "Total Sales"
    .Function = xlSum
    .NumberFormat = "#,##0_-"
End With

With ptSales.PivotFields("Units")
    .Orientation = xlDataField
    .Caption = "Units %"
    .Calculation = xlPercentOfTotal
End With
```

Excel Charts

Excel Charts are one of the most complicated sections in the Object Model. The hierarchy of an individual Chart object is fairly obvious, the principal issue is to access the Chart object itself. You can use the ActiveChart property for the current chart but identifying a specific chart can be a problem.

Chart Objects

Excel has two types of chart, a chart on a chart sheet or an embedded chart in a worksheet. There is no ChartSheet object, the Charts property of the Application object returns a Sheets collection containing one Chart object for each chart sheet. It does not contain the Chart objects for the embedded charts.

In the case of the embedded charts, the Chart object is not contained directly in the worksheet. Rather, the worksheet contains a ChartObject object that is a container for the Chart object. Confused? In practice it means that you have to include .Chart in the object reference for the Chart elements, like the axes but not for the Chart area.

Thus, the object reference for the chart sheet, "Chart1" is as follows:

```
ThisWorkbook.Charts("Chart1")
```

Whereas, the reference for "Chart 1" on "Sheet1" is:

```
Worksheets("Sheet1").ChartObjects("Chart 1").Chart
```

It is advisable to examine your recordings carefully and experiment using the Immediate Window before starting your code. Embedded charts in particular.

An object reference like this for the first chart on the worksheet will fail:

```
ChartObjects(1).Name
```

You must return the Sheet object and the Chart object:

```
ActiveSheet.ChartObjects(1).Chart.Name
```

The following procedure creates an embedded chart.

```
Sub CreateEmbeddedChart()
      Dim MyChart As ChartObject
      Dim c       As Long
      Dim r       As Long

      'Get worksheet data for positioning chart.
      c = Columns(1).Width
      r = Rows(1).Height

      'Position chart using worksheet units.
      Set MyChart = ActiveSheet.ChartObjects.Add( _
                        Left:= c * 3, Top:= r * 0.5, _
                        Width:= c * 8, Height:= r * 20)
```

```
      With MyChart
            'Define the Chart type.
            .Chart.ChartType = xlLine

            'Add a data series.
            .Chart.SeriesCollection.Add _
                  Source:=ActiveSheet.Range("A1:B6"), _
                  Rowcol:=xlColumns, _
                  Serieslabels:=True, _
                  Categorylabels:=True

            'Plot area fill colour to blue.
            .Chart.PlotArea.Interior.ColorIndex = 5

            'Add a Chart title.
            .Chart.ChartTitle.Caption = "Plot for " & Date

      End With
End Sub
```

Arranging Charts on a Worksheet

In the following example all the ChartObjects on a worksheet are sized to uniform dimensions and then lined up to worksheet row and column locations. The resulting arrangement is sets of four charts across the worksheet, aligning to columns A,E,I and M, starting a new set of four every 16 rows.

```
Sub LineUpCharts()
    Dim oWSht      As Worksheet
    Dim rSize      As Long
    Dim cSize      As Long
    Dim rAlign     As Long
    Dim cAlign     As Long
    Dim i          As Integer

    Set oWSht = Worksheets("Sheet1")

    'Get worksheet dimension data.
    rSize = oWSht.Rows(1).Height
    cSize = oWSht.Columns(1).Width

    'Initialise row and column alignment variables.
    rAlign = 2
    cAlign = 1

    'Loop through the charts.
    For i = 1 To oWSht.ChartObjects.Count

          With oWSht.ChartObjects(i)

                'Size the chart.
                .Height = rSize * 16
                .Width = cSize * 4

                'Align chart to worksheet rows and columns.
                .Top = oWSht.Rows(rAlign).Top
                .Left = oWSht.Columns(cAlign).Left

          End With

          'Increment column alignment values.
          cAlign = cAlign + 4
```

```
          'Start a new set of four charts.
          If i Mod 4 = 0 Then
            rAlign = rAlign + 16
            cAlign = 1
          End If

   Next

End Sub
```

Embedding Chart Data Series

The following example converts all chart source data from cell references to arrays of constants to make the charts portable and independent of their data (i.e. to mimic pressing F9 in a SERIES formula)

The procedure assumes all charts are embedded charts on worksheets. Note how the loop goes through the worksheets, through each chart on each worksheet and finally through each data series in each chart. The code listing for the function procedure DerivedValues follows the listing for the Sub.

```
Public Sub ChartConstants()

      Dim sPrompt            As String
      Dim iAns               As Integer
      Dim oWSht              As Worksheet
      Dim oChrt              As ChartObject
      Dim oSeries            As Series
      Dim sOldFormulaString  As String
      Dim sFormulaString     As String
      Dim sNewFormulaString  As String
      Dim sArg1              As String
      Dim sArg2              As String
      Dim sArg3              As String
      Dim sArg4              As String
      Dim iComma1            As Integer
      Dim iComma2            As Integer
      Dim iComma3            As Integer
      Dim iBracket1          As Integer
      Dim iBracket2          As Integer

   On Error GoTo ErrChartConstants

   'User prompt.
   sPrompt = "This macro breaks the link between your charts" & _
             " and the data on which they depend." & vbCr & _
             "Do you want to continue?"

   iAns = MsgBox(sPrompt, vbYesNo + vbQuestion, "Unlink Charts")

     'Action cancelled.
   If iAns = vbNo Then GoTo ExitChartConstants

     'Loop for worksheets.
     For Each oWSht In Worksheets
           'Loop for chart objects.
           For Each oChrt In oWSht.ChartObjects
                 'Loop for chart data series.
                 For Each oSeries In oChrt.Chart.SeriesCollection

                       'Manipulate formula string.
                       sOldFormulaString = CStr(oSeries.Formula)
```

```
                        With Application.WorksheetFunction

                        'Reduce the value of the first argument.
                        iBracket1 = .Find("(", sOldFormulaString)
                        iComma1 = .Find(",", sOldFormulaString)
                        sFormulaString = Mid(sOldFormulaString, _
                              iBracket1 + 1, iComma1 - iBracket1 - 1)
                        sArg1 = DerivedValues(sFormulaString)

                        'Reduce the value of the second argument.
                        iComma2 = .Find(",", sOldFormulaString, _
                              iComma1 + 1)
                        sFormulaString = Mid(sOldFormulaString, _
                              iComma1 + 1, iComma2 - iComma1 - 1)
                        sArg2 = DerivedValues(sFormulaString)

                        'Reduce the value of the third argument.
                        iComma3 = .Find(",", sOldFormulaString, _
                              iComma2 + 1)
                        sFormulaString = Mid(sOldFormulaString, _
                              iComma2 + 1, iComma3 - iComma2 - 1)
                        sArg3 = DerivedValues(sFormulaString)

                        'Reduce the value of the forth argument.
                        iBracket2 = .Find(")", sOldFormulaString)
                        sFormulaString = Mid(sOldFormulaString, _
                              iComma3 + 1, iBracket2 - iComma3 - 1)
                        sArg4 = CStr(sFormulaString)

                        End With

                        'Construct formula string from derived 'values.
                        sNewFormulaString = "=SERIES(" & sArg1 & "," _
                                          & sArg2 & "," _
                                          & sArg3 & "," _
                                          & sArg4 & ")"

                        'Substitute new formula string for old.
                        oSeries.Formula = sNewFormulaString

                    Next oSeries
                Next oChrt
        Next oWSht

        'Confirm completion.
        MsgBox "Chart formulas are converted.", vbInformation

        ExitChartConstants:
        Exit Sub

        ErrChartConstants:
          sPrompt = "The following unexpected error occurred: " _
                    & vbCrLf & _
                    Err.Description & _
                    "." & " Error Number: " _
                    & Err.Number & vbCrLf & _
                    "Chart not converted." & vbCrLf & _
                    "Click OK to continue."

        MsgBox sPrompt, vbCritical, "Non Fatal Error"

        Resume Next

End Sub
```

```
Private Function DerivedValues(sFormulaString As String) As String
    'Accepts : Sheet and Cell references in formula language.
      'Returns : Values of those references as valid string 'expressions in
      formula language.

    Dim iExternal       As Integer
    Dim sSheetRef       As String
    Dim sRangeRef       As String
    Dim vCellValues     As Variant
    Dim vElement        As Variant
    Dim vFormulaArray   As Variant

    On Error GoTo 0
      'Force a zero-length string to Empty.
      If sFormulaString = "" Then
            DerivedValues = Empty
            Exit Function
      End If

      'Identify objects and return cell values.
      iExternal = Application.WorksheetFunction.Find("!", _
                                    sFormulaString)
      sSheetRef = Left(sFormulaString, iExternal - 1)
      sRangeRef = Mid(sFormulaString, iExternal + 1)
      vCellValues = Sheets(sSheetRef).Range(sRangeRef).Value

      'Test for an array.
      If Not IsArray(vCellValues) Then
            'Return the single value.
            DerivedValues = """" & CStr(vCellValues) & """"
      Else
            'Construct a string from the array elements.
            For Each vElement In vCellValues

                  'Force empty values to zero.
                  If IsEmpty(vElement) Then vElement = 0

                  'Force strings to literal strings.
                  If Not IsNumeric(vElement) Then
                        vElement = """" & vElement & """"
                  End If

                  vFormulaArray = vFormulaArray & vElement & ","
            Next

            'Remove trailing comma.
            vFormulaArray = Left(vFormulaArray, _
                              Len(vFormulaArray) - 1)

            'Enclose the expression in braces.
            DerivedValues = CStr("{" & vFormulaArray & "}")

      End If

End Function
```

Application Interaction

All MS Office applications are automation clients and servers so that you can use VBA as a bridge language to interact with the services provided by other applications.

Creating Object Model References

Before you can use another object model you must create a reference to the Class containing the Type Library that you wish to use. Declare an object variable to hold the reference to the object and then assign a reference to the object to the variable.

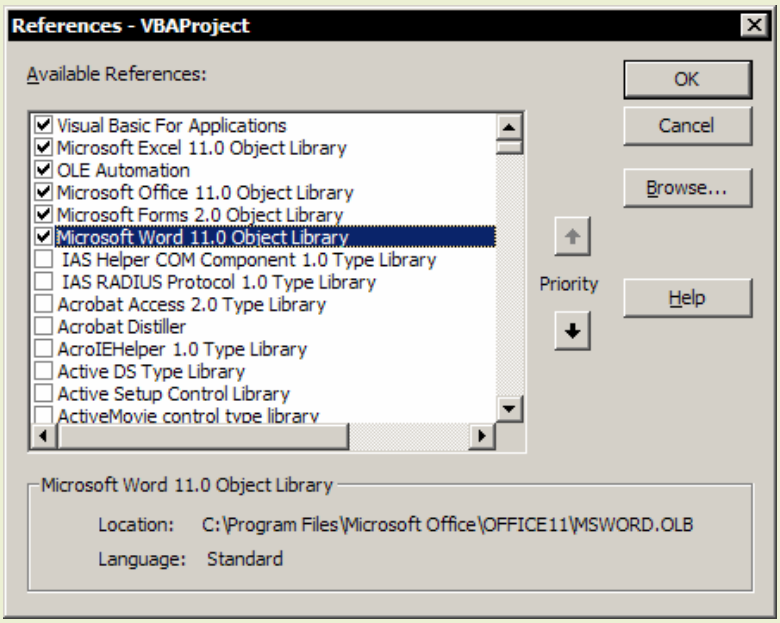
There are two methods, Early Binding and Late Binding. Early Binding is the preference as it is more efficient and allows better use the resources of the VB editor to develop and test your code.

Late Binding

Use the CreateObject or GetObject functions to return an object reference. This gives you a late bound interface meaning that as you write your code in Excel you will not be able to look up Help for the other object model or use statement completion. Here is a late bound instance of MS Word:

```
Sub UsingWordLateBinding()

      'Declare a generic variable to hold the reference.
      Dim wdApp As Object

      Set wdApp = CreateObject("Word.Application")

      'To see the application's interface.
      wdApp.Visible = True

      'Manipulate Word objects.
      wdApp.Documents.Add

End Sub
```

Early Binding



Add a reference to your project using the References dialog. In the VB editor menu choose *Tools*, *References*.

Check the relevant box and move it up the priority list nearer to the top.

Next, you declare an object variable of the specific type and then you use the New keyword to create an instance of the application:

```
Dim wdApp As Word.Application
Set wdApp = New Word.Application
```

Finally, write the code required to manipulate Word. You will see all of the relevant documentation in the Object Browser and the Word Object Library references are available in the Complete Word drop down lists. Again, you have to make the other application visible if you want to see it on your screen. Your code is far more efficient if you do not display the visual interface. However, it is a good idea to have the other application visible while testing your code.

Interacting with MS Word

```
Sub WordAutomationEarlyBinding()

      Dim wdWord        As Word.Application
      Dim wdWordDoc     As Word.Document
      Dim wdWordSel     As Word.Selection
      Dim PrintTime     As Integer
      Dim StartTime     As Single

      On Error GoTo ErrorHandler

      Set wdWord = New Word.Application
      Set wdWordDoc = wdWord.Documents.Add
      Set wdWordSel = wdWord.Selection

      With wdWordSel
            .TypeText "Have a nice day"
            .WholeStory
            .Font.Name = "Arial"
            .Font.Size = 12
            .Font.Bold = wdToggle
      End With

      wdWordDoc.PrintOut

      'Timer to allow for print spooling.
      PrintTime = 20
      StartTime = Timer
      Do While Timer < StartTime + PrintTime
            'Yield to system.
            DoEvents
      Loop

      wdWord.Quit

      'Destroy objects.
      Set wdWordSel = Nothing
      Set wdWordDoc = Nothing
      Set wdWord = Nothing

      Exit Sub

      ErrorHandler:
      MsgBox "Unexpected error. " & Err.Number

End Sub
```

This is the type of code required to automate Excel from another application:

```
Sub ExcelAutomationEarlyBinding()

      Dim oXLApp        As Excel.Application
      Dim oXLWBook      As Excel.Workbook
      Dim oXLWSht       As Excel.Worksheet

      Set oXLApp = New Excel.Application
      Set oXLWBook = oXLApp.Workbooks.Add
      Set oXLWSht = oXLWBook.Worksheets.Add

      With oXLWSht
            .Cells(1, 1) = "Tom"
            .Cells(1, 2) = "Dick"
            .Cells(1, 3) = "Arry"
      End With

      oXLApp.Visible = True

      oXLApp.Quit

      Set oXLApp = Nothing
      Set oXLWBook = Nothing
      Set oXLWSht = Nothing

End Sub
```

Interacting with MS Access

When you write code to work with an MS Access database you need to use the DAO object model to manipulate data stored in Tables and the Access object model to display Forms or print Reports etc. The Access application does not contain its own data.

To copy the contents of a DAO Recordset to an Excel worksheet, set a reference to MS DAO using the References dialog and then use the CopyFromRecordSet Method of the Range object. The following example opens the database 'DB1' and copies the first 10 columns and 10 rows from the 'ClosingPrices' table to the current worksheet, starting with cell reference A1.

```
Sub ReturningDAORecordset()

    Dim rs         As Recordset
    Dim ReturnVal  As Integer

    Set rs = _
            DBEngine.OpenDatabase("c:\db1.mdb"). _
            OpenRecordset("ClosingPrices")

    ReturnVal = Range("A1").CopyFromRecordSet(rs, 10, 10)

End Sub
```

You can omit the column and row values to return the entire Recordset. In the example, the value of the ReturnVal variable is not being used for any specific purpose, you would usually use the variable for validation purposes. Copying begins at the current row of the Recordset object. After copying is completed, the EOF property of the Recordset object is set to TRUE.

Send Keys

If the application that you want to use in your code does not have a programmable interface then use a combination of Shell and SendKeys to interact with it.

```
Sub RunCalculator()

    Dim ReturnValue     As Double
    Dim i               As Integer

    'Run Calculator program.
    ReturnValue = Shell ("CALC.EXE", 1)

    'Activate the Calculator.
    AppActivate ReturnValue

    'Set up counting loop.
    For i = 1 To 20
       SendKeys i & "{+}", True
    Next i

    'Copy result to Clipboard.
    SendKeys "^C", True

    'Send ALT+F4 to close Calculator.
    SendKeys "%{F4}", True

    'Return data to Excel.
    ActiveSheet.Paste

End Sub
```

User Defined Data Type

Sets of related data can be stored in user defined data types. Rather than have three separate variables to contain name, address and birth date data a single data type containing all three can be defined.

```
Type MyData
  Name      As String
  Address   As String
  Birthday  As Date
End Type
```

The Type statement is used at the module level to define a user-defined data type containing one or more elements. User-defined data types can contain elements of any data type, an array, or a previously defined user-defined type.

User Defined Data Types are typically used for the storage of data records. The following example shows the use of the data type, Music.

```
Type Music

  Composer  As String
  Title     As String
  Opus      As Integer

End Type
```

```
Sub Report()
      Dim MusicTitle    As Music
      Dim msg           As String

      MusicTitle.Composer = "Hector Berlioz"
      MusicTitle.Title = "Le Carnaval Romain"
```

```
      MusicTitle.Opus = 9

      msg = "You are listening to " & _
                  MusicTitle.Title _
                  & ". Opus " & _
                  MusicTitle.Opus _
                  & " by " & _
                  MusicTitle.Composer

      MsgBox msg

End Sub
```

Enumerations

You will notice from recorded macros that VBA uses a number of internal constants to identify key values. This makes the code much easier to read. Constant identifiers such as vbYes or xlLandscape are easier to implement and interpret than their actual values.

You can declare your own enumeration variables where you would otherwise have to use numeric constants. For example, fill colours have to be specified as index values in your current colour palette. It is difficult to remember the corresponding number for each colour.

Enumeration variables are declared at the module level with an Enum statement. The elements of the Enum type are initialised to constant values using either positive or negative numbers.

```
Enum MyFillColours

      Red = 3
      Green = 43
      Yellow = 6
      Blue = 49

End Enum
```

```
Sub Main()

      'Colour the cells.
      With ActiveCell
        .Offset(0, 0).ColorIndex = Red
        .Offset(1, 0).ColorIndex = Green
        .Offset(2, 0).ColorIndex = Yellow
        .Offset(3, 0).ColorIndex = Blue
      End With

End Sub
```

By Reference, By Value

Variables may be passed from one procedure to another By Reference or By Value using the statements ByRef or ByVal. All arguments are passed to procedures by reference, unless you specify otherwise.

Passing By Value sends a copy of the original variable. Changes to the argument within the procedure are not reflected back to the original variable. Passing By Reference gives direct access to the variable. The statement is made by the calling procedure. Data types must be consistent.

Passing variables to a subroutine. In the following example the variables x and y are passed to the subroutine Sub2 when it is called by Main. x is passed By Reference and y is passed By Value. The subroutine manipulates the two variables locally but when the flow of control returns to Main the value of the y variable is unchanged.

The same rules apply for passing argument values to a function procedure.

```
Sub Main()

      Dim x As Integer
      Dim y As Integer

     x = 50
      y = 100
      Call Sub2(ByRef x, ByVal y)
      MsgBox x & y

End Sub
```

```
Sub Sub2(x As Integer,y As Integer)

      x = x + 10
      y = y * 2

End Sub
```

Declare the relevant Data Type for the received values in the subroutine otherwise they are stored locally as Variants. The Data Type received must match the Data Type passed.

By Name, By Order

Understanding named and optional argument values. When you call a Sub or Function procedure, you can supply arguments by order, in the order they appear in the procedure's definition, or you can supply the arguments by name without regard to position. Arguments are either optional or required.

The methods of Excel's objects are internal procedures and the same rules apply. For example, the Worksheets object has an Add method that has four optional parameters. (You can see these as you type; press the spacebar after Add and the syntax diagram appears, optional parameters are contained in square brackets)

```
Worksheets.Add([Before],[After],[Count],[Type])
```

To add three sheets after the first sheet using the By Name convention:

```
Worksheets.Add After:= Worksheets(1), Count:= 3
```

or

```
Worksheets.Add Count:= 3, After:= Worksheets(1)
```

To add three sheets after the first sheet using the By Order convention:

```
Worksheets.Add ,Worksheets(1), 3
```

To add three sheets after the first sheet using a combination of both conventions:

```
Worksheets.Add ,Worksheets(1), Count:= 3
```

A named argument consists of the argument name followed by a colon and an equals sign (**:=**), then followed by the argument value. Never use just the equals sign.

Named arguments are especially useful when you are calling a procedure that has optional arguments. If you use named arguments, you do not have to include commas to denote missing positional arguments. Using named arguments makes it easier to read your code.

The parenthesis are only required when you are using the function form to return a value to a variable. In the following example, omitting the parenthesis around the "After" argument would produce a syntax error:

```
Dim MyNewSheet As Worksheet

Set MyNewSheet = Worksheets.Add(After:= Worksheets(1))
```

Classes

Classes define objects. Every Excel object is an *instance* (a copy) of a particular Excel Class. A worksheet object is an instance of the Worksheet Class. Classes are object templates containing their collection of methods and properties. In our VBA procedures we use the Excel objects created for us and rarely need to create our own.

However, for complicated and difficult code structures it is sometimes useful to take an object-orientated approach by creating our own code objects, which are supersets of the existing Excel objects. This will promote simplicity and easier maintenance of the code contained in general modules by allowing us to re-use rather than repeat fragments of code that are frequently required.

Creating an Object

To create your own object you need a Class Module to contain the property and method definitions. Then an instance of the Class creates the object.

For example, we want to create a MyWbk object to use in our procedures in a general module. The object will have a Save method that does not actually save the workbook but instead sets the Saved property of the workbook to TRUE. The object will also have a set of read-only properties listed in the table below:

Property Name	Data Returned
PathName	The full file name and path.
BookName	The workbook name with the .xls extension removed.
NonBlanks	Count of the workbook's cells containing formulas or constants.

The file name and path is directly available as an existing Excel property but the other two are rather more specialised requiring the manipulation of existing properties and we want to be able to retrieve the data without repeating the code every time it is required.

Using a Class Module

Insert a Class module into the Project using the Insert menu and then use the Properties window to set the Name property as clsMyWbk. Enter the code into the Class module, using *Insert*, *Procedure* to reduce the amount of hand typing required.

The Save method is a Public function in the Class module and the three Properties are defined by pairs of Public Property procedures and associated Private procedures which calculate the values for these public properties. The role of a Property procedure is to expose a property value to the outside world.

Code in the Class Module

```
Private m_PathName      As String
Private m_BookName      As String
```

```
Public Function Save()
    ThisWorkbook.Saved = True
End Function
```

```
Public Property Get PathName() As String
  Call GetPathName
  PathName = m_PathName
End Property

Private Sub GetPathName()
  m_PathName = ThisWorkbook.FullName
End Sub
```

```
Public Property Get BookName() As String
  Call GetBookName
  BookName = m_BookName
End Property

Private Sub GetBookName()
    m_BookName = ThisWorkbook.Name
    'Remove the file extension if workbook already saved.
    If Not ThisWorkbook.Path = "" Then
            m_BookName = Left(m_BookName, Len(m_BookName) - 4)
    End if
End Sub
```

```
Public Property Get NonBlanks() As Long
  NonBlanks = CountNonBlanks()
End Property
```

```
Private Function CountNonBlanks() As Long
    Dim wSht As Worksheet
    Dim x As Long, y As Long, z As Long
    On Error Resume Next
    'Loop through the worksheets.
    For Each wSht In Worksheets
            'Count the cells containing constants.
            x = wSht.Cells.SpecialCells(xlCellTypeConstants).Count
            'Count the cells containing formulas.
            y = wSht.Cells.SpecialCells(xlCellTypeFormulas).Count
            'Aggregate the x and y values in z.
            z = z + x + y
            x = 0
            y = 0
      Next
    CountNonBlanks = z
End Function
```

Then you return to your general module to create an instance of the class, clsMyWbk by declaring a Public variable of the specific Class Type and using the New keyword.

The object, MyWbk is of Type clsMyWbk (as defined by the clsMyWbk Class) and we can access its associated methods and properties using the usual Object.Method or Object.Property syntax in our code. Object references are available in Complete Word.

Code in the General Module

```
Public MyWbk As New clsMyWbk

Sub Main()
  MsgBox MyWbk.NonBlanks
  MsgBox MyWbk.BookName
  MsgBox MyWbk.PathName

  MyWbk.Save
End Sub
```

The object only exposes its Public properties and procedures and the internal workings of the Class, how these property values were calculated, are hidden. The object is a container for a collection of properties and procedures. This is the theory of *encapsulation* where complex Private procedures are available through a simpler interface of Public methods and properties.

In the following example we need to set and reset various Excel application and document settings in our procedures. Instead of using a series of subroutine calls, we create a Class, 'clsAppSet' to contain all of our settings, create the object, 'AppSet' and then simply apply them by using the Methods of the object.

Code in the Class Module

```
Private m_StatusBar As Boolean

Public Function LockOn()
    Dim wks As Worksheet

    With Application
            .DisplayStatusBar = m_StatusBar
            .StatusBar = False
            .ScreenUpdating = True
            .DisplayAlerts = True
            .Interactive = True
    End With
    With ThisWorkbook
            For Each wks In .Worksheets
                  wks.Protect Password:="TopSecret"
            Next
            .Protect Password:="TopSecret", Structure:=True
    End With
End Function
```

```
Public Function LockOff()
    Dim wks As Worksheet

    With ThisWorkbook
            For Each wks In .Worksheets
                  wks.Unprotect Password:="TopSecret"
            Next
            .Unprotect Password:= "TopSecret"
    End With
    With Application
            Let m_StatusBar = .DisplayStatusBar
            .DisplayStatusBar = True
            .ScreenUpdating = False
            .DisplayAlerts = False
            .EnableCancelKey = xlDisabled
            .Interactive = False
    End With
End Function
```

Code in the General Module

In any module where these procedures are required, declare the variable 'AppSet' as Class 'clsAppSet' to create the object:

```
Dim AppSet As New clsAppSet
```

Apply the Methods wherever required in the procedure:

```
AppSet.LockOff
```

To save memory, destroy the object when it is no longer required:

```
Set AppSet = Nothing
```

When you need the same procedures again for another Project, just insert a copy of the entire Class module.

Lotus 1-2-3 Translation

The use of Classes often seems to be more in the realm of the "programmer" than the casual macro developer but a basic understanding of the process reveals that it is an excellent method of making macros much simpler and easier to produce by allowing you to readily recall expressions that you regularly use and avoid having to go back to macros that you have already done to copy and paste lines of code.

A good example of this are the statements required for cell selection and movement on a worksheet. Many macro writers find that one of their principle tasks is to translate legacy macros that were written in the Lotus 1-2-3 Classic macro language. In these macros, positioning the cell pointer is crucial and much of the code in the macro consists of cell movement and selection.

It is distressing to discover that simple Lotus instructions like {D 2} have to be translated into clumsy constructions such as *ActiveCell.Offset(2,0).Select* and it is quite difficult to determine exactly how common Lotus command sequences such as {ANCHOR}{END}{DOWN}~ should be translated at all.

The Move Object

This section describes how to produce a user-defined 'Move' Object which is a Class that can be copied into any Excel workbook and provides an easy and direct translation for Lotus 1-2-3 moving and selecting commands into their Excel VBA equivalents.

The 'Move' object contains the following methods:

Down	Move down by one or by a defined number of cells
Right	Move right by one or by a defined number of cells
Up	Move up by one or by a defined number of cells
Left	Move left by one or by a defined number of cells
Home	Move to cell A1
EndDown	Move down to the end of the current region
EndRight	Move right to the end of the current region
EndUp	Move up to the end of the current region
EndLeft	Move left to the end of the current region
SelectEndDown	Extend the selection down to the end
SelectEndRight	Extend the selection right to the end
SelectEndUp	Extend the selection up to the end
SelectEndLeft	Extend the selection left to the end
SelectEndDownAndRight	Extend the selection down and to the right
SelectEndUpAndLeft	Extend the selection up and to the left

Code in the Class Module

Firstly, insert a Class module into the current project by choosing *Insert*, *Class Module* and then enter the following procedures into the module:

```
Public Function Down(Optional Number As Integer)
    If Number = 0 Then Number = 1
    ActiveCell.Offset(Number, 0).Select
End Function
```

```
Public Function Up(Optional Number As Integer)
    If Number = 0 Then Number = 1
    ActiveCell.Offset(-Number, 0).Select
End Function
```

```
Public Function Left(Optional Number As Integer)
    If Number = 0 Then Number = 1
    ActiveCell.Offset(0, -Number).Select
End Function
```

```
Public Function Right(Optional Number As Integer)
    If Number = 0 Then Number = 1
    ActiveCell.Offset(0, Number).Select
End Function
```

```
Public Function Home()
    Range("A1").Select
End Function
```

```
Public Function EndDown()
    ActiveCell.End(xlDown).Select
End Function
```

```
Public Function EndUp()
    ActiveCell.End(xlUp).Select
End Function
```

```
Public Function EndRight()
    ActiveCell.End(xlToRight).Select
End Function
```

```
Public Function EndLeft()
    ActiveCell.End(xlToLeft).Select
End Function
```

```
Public Function SelectEndDown()
    Dim x As Long, y As Long
    x = ActiveCell.Row
    y = ActiveCell.End(xlDown).Row + 1
    ActiveCell.Resize(y - x).Select
End Function
```

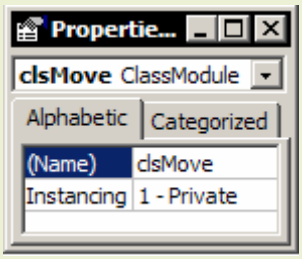
```
Public Function SelectEndUp()
    Dim x As Long, y As Long, z As Long
    x = ActiveCell.Row
    y = ActiveCell.Column
    z = ActiveCell.End(xlUp).Row
    Range(Cells(x, y), Cells(z, y)).Select
End Function
```

```
Public Function SelectEndRight()
    Dim x As Long, y As Long
    x = ActiveCell.Column
    y = ActiveCell.End(xlToRight).Column + 1
    ActiveCell.Resize(, y - x).Select
End Function
```

```
Public Function SelectEndLeft()
    Dim x As Long, y As Long, z As Long
    x = ActiveCell.Row
    y = ActiveCell.Column
    z = ActiveCell.End(xlToLeft).Column
    Range(Cells(x, y), Cells(x, z)).Select
End Function
```

```
Public Function SelectEndDownAndRight()
    Dim x As Long, y As Long
    x = ActiveCell.End(xlDown).Row
    y = ActiveCell.End(xlToRight).Column
    Range(ActiveCell, Cells(x, y)).Select
End Function
```

```
Public Function SelectEndUpAndLeft()
    Dim x As Long, y As Long
    x = ActiveCell.End(xlUp).Row
    y = ActiveCell.End(xlToLeft).Column
    Range(ActiveCell, Cells(x, y)).Select
End Function
```



Next, set the Name property of the Class module to 'clsMove'. Choose *View*, *Properties Window* and enter the relevant text into the property page (you can give the Class any name you prefer) Insert a general module into the project; choose *Insert*, *Module* and then create an instance of the class and, finally, use the methods of the 'Move' object as you normally do by entering the usual Object.Method statements into the code.

Code in the General Module

Create an instance of the class by entering the following statement into the declarations section (the top of the module) of the general module:

```
Dim Move As New clsMove
```

The 'Move' object and all its associated methods are now available in the Complete Word listings. To move the active cell down by one cell in your macro, instead of entering the usual long-winded:

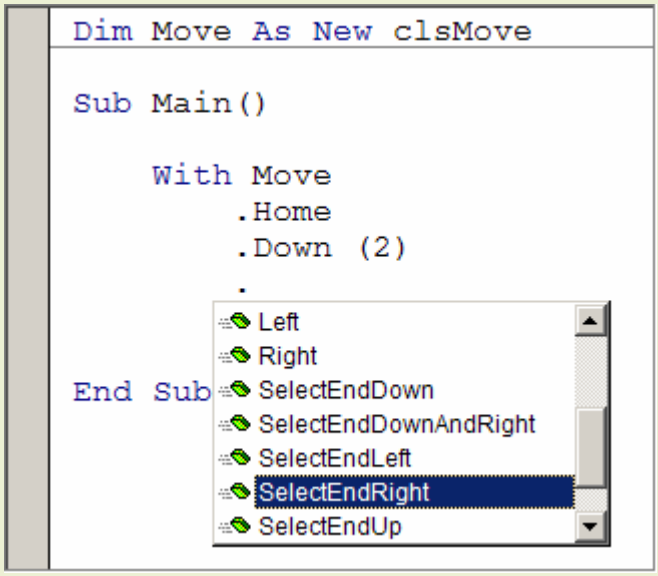
```
ActiveCell.Offset(1, 0).Select
```

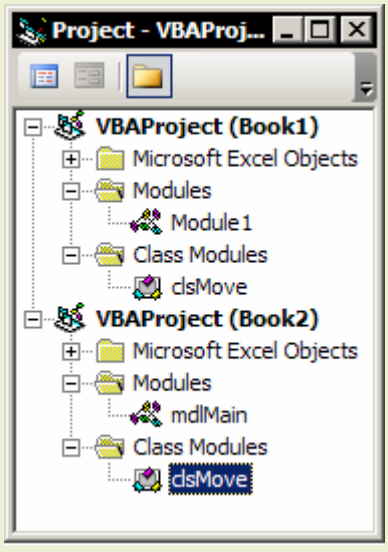
You can enter the simple statement:

```
Move.Down
```

The directional move methods accept an optional number argument where you can specify how many cells you wish to move. To move right by 5 cells, enter the following statement:

```
Move.Right(5)
```





When you want to use the 'Move' object again in another workbook just copy the Class module to the other project; the easiest way to do this is to Drag and Drop the module in the Project Explorer Window.

If you were undertaking extensive translation of Lotus Classic macros it would be worthwhile considering the creation of a Lotus Class module where all the commands could be stored with their relevant equivalents in the Excel VBA language. Then you could enter all your new Excel macros like this:

```
Lotus.GetLabel
Lotus.WindowsOff
```

Whatever purpose you put them to, Class modules are an ideal method of storing all those favourite Excel VBA expressions and constructions that you tend to use time and time again.

File Operations

File operations can be incorporated into your macros by using the statements of the VBA File System Class.

For example

Create a new directory on the current drive.	`MkDir "Data"`
Delete a file on disk.	`Kill "C:\TestData\Test.txt"`
Delete all *.xls files in the current directory.	`Kill "*.xls"`
Remove an existing empty directory.	`RmDir "C:\TestData"`
Change the default directory.	`ChDir "C:\TestData"`
Return the current path.	`Dim strPath As String` `strPath = CurDir`

Opening All files

The following procedure opens all the files in a specific directory, retrieving each file name using the Dir function. Specify the path the first time that you call the Dir function and to retrieve the subsequent file names, call Dir again but with no argument. When no more file names are available, the function returns a zero-length string, "".

```
Sub OpenAllFiles()
    Dim strPath     As String
    Dim strFileName As String

    'Set the path.
    strPath = "C:\Excel_Files\"
    ChDir strPath

    'Retrieve the first entry.
    strFileName = Dir(strPath)

    'File opening loop.
    Do Until strFileName = ""
        'Open the file.
        Workbooks.Open Filename:=strFileName
        'Retrieve the next entry.
        strFileName = Dir
    Loop

End Sub
```

Writing text files

You can save Excel files as text files in a variety of different formats but to really control and manipulate the data to satisfy specialised requirements you have to create a loop to read the cell values and then write the text file directly to disk using the Open, Write and Close statements.

In the following procedure, the cell data in a worksheet has to be written as a continuous string of comma separated values with each entry padded out with space characters to a constant length of 25 characters. Firstly, the cell data is manipulated and stored in the variable 'Data' and then the contents of the variable is written to disk.

```
Sub GenerateTextFile()
   Dim FirstRecord As Boolean
   Dim Data        As String
   Dim CellEntry   As Variant
   Dim Cell        As Range
```

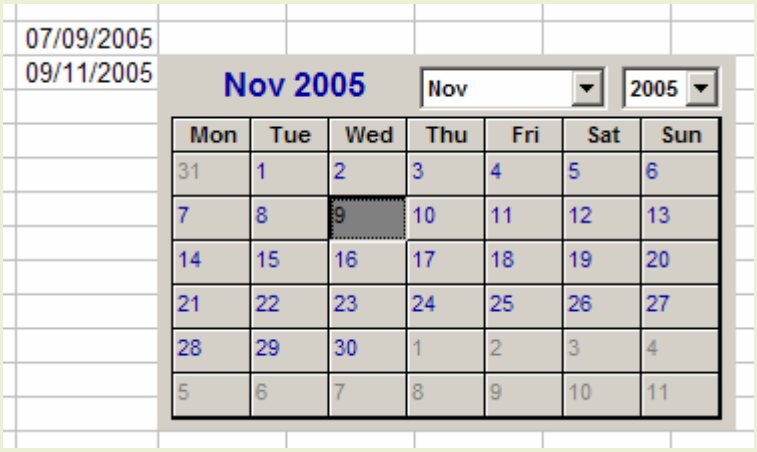
```
    Dim iLen        As Integer
    Dim iNumSpaces  As Integer
    Dim i           As Integer
    Dim FileNumber  As Integer
    Const ENTRYLEN  As Integer = 25

    'Initialise.
    Let Data = ""
    Let FirstRecord = True

    'Loop to create text string.
    For Each Cell In Range("A1").CurrentRegion

      'Store the cell value.
      Let CellEntry = Cell.Value

      'Coerce numbers to text.
      If IsNumeric(CellEntry) Then
         CellEntry = Application.WorksheetFunction.Text(CellEntry, "0")
      End If

      'Pad the entry with spaces.
      Let iLen = Len(CellEntry)
      If iLen < ENTRYLEN Then
          iNumSpaces = ENTRYLEN - iLen
          For i = 1 To iNumSpaces
              CellEntry = CStr(CellEntry) & " "
          Next
      ElseIf iLen > ENTRYLEN Then
          'Reduce to 25 characters if over.
          CellEntry = Left(CellEntry, ENTRYLEN )
      End If

      'Write the text string.
      If FirstRecord Then
          Data = CellEntry
      Else
          Data = Data & "," & CellEntry
      End If
      Let FirstRecord = False

  Next

  'Write the text file data to disk.
  FileNumber = FreeFile
  Open "C:\Dump\TEST.TXT" For Output As #FileNumber
  Write #FileNumber, Data
  Close #FileNumber

End Sub
```

Make sure the text file exists before you attempt to write data to it. It is quite in order to use an application like Windows Notepad to create a text file containing no data. The text output into the file would appear like this:

"UK ,North ,Soap ,1789 ,81460
,Jan ,PR960001 , " etc.

Using ActiveX Controls

You can place ActiveX controls directly on the worksheet and control their position, appearance and behaviour using the worksheet's Event procedures. Right-click any visible toolbar and choose *Control Toolbox*.



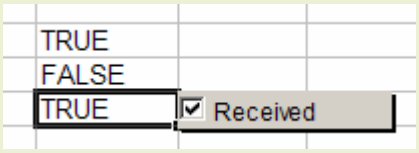
In this example, column D on the worksheet has to have dates entered into the cells. When you select a cell in the column a Calendar control appears, you specify the date and it is entered into the active cell.

Draw the control on the worksheet and then right-click the sheet tab and choose *View Code*. Enter the following event procedures:

```
Private Sub Calendar1_Click()
  ActiveCell.Value = ActiveSheet.OLEObjects("Calendar1").Object.Value
End Sub
```

```
Private Sub Worksheet_SelectionChange(ByVal Target As Range)
  'If ActiveCell is in column D align control to cell and show.
  If ActiveCell.Column = 4 Then
    With ActiveSheet.OLEObjects("Calendar1")
      .Top = ActiveCell.Top
      .Left = ActiveCell.Offset(0, 1).Left
      .Visible = True
    End With
  Else
     'Otherwise hide the control.
     ActiveSheet.OLEObjects("Calendar1").Visible = False
  End If
End Sub
```

This example is exactly the same as the previous but uses a Check Box control. The object names are all shown in the Object list (top right hand side) of the sheet module.



This example shows a list box when the cell is selected, the list box contains a list of currencies. As you select a currency the corresponding exchange rate is entered into the active cell. The *ListFillRange* property of the control refers to a range of cells on the worksheet containing foreign exchange data.

USD	1.74
EUR	1.39
DKK	11.2

The *BoundColumn* property of the list box control is set to the value of 2 so that the control returns the value in the second column of the range of cells, the actual exchange rate rather than the name of the currency. The click event procedure for the list box is not necessary as the *Worksheet_SelectionChange* event contains the following statement which links the active cell to the list box control to return the relevant value into the cell.

```
ActiveSheet.OLEObjects("ListBox1").LinkedCell = ActiveCell.Address
```

When you have finished setting all the control object properties, click the *Exit Design Mode* control (Set square, ruler and pencil) to activate the controls. Please note that it is

also possible to achieve similar interactive effects in worksheet cells by using *Data*, *Validation* in the main Excel menu. Less sophisticated but much easier.

Using the Windows API

You have access to the Windows Application Programming Interface through VBA and you can use the WIN API to control your system: manage the display of windows, communicate with other devices, return information about the operating system, available memory etc. There are hundreds of functions that you can call but you will not find any documentation on these in Excel, you must search elsewhere.

When you have discovered the documentation then you must correctly implement the function call in your VBA procedure. The VBA compiler does not recognise WIN API functions so you must include a Declare statement in your module declarations section (top of the module) directing the compiler where to find the function. Then you call the function in your procedure taking particular care that you match the required data types.

In the following example we are using the WIN API function, GetUserName to retrieve the registered user name from the system:

```
Private Declare Function GetUserName Lib "advapi32.dll" Alias _
         "GetUserNameA" (ByVal lpBuffer As String, nSize As Long) As Long
```

```
Sub MyGetUserName()
  Dim Buffer As String * 25
  Dim ReturnValue As Long, UserName As String

  ReturnValue = GetUserName(Buffer, 25)
  UserName = Left(Buffer, InStr(Buffer, Chr(0)) - 1)
  MsgBox UserName

End Sub
```

The user name is retrieved into the variable 'Buffer', which is a 25 character length string. Any unnecessary characters are then stripped out. All the WIN API functions have to be used in the function form, so you need to assign the function to a variable, in this case the variable 'ReturnValue'. The value of the variable has no particular use other than to test whether the function has failed or not.

There are a number of books available on the WIN API and you can also search in the Microsoft Knowledge Base. The information that you need to find is the name of the function required, how to properly declare the function and (hopefully) an example that you can copy.

Case Studies

Case Study 1. Using the Personal Workbook

Recording a macro in the Personal Macro Workbook to hide error values in worksheet cells. A Custom Menu Item in Excel's Format menu triggers the macro.

```
Sub HideErrorValues()
      Selection.Font.ColorIndex = 2
      Selection.NumberFormat = "[Black] General"
End Sub
```

Intended to hide divide by zero errors (#DIV/0!) the macro will hide all cell error values by changing the Font colour to white and forcing numbers to Black in the General Number Format. To be really effective the macro should be more sophisticated and take into account the existing cell number format and font colour.

Case Study 2. Looping through Cells

Inserting blank rows into an Excel list.

```
Sub InsertIntoList()

      [A2].Select

      Do Until ActiveCell = ""

            If ActiveCell = ActiveCell.Offset(1, 0) Then
                  ActiveCell.Offset(1, 0).Select
            Else
                  ActiveCell.Offset(1, 0).EntireRow.Insert
                  ActiveCell.Offset(2, 0).Select
            End If
      Loop

End Sub
```

Case Study 3. Processing a Text File

Breaking down the process into subroutines.

```
Public Sub Main()

      'Loop to examine all rows.
      With Application
            .ScreenUpdating = False
            .EnableCancelKey = xlDisabled

            [D1].Select

            x = ActiveSheet.UsedRange.Rows.Count

            For i = 1 To x
                  Call Finder
                  .StatusBar = Format(i / x, "0%") & " Complete."
            Next

            .ScreenUpdating = True
            .StatusBar = False

      End With

End Sub
```

Procedures continue overleaf.

```
Private Sub Finder()
      'Len returns the length in characters of an expression. Trim removes
      'leading and trailing space characters.

      'Locate 4 character codes.
      If Len(Trim((ActiveCell.Offset(0, -3))) = 4 Then
            Call Copier
      Else
            ActiveCell.EntireRow.Delete
      End If

End Sub
```

```
Private Sub Copier()

      'Copy cell values.
      With ActiveCell
            .Offset(0, 0) = .Offset(1, -2)
            .Offset(0, 1) = .Offset(1, -1)
            .Offset(1, 0).Select
      End With

End Sub
```

Case Study 4. Writing a Loop

Adjusting the width of alternate columns on a worksheet.

```
Sub AlternateColumnsConcrete()

      x = ActiveSheet.UsedRange.Columns.Count
      [A1].Select
      For i = 1 To x Step 2
            With ActiveCell
                  .ColumnWidth = 10
                  .Offset(0, 1).ColumnWidth = 5
                  .Offset(0, 2).Select
            End With
      Next

End Sub
```

```
Sub AlternateColumnsAbstract()

      x = ActiveSheet.UsedRange.Columns.Count

      For i = 1 To x Step 2
            Columns(i).ColumnWidth = 10
            Columns(i + 1).ColumnWidth = 5
      Next

End Sub
```

See overleaf for the next model answer.

```
Sub AlternateColumnsOddEven()

      x = ActiveSheet.UsedRange.Columns.Count

      For i = 1 To x
            If i Mod 2 = 0 Then
                  'Column number is even.
                  Columns(i).ColumnWidth = 5
            Else
                  'Column number is odd.
                  Columns(i).ColumnWidth = 10
            End If
      Next

End Sub
```

In the last procedure we needed to determine if a column number was an even number. We tested for modulo 2, is the number divisible by 2, leaving a remainder of zero? The modulus, or remainder operator, Mod is invaluable for any type of interval calculation. For example, performing a certain action every fifth iteration of a For...Next loop.

Case Study 5. Using Control Structures

The workbook must have exactly 12 worksheets. You may have any number of worksheets when you start but you end up with 12. No specific order is required.

```
Sub ExactlyTwelveSheetsCaseStatement()
      Dim iNumShts      As Integer
      Dim i             As Integer
      Const TARGET_SHTS As Integer = 12

      'Count the sheets.
      iNumShts = Worksheets.Count
      Select Case iNumShts
            Case TARGET_SHTS
                  Exit Sub
            'Add if too few.
            Case Is < TARGET_SHTS
                  Worksheets.Add Count:=TARGET_SHTS - iNumShts
            'Delete if too many.
            Case Is > TARGET_SHTS
                  With Application
                        .DisplayAlerts = False
                        For i = 1 To iNumShts - TARGET_SHTS
                              Worksheets(1).Delete
                        Next
                        .DisplayAlerts = True
                  End With
      End Select

End Sub
```

See overleaf for the next model answer.

```
Sub ExactlyTwelveSheetsIfThenElse()
      Dim iNumShts      As Integer
      Dim i             As Integer
      Const TARGET_SHTS As Integer = 12

      'Count the sheets.
      iNumShts = Worksheets.Count
      'Add sheets if too few.
      If iNumShts < 12 Then
            Worksheets.Add Count:=TARGET_SHTS - iNumShts
      'Delete sheets if too many.
      ElseIf iNumShts > 12 Then
            With Application
                  .DisplayAlerts = False
                  For i = 1 To iNumShts - TARGET_SHTS
                        Worksheets(1).Delete
                  Next
                  .DisplayAlerts = True
            End With
      End If

End Sub
```

```
Sub ExactlyTwelveSheetsDoLoop()
      Dim iNumShts      As Integer
      Dim i             As Integer
      Const TARGET_SHTS As Integer = 12

      'Count the sheets.
      iNumShts = Worksheets.Count
      Application.DisplayAlerts = False
      Do Until iNumShts = TARGET_SHTS
            'Add a sheet if too few.
            If iNumShts < 12 Then
                  Worksheets.Add
                  iNumShts = Worksheets.Count
            'Delete a sheet if too many.
            ElseIf iNumShts < 12 Then
                  Worksheets(1).Delete
                  iNumShts = Worksheets.Count
            End If
      Loop
      Application.DisplayAlerts = True

End Sub
```

```
Sub DeleteThenInsert()
      Dim i As Integer

      Application.DisplayAlerts = False
      'Delete all sheets except for one.
      For i=1 To Worksheets.Count-1
            Worksheets(1).Delete
      Next
      'Then add 11 to make 12.
      Worksheets.Add Count:= 11

End Sub
```

Case Study 6. Declaring and Typing Variables

Option Explicit is entered in the Declarations Section, you must declare your variables.

```
Option Explicit
```

Faulty Code:

```
Sub Main()

      x = 1.54
      y = 5000
      NewSht = Worksheets.Add(After:=Worksheets(1))
      MyArea = Worksheets(1).UsedRange

End Sub
```

Corrected:

```
Sub Main()
      Dim x       As Double
      Dim y       As Integer
      Dim NewSht  As Worksheet
      Dim MyArea  As Range

      x = 1.54
      y = 5000
      Set NewSht = Worksheets.Add(After:=Worksheets(1))
      Set MyArea = Worksheets(1).UsedRange

End Sub
```

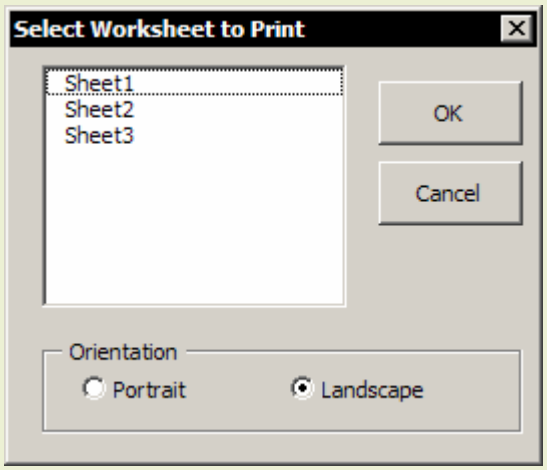
Case Study 7. Creating an Add-In Function

Creating an Add-In function for Excel to validate table calculations. Create the procedure and then generate an Add-In from the module.

```
Function CheckSum(Row_Totals,Column_Totals)

      x = Application.Sum(Row_Totals)
      y = Application.Sum(Column_Totals)

      If x <> y Then
            CheckSum = "BADSUM!"
      Else
            CheckSum = x
      End If

End Function
```

Case Study 8. Creating a User Form

Design and Code the following User Form.



The list has to show all the Worksheets in the Workbook. You select a sheet from the list, click the OK button and that sheet is printed in the orientation of your choice.

Double clicking an item in the list should have same effect as selecting and clicking the OK button.

Closing the dialog or clicking the Cancel button should cancel the entire process.

Landscape should be the default orientation setting when the Form is initially displayed.

Code in the General Module

```
Public g_intSheetIndex  As Integer
Public g_bolLandscape   As Boolean
Public g_bolPrintReport As Boolean
```

```
Public Sub PrintSelectedWorksheet()

    g_intSheetIndex = 1
    g_bolLandscape = True
    g_bolPrintReport = False

    frmPrintReport.Show

    If Not g_bolPrintReport Then
            Exit Sub
    End If

    With Worksheets(g_intSheetIndex)

            With .PageSetup
                  If g_bolLandscape Then
                        .Orientation = xlLandscape
                  Else
                        .Orientation = xlPortrait
                  End If
            End With

            .PrintOut

    End With

End Sub
```

See overleaf for the code in the User Form object.

Code in the Form Object Module

```
Private Sub UserForm_Initialize()
  Dim WSht As Worksheet

  For Each WSht In Worksheets
      lstWorksheets.AddItem WSht.Name
  Next
  optLandscape.Value = True
  optPortrait.Value = False

End Sub
```

```
Private Sub cmdOKButton_Click()
    g_bolPrintReport = True
    Unload Me
End Sub
```

```
Private Sub cmdCancelButton_Click()
    g_bolPrintReport = False
    Unload Me
End Sub
```

```
Private Sub lstWorksheets_Click()
  'Note the adjustment required for zero base.
    g_intSheetIndex = lstWorksheets.ListIndex + 1
End Sub
```

```
Private Sub lstWorksheets_DblClick(ByVal Cancel As
    MSForms.ReturnBoolean)

    g_intSheetIndex = lstWorksheets.ListIndex + 1
    g_bolPrintReport = True
    Unload Me

End Sub
```

```
Private Sub optLandscape_Click()
    g_bolLandscape = True
End Sub
```

```
Private Sub optPortrait_Click()
    g_bolLandscape = False
End Sub
```

```
Private Sub UserForm_QueryClose(Cancel As Integer, _
     CloseMode As Integer)

    If CloseMode <> vbFormCode Then
       g_bolPrintReport = False
    End If

End Sub
```

Case Study 9. Handling Workbook files

When the file opens, update the history data file from external documents. Match the country data from each file to the country summary in the target file and copy the data into the correct column based on the current calendar date. Assume that the file name is always good and that the data is up to date.

This Case Study practices manipulating arrays and writing a complex loop.

```
Option Explicit

Public Sub ConsolidateDataFromFiles()

      Dim vRegions      As Variant
      Dim vRegion       As Variant
      Dim oTargetBook   As Workbook
      Dim oTargetSheet  As Worksheet
      Dim oMatchRange   As Range
      Dim oTargetRange  As Range
      Dim oSourceBook   As Workbook
      Dim oSourceSheet  As Worksheet
      Dim oSourceRange  As Range
      Dim sFileName     As String
      Dim iSourceRowLen As Integer
      Dim iRowIndex     As Integer
      Dim iColIndex     As Integer
      Dim i             As Integer

      Const PATH_NAME As String = "C:\My Documents"

      With Application
            .ScreenUpdating = False
            .EnableCancelKey = xlDisabled
      End With

      'Identify target column as today's date.
      iColIndex = Day(Date)

      'List of file names.
      vRegions = Array ("Africa", _
               "Asia Pacific", _
               "Middle East", _
               "Western Europe", _
               "Eastern Europe", _
               "North America", _
               "Latin America")

      'Initialise objects.
      Set oTargetBook = ThisWorkbook
      Set oTargetSheet = oTargetBook.Worksheets(1)
      Set oMatchRange = oTargetSheet.Range("CountryNames")
      Set oTargetRange = oTargetSheet.Range("DataTable")

      'Point to the directory where files are stored.
      ChDrive Left(PATH_NAME, 3)
      ChDir PATH_NAME
```

Procedure continues overleaf…

```
   'Loop through each file.
      For Each vRegion In vRegions

            'Identify the file name.
            sFileName = vRegion & ".xls"
            'Progress message.
            Application.StatusBar = _
                  "Loading data from " & vRegion & " , please wait."

            'Open the file.
            Set oSourceBook = Workbooks.Open( _
                        FileName:=PATH_NAME & sFileName)
            Set oSourceSheet = oSourceBook.Worksheets(1)

            'Measure the data set, less the header row.
            iSourceRowLen = _
                  oSourceSheet.Cells(1, 1). _
                        CurrentRegion.Rows.Count - 1
            Set oSourceRange = _
                        oSourceSheet.Range(Cells(2, 1),_
                              Cells(iSourceRowLen, 2))

            'Loop through the cells.
            For i = 1 To iSourceRowLen
                  'Locate the row in target document.
                  iRowIndex = Application.WorksheetFunction.Match _
                        (oSourceRange.Cells(i, 1), oMatchRange, 0)
                  'Copy the data.
                  oSourceRange.Cells(i, 2).Copy _
                        oTargetRange.Cells(iRowIndex, iColIndex)
            Next i

            'Close the Source file.
            With oSourceBook
                  .Saved = True
                  .Close
            End With

            'Destroy Objects.
            Set oSourceRange = Nothing
            Set oSourceSheet = Nothing
            Set oSourceBook = Nothing

   Next vRegion

      'Save the Target file.
      oTargetBook.Save

      'Destroy Objects.
      Set oTargetRange = Nothing
      Set oMatchRange = Nothing
      Set oTargetSheet = Nothing
      Set oTargetBook = Nothing

      'Confirmation message.
      MsgBox "Updates for " & Format(Date, "dddd d MMMM yyyy") _
                  & vbCr & "were sucessfully completed.", _
                  Buttons:=vbInformation, Title:="Data Updated"

      With Application
            .ScreenUpdating = True
            .StatusBar = False
      End With
End Sub
```

Case Study 10. Refreshing Pivot Tables

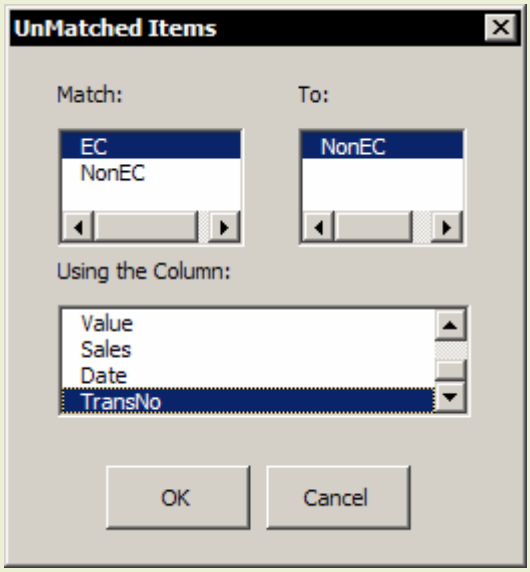
Automatically Refresh all Pivot Tables every 30 seconds.

```
Sub Auto_Open()
    Application.OnTime Now + TimeValue("00:00:30"), "RefreshData"
End Sub
```

```
Sub RefreshData()
      Dim wSheet As Worksheet
      Dim pTable As PivotTable

      With Application
            .DisplayStatusBar = True
            .StatusBar = "Refreshing Pivot Tables..."
            For Each wSheet In Worksheets
                  For Each pTable In wSheet.PivotTables
                        pTable.RefreshTable
                  Next
            Next
            .StatusBar = False
      End With

      Call Auto_Open

End Sub
```

Case Study 11. Unmatched Items

Design and code the following User Form:



The macro is designed to compare two worksheets containing lists in the same workbook and detect items in a common column that are not matched on the other worksheet.

The top two list boxes should show all the worksheets in the workbook but when you select a worksheet in the "Match:" list box then that worksheet should not be displayed in the "To:" list box.

The "Using the Column:" box is populated by the values in the header row of the "Match" worksheet.

The macro produces an exception report on a new worksheet which is inserted at the end of the workbook.

Each item on the exception report should give the record details and the row reference of the unmatched item.

The case study has two sections: the first section is the graphical design of the User Form and the corresponding procedures to populate the list boxes and validate the user's choices.

The second section is the main process in the general module; to show the User Form, to terminate the procedure if the Cancel button is clicked and to carry out the matching process and report generation if the OK button is clicked.

The matching process is carried out using Excel's MATCH function. The two ranges to match are defined and when an unmatched item is found its details are recorded in the exception report.

Code in the General Module

```
Option Explicit

Public g_BaseSheet           As String
Public g_CompareSheet        As String
Public g_MatchColumnNumber   As Integer
Public g_CompareColumnNumber As Integer

Public Sub UnMatchedItems()
  Dim wksMatch       As Worksheet
  Dim wksTo          As Worksheet
  Dim wksReport      As Worksheet
  Dim rngMatch       As Range
  Dim rngTo          As Range
  Dim rngCell        As Range
  Dim rngRecordID    As Range
  Dim rngCopy        As Range
  Dim rngDestination As Range
  Dim dblThisRow     As Double
  Dim dblNextRow     As Double
  Dim MatchItem      As Variant

  'Show User Form.
  frmMatcher.Show

  'Process User Form selections.
  Select Case frmMatcher.cmdOK.Tag
    Case False

      'Form cancelled.
      Unload frmMatcher
      'Terminate macro.
      GoTo UnMatchedItems_Exit

    Case True

      'Initialise Objects.
      Set wksMatch = Worksheets(g_BaseSheet)
      Set wksTo = Worksheets(g_CompareSheet)
      Set wksReport= _
         Worksheets.Add(After:=Worksheets(Worksheets.Count))

      'Enter title on exception report sheet.
      wksReport.Cells(1) = "Exception report; items on " _
                & g_BaseSheet & _
              " with no matching item on " & g_CompareSheet

      'The base range to match.
      With wksMatch
        Set rngMatch = .Range(.Cells(2, g_MatchColumnNumber), _
          .Cells(.Cells(1).CurrentRegion.Rows.Count, _
           g_MatchColumnNumber))
      End With

      'The range to match the base range to.
      With wksTo
        Set rngTo = .Range(.Cells(2, g_CompareColumnNumber), _
          .Cells(.Cells(1).CurrentRegion.Rows.Count, _
           g_CompareColumnNumber))
      End With
```

Procedure continues overleaf…

```
        'Loop to find unmatched, the MATCH function returns an error
        'when a match is not found. Record the details of each error.
        On Error GoTo UnMatchedItem
        For Each rngCell In rngMatch
          Let MatchItem = _
            Application.WorksheetFunction.Match(rngCell, rngTo, 0)
        Next

           'Destroy objects.
        Set rngCell = Nothing
        Set rngRecordID = Nothing
        Set rngCopy = Nothing
        Set rngDestination = Nothing
        Set rngMatch = Nothing
        Set rngTo = Nothing
        Set wksMatch = Nothing
        Set wksTo = Nothing
        Set wksReport = Nothing

        'Unload the User Form, it is hidden but still loaded.
        Unload frmMatcher

    End Select

    Exit Sub

UnMatchedItem:

    'Store the row reference number.
    Let dblThisRow = rngCell.Row

    'Find the next free row on the exception report.
    Let dblNextRow = wksReport.Cells(1).CurrentRegion.Rows.Count + 1

    'Enter the row reference data into the exception report.
    Set rngRecordID = wksReport.Cells(dblNextRow, 1)
    rngRecordID.Value = "Row " & dblThisRow

    'The range to copy.
    With wksMatch
      Set rngCopy = .Range(.Cells(dblThisRow, 1), _
        .Cells(dblThisRow, .Cells(1).CurrentRegion.Columns.Count))
    End With

    'The range to copy it to.
    Set rngDestination = wksReport.Cells(dblNextRow, 2)

    'Copy the record data.
    rngCopy.Copy Destination:=rngDestination

    'Go back into the matching loop.
    Resume Next

UnMatchedItems_Exit:

    'This is the main exit point from the procedure.


End Sub
```

Code in the Form Object Module

```
Option Explicit

Dim m_MatchDescription  As String
Dim Sheet               As Worksheet

Private Sub UserForm_Initialize()
  'Initialise controls.
  cmdOK.Tag = False

  For Each Sheet In Worksheets
    lstBase.AddItem Sheet.Name
    lstCompare.AddItem Sheet.Name
  Next
End Sub

Private Sub cmdOK_Click()
  Dim strErrorMessage As String
  Dim bHeaderFound    As Boolean
  Dim iColCount       As Integer
  Dim i               As Integer
  Const ZLS           As String = ""

  'Validation test #1. That both sheets were specified.
  If g_BaseSheet = ZLS Then
    Let strErrorMessage = "You did not specify the Match worksheet."
    GoTo cmdOK_Click_Exit
  ElseIf g_CompareSheet = ZLS Then
    Let strErrorMessage = "You did not specify the To worksheet."
    GoTo cmdOK_Click_Exit
  End If

  'Validation test #2. That the sheets are different.
  If g_BaseSheet = g_CompareSheet Then
    Let strErrorMessage = "You must specify different worksheets."
    GoTo cmdOK_Click_Exit
  End If

  'Validation test #3. That the row header was specified.
  If g_MatchColumnNumber = 0 Then
    Let strErrorMessage = "You did not specify the Column to Match"
    GoTo cmdOK_Click_Exit
  End If

  'Validation test #4. That the row header is found in the compare sheet.
  Let bHeaderFound = False
  Let iColCount = _
     Worksheets(g_CompareSheet).Cells(1).CurrentRegion.Columns.Count

  For i = 1 To iColCount
    If m_MatchDescription = Worksheets(g_CompareSheet).Cells(1, i) Then
      Let bHeaderFound = True
      Let g_CompareColumnNumber = i
      Exit For
    End If
  Next

  If Not bHeaderFound Then
    Let strErrorMessage = _
        "Could not find a matching column in the To worksheet."
    GoTo cmdOK_Click_Exit
  End If
```

Procedure continues overleaf:

```
  'Input validated; proceed to main process.
  cmdOK.Tag = True
  Me.Hide

  Exit Sub

cmdOK_Click_Exit:
  MsgBox strErrorMessage, vbCritical + vbOKOnly, "Invalid Input"

End Sub
```

```
Private Sub cmdCancel_Click()
  cmdOK.Tag = False
  Me.Hide
End Sub
```

```
Private Sub lstBase_Click()
  Dim iColCount As Integer
  Dim i         As Integer

  Let g_BaseSheet = lstBase.Text

  'Repopulate compare list box to exclude selected item.
  lstCompare.Clear

  For Each Sheet In Worksheets
    If Not Sheet.Name = g_BaseSheet Then
      lstCompare.AddItem Sheet.Name
    End If
  Next

  'Populate header row list box with row headers.
  Let iColCount = _
    Worksheets(g_BaseSheet).Cells(1).CurrentRegion.Columns.Count

  lstHeaderRow.Clear

  For i = 1 To iColCount
    lstHeaderRow.AddItem Worksheets(g_BaseSheet).Cells(1, i)
  Next

End Sub
```

```
Private Sub lstCompare_Click()
  Let g_CompareSheet = lstCompare.Text
End Sub
```

```
Private Sub lstHeaderRow_Click()
  Let g_MatchColumnNumber = lstHeaderRow.ListIndex + 1
  Let m_MatchDescription = lstHeaderRow.Text
End Sub
```

The Object names used in the procedures are:

User Form	frmMatcher
OK Button	cmdOK
Cancel Button	cmdCancel
Left hand worksheets list box	lstBase
Right hand worksheets list box	lstCompare
Lower list box	lstHeaderRow

Index