MICROSOFT EXCEL
VISUAL BASIC FOR APPLICATIONS
ADVANCED
NOTE

- Unless otherwise stated, screenshots of dialog boxes and screens in this book were taken using Excel 2003 running on Window XP Professional. There may, therefore, be minor differences if you are using other versions of Excel, or if you are running on Windows 2000, Windows Vista or Windows 7.

- Concepts, discussions, procedures and functionality, however, remain unchanged.
# Contents

NOTE .................................................................................................................................................. 2  

LESSON 1 – ERROR HANDLING & TRAPPING .............................................................................. 5
  ERROR HANDLING USING IF ........................................................................................................ 6
  ERROR TRAPPING .......................................................................................................................... 8
  TRAPPING ERRORS WITH ERR NUMBERS .................................................................................. 12
    Exercise ....................................................................................................................................... 17

LESSON 2 - BUILT-IN DIALOG BOXES AND CUSTOM USERFORMS ........................................... 19
  EXCEL DIALOG BOXES ................................................................................................................ 20
  USER-DEFINED FORMS .................................................................................................................. 20
  INSERTING A USERFORM INTO A WORKBOOK .......................................................................... 21
  ADDING CONTROLS TO A FORM ................................................................................................... 21
  FORM CONTROL DESIGN TOOLS AND TECHNIQUES ............................................................. 24
  CONTROL PROPERTIES ............................................................................................................... 25
  PROGRAMMING A USERFORM ..................................................................................................... 27
  FORM EVENTS .............................................................................................................................. 31
  DISPLAYING A USERFORM .......................................................................................................... 32
    Exercise ....................................................................................................................................... 34

LESSON 3 – ACTIVEX WORKSHEET CONTROLS .................................................................... 37
  OVERVIEW AND DEFINITION ....................................................................................................... 38
  ACTIVEX CONTROLS ..................................................................................................................... 39
  CONTROL PROPERTIES .................................................................................................................. 40
  ADDING ACTIVEX CONTROLS TO A WORKSHEET ..................................................................... 41
  MOVING A CONTROL ...................................................................................................................... 43
  SIZING A CONTROL ......................................................................................................................... 44
  DELETING A CONTROL .................................................................................................................... 44
  RETURNING A CONTROL’S VALUE .............................................................................................. 44
  CREATING A LISTBOX ................................................................................................................... 48
  CREATING OPTION BUTTONS ....................................................................................................... 49
  CREATING A SPIN BUTTON ............................................................................................................ 50
  CREATING A COMBO BOX .............................................................................................................. 51
    Exercise ....................................................................................................................................... 53

LESSON 4 – EVENT PROCEDURES ......................................................................................... 54
  OVERVIEW ..................................................................................................................................... 55
  TYPES OF EVENT ............................................................................................................................ 55
  WRITING CODE TO EVENTS .......................................................................................................... 57
  EVENT PROCEDURE ARGUMENTS ............................................................................................... 59
  COMMON WORKBOOK EXAMPLES ............................................................................................... 60
  NON OBJECT-RELATED EVENTS (OnTime) ..................................................................................... 62
    Exercise ....................................................................................................................................... 64

LESSON 5 – WORKING WITH CHARTS ..................................................................................... 65
  OVERVIEW ..................................................................................................................................... 66
  CREATING CHARTS .......................................................................................................................... 66
  SIZING AND POSITIONING CHARTS ............................................................................................... 67
  SIZING AND POSITIONING MULTIPLE CHARTS .......................................................................... 70
  NAMING A CHART ............................................................................................................................ 71
  ADDING A SERIES TO A CHART ...................................................................................................... 72
  DELETING SERIES FROM A CHART ................................................................................................. 73
    Exercise ....................................................................................................................................... 74

LESSON 6 – WORKING WITH PIVOT TABLES ......................................................................... 75
  THE PIVOT TABLE OBJECT .......................................................................................................... 76
  CREATING A PIVOT TABLE ............................................................................................................ 77
  NAMING PIVOT TABLES .................................................................................................................. 79
  MANIPULATING PIVOT TABLE FIELDS ........................................................................................... 80
REFRESHING PIVOT TABLES ................................................................. 82
  Exercise ........................................................................ 84

LESSON 7 – COMMUNICATING WITH OTHER OFFICE APPLICATIONS ............. 86
  OVERVIEW ........................................................................ 87
  ADDING A REFERENCE TO THE APPLICATION’S OBJECT LIBRARY .......... 87
  DECLARING THE OBJECT VARIABLE .................................... 88
  SETTING THE VARIABLE ................................................... 89
  USE METHODS AND PROPERTIES TO AUTOMATE THE OTHER APPLICATION ........................................... 90
  EXAMPLES ....................................................................... 90
  Exercise ........................................................................ 91

APPENDIX I – LIST OF TRAPPABLE ERRORS AND THEIR CODES ......................... 92
APPENDIX II – ADDING INTERACTIVITY TO A MESSAGE BOX ............................. 96
APPENDIX III – SOLUTIONS TO EXERCISES ...................................................... 101
INDEX ................................................................................ 115
In this lesson, you will learn how to:

- Handle errors using an If statement
- Trap errors using the On Error statement
- Trap errors by means of their error codes
**ERROR HANDLING USING IF**

**Discussion**

If an error occurs whilst a macro is running, it will either cause the macro to fail (run-time error), or make it behave unpredictably.

If the error is anticipated, however, the code can be written in such a way that makes the procedure continue to execute, or informs the user what the problem is and gives them the opportunity to take corrective action and continue, or take an alternative course of action.

While it is almost impossible to anticipate every run-time error that might occur, a good developer will always add some error handling code which as a minimum, terminates the procedure with a user-friendly message instead of the infamous and often inexplicable **Run-time error** window!

The simplest way to deal with errors, if possible, is to use an **If... Then... Else** statement. This can check for a criteria and allow execution of the rest of the code only if that criteria is true, otherwise, it could terminates the procedure or make it follow a different course of action.

Consider the examples below:

```vba
If Selection.Cells.Count >1 Then
    ...code
Else
    MsgBox “This macro requires more than one cell to be selected.”
    Exit Sub
End If
```

Or,

```vba
If ActiveCell.Address = “$A$1” Then
    ...code
Else
    MsgBox “Incorrect cell selected. Select A1 and run again”
    Exit Sub
End If
```

In both cases, the selection is tested and if true, a message is displayed to explain the problem before terminating execution of the procedure.

In the following example, an input box is displayed for the user to enter a number. If the user accidentally enters a text value or clicks Cancel (both string values), a run-time error occurs when the procedure attempts to use the string in the calculation.

```vba
interestRate = InputBox(“Enter today’s interest rate”)
Range(“B1”).Value = Range(“A1”).Value * interestRate
```
Type mismatch run-time error

By using an If... Then... Else function, together with the IsNumeric function to check that the input is valid before attempting the calculation, the run time error can be handled in a more user-friendly way, eg:

```vba
interestRate = InputBox("Enter today's interest rate")

If IsNumeric(interestRate) And interestRate >= 0 Then
    Range("B1").Value = Range("A1").Value * interestRate
Else
    MsgBox "Input is not valid or user clicked Cancel"
End If
```

There are several Is... functions in VBA that can be used in a similar way:

- IsArray
- IsDate
- IsEmpty
- IsError
- IsMissing
- IsNull
- IsObject

Procedures

1. Launch or switch to the VB Editor.
2. Identify in the Project Explorer pane, the workbook (VBA project) that you want to add code to.
3. Open the module sheet containing the code or, insert a new module.
4. Position the cursor in the procedure where you want to add the error handling code.
5. Type If.
6. Enter a valid test to validate a condition encountered by the procedure at that point.
7. Type Then.
8. Press Enter.
9. Type code to execute if the test is true.
10. Press Enter.
11. Type Else.
12. Press Enter.
13. Type code to execute if the test is false.
14. Press Enter
15. Type End If.

**ERROR TRAPPING**

**Discussion**

While If...Then...Else is a good, simple way of testing for correct input, it is not able to handle all errors under all circumstances. In most cases, error trapping code will almost certainly be needed. This comes in three forms:

- **On Error Resume Next**
- **On Error GoTo**
- **Resume**

The error “trap” is “turned on” and “sits in wait” until an error occurs or until the “trap” is “turned off”. The On Error statement is normally written into the procedure just above where the anticipated error may occur. In many cases, this will be at the top of the procedure just below any variable and/ or constant declarations.

**On Error Resume Next**

Some errors can be safely ignored; a procedure encountering non-significant errors can often still execute successfully through to the end.

**On Error Resume Next** tells the procedure to ignore any statements producing a run-time error, and continue with the next statement.

The code below is designed to hide all toolbars. Some toolbars, however, do not have a visible property and the following run-time error will be generated.

```vba
For Each cmdBar In Application.CommandBars
    If cmdBar.Visible = False Then cmdBar.Visible = True
Next cmdBar
```

![Error Message](image.png)
By writing the code as follows, the run-time error window will be ignored and the procedure will successfully continue and hide all the toolbars.

```vba
On Error Resume Next
For Each cmdbar In Application.CommandBars
    If cmdbar.Visible = False Then cmdbar.Visible = True
Next
```

The following example deletes ALL files from three folders (Data1, Data2 and Data3) on the H:\ drive.

```vba
Kill "h:\Data1\*.*"
Kill "h:\Data2\*.*"
Kill "h:\Data3\*.*"
```

If a folder does not contain any files, the following run-time error message will occur:

This is a classic case of where the error can be ignored; if the folder is already empty, then it does not need emptying, so VBA can safely ignore it and continue with the next one.

```vba
On Error Resume Next
Kill "h:\Data1\*.*"
Kill "h:\Data2\*.*"
Kill "h:\Data3\*.*"
```

In the earlier example on page 6 where the user is prompted for a numeric input, On Error Resume Next would negate the need for the If... Then... Else.

```vba
On Error Resume Next
interestRate = InputBox("Enter today's interest rate")
Range("B1").Value = Range("A1").Value * interestRate
```
The result in this case, however, is hardly ideal because although the run-time error message does not appear, the procedure results in nothing happening! This is where a different On Error statement is needed.

**On Error GoTo <label>**

On Error GoTo diverts the code to a specific location further down the procedure (usually at the end just above End Sub) where an "error handling routine" takes over. This location can be marked with a "label" (a made-up word followed by a colon (:)) or a line number.

The error handling routine can just be a message explaining the problem and ending the procedure. Alternatively, it could explain the problem, advice on how to rectify it and provide the opportunity to return to where the error occurred and try again.

In the example on page 9, using On Error GoTo would be as follows:

```vba
On Error GoTo errhandle

interestRate = InputBox("Enter today's interest rate")
Range("B1").Value = Range("A1").Value * interestRate

errhandle:
MsgBox "Invalid Data entered or user clicked Cancel"

End Sub
```

It is true that this code in its current form does not contribute any more than using the If...Then...Else explained on page 7. Its advantage, however, lies in its ability to return to where the error occurred and attempt the statement again. This involves the use of the word **Resume**.

**Resume**

The keyword Resume tells VBA to retry the statement that failed. It can only be used as part of an error handling routine, and is always used on its own (eg. On Error Resume will cause an error). The error handling routine above, therefore, could explain the problem the return the user to try again as follows:

```vba
On Error GoTo errhandle

interestRate = InputBox("Enter today's interest rate")
Range("B1").Value = Range("A1").Value * interestRate

errhandle:
MsgBox "Invalid Data entered or user clicked Cancel"
Resume

End Sub
```
The code above, however, does not give the user an opportunity NOT to return and try again. So the next step in making the error handling routine as user-friendly as possible, is to provide an interactive message box displaying a message explaining the problem, asking whether the user WANTS to try again, and providing two buttons (Yes and No) with which to respond.

Creating an interactive message box is covered in the Excel Introduction to VBA Course. An extract of the course materials are given in Appendix IV on page 96. The example below assumes knowledge and experience of this topic.

```vba
On Error GoTo errhandle
    interestRate = InputBox("Enter today's interest rate")
    Range("B1").Value = Range("A1").Value * interestRate

errhandle:
    response = MsgBox("Invalid Data Entered. Do you want to try again?", vbYesNo)
    If response = vbYes Then
        Resume
    End If
End Sub
```

The final (and very important!) step is to prevent the error handling routine being executed when the procedure runs WITHOUT any errors. This is achieved by including the words Exit Sub immediately above the error routine label. The complete procedure with error handling code is given below:

```vba
Sub CalculateInterest

    Dim interestRate as Single

    On Error GoTo errhandle

    interestRate = InputBox("Enter today's interest rate")
    Range("B1").Value = Range("A1").Value * interestRate

    Exit Sub

errhandle:
    response = MsgBox("Invalid Data Entered. Do you want to try again?", vbYesNo)
    If response = vbYes Then
        Resume
    End If

End Sub
```
Resume Next can be used as an alternative to Resume in an error handling routine. It passes control to the line following the statement that caused the error.

Procedures

1. Launch or switch to the VB Editor.
2. Identify in the Project Explorer pane, the workbook (VBA project) that you want to add code to.
3. Open the module sheet containing the code or, insert a new module.
4. Position the cursor on the line below the Sub statement.
5. Type On Error GoTo errhandle (or other “label” to identify a point further down in the procedure where the procedure must divert to if a runtime error occurs.
6. At the bottom of the procedure and immediately above the End Sub statement, type errhandle (or other label used).
7. Type a colon.
8. Press Enter.
9. Type appropriate error handling code for the anticipated error using the examples above.
10. Position the cursor on the line immediately above the errhandle label.
11. Type Exit Sub.
12. Press Enter.

TRAPPING ERRORS WITH ERR NUMBERS

Discussion

Most run-time errors generate error numbers (see Appendix II on page 37). When the On Error Goto <Label> is used to trap errors, the number of any error is returned by the Err Function that acts like a public variable. The value of Err, therefore, can be tested and acted upon using an If... Then... Else statement.

In the example below, there exists the possibility of two different run-time errors occurring:

- **Err number 13** - due to incorrect data being entered
- **Err number 9** - due to the Interest Calcs sheet not existing in the workbook
Sub CalculateInterest

Dim interestRate as Single
On Error GoTo errhandle

interestRate = InputBox("Enter today’s interest rate")
Sheets("Interest Calcs").Activate
Range("B1").Value = Range("A1").Value * interestRate

Exit Sub
errhandle:
response = MsgBox("Invalid Data Entered. Do you want to try again?", vbYesNo)
If response = vbYes Then
Resume
End If

End Sub

The solution is to substitute the following code as the error handling routine:

errhandle:

If Err.Number = 13 Then
    response = MsgBox("Invalid Data Entered. Do you want to try again?", vbYesNo)
End If

If response = vbYes Then
    Resume
ElseIf Err.Number = 9 Then
    MsgBox ("Sheet not found. Please check sheet names and re-run procedure")
Else
    MsgBox "Unexpected error. Procedure will terminate"
End If

Procedures

1. Launch or switch to the VB Editor.
2. Identify in the Project Explorer pane, the workbook (VBA project) that you want to add code to.
3. Open the module sheet containing the code or, insert a new module.
4. Position the cursor on the line below the Sub statement.
5. Type **On Error GoTo errhandle** (or other “label” to identify a point further down in the procedure where the procedure must divert to if a runtime error occurs.

6. At the bottom of the procedure and immediately above the **End Sub** statement, type `errhandle` (or other label used).

7. Type a **colon**.
8. Press `Enter`.
9. Type **If**.
10. Type a **space**.
11. Type `Err.Number =`.
12. Type the anticipated error number.
13. Type a **space**.
14. Type **Then**.
15. Press `Enter`.
16. Type appropriate code to handle the error generated by the error number in the previous test.
17. Press `Enter`.
18. If it is anticipated that the procedure may generate additional error numbers, type **ElseIf**.
19. Type a **space**.
20. Type `Err.Number =`.
21. Type the anticipated error number.
22. Type a **space**.
23. Type **Then**.
24. Press `Enter`.
25. Type appropriate code to handle the error generated by the error number in the previous test.
26. Press `Enter`.
27. Continue as described in 18 – 26 above for any further anticipated error codes.
28. Press `Enter`.
29. Type **Else**.
30. Press `Enter`.
31. Type `MsgBox “An unexpected error has occurred”`. This is to cover any errors NOT anticipated by the `If`/`ElseIf(s)` above.
32. Press `Enter`.
33. Type **End If**.
34. Position the cursor on the line immediately **above** the `errhandle` label.
35. Type **Exit Sub**.
36. Press **Enter**.

**On Error GoTo 0**
This statement disables a previous **On Error Resume Next** or **On Error Go To Label** statement. When the next error occurs, an error message will be generated and the procedure will fail, eg:

```vba
On Error GoTo 0 ' trap errors from here onwards

Kill "h:\Data1\.*"
Kill "h:\Data2\.*"
Kill "h:\Data3\.*"

On Error GoTo 0 'stop trapping errors from here onwards
```

### Procedures

1. Launch or switch to the **VB Editor**.
2. Identify in the **Project Explorer** pane, the workbook (VBA project) that you want to add code to.
3. Open the module sheet containing the code or, insert a new module.
4. Position the cursor in the procedure where you want the error handling no longer to have an effect.
5. Type **On Error GoTo 0**.
6. Press **Enter**.

**Err.Description**
A common way of creating a user-friendly run time error message is by creating a custom message box quoting the error number and the error description. For example, the code could be as follows (line continuation characters have been used for clarity but are not required):

```vba
MsgBox "The error " & Err.Description & 
" has occurred. Please contact the Helpdesk and quote error number " & Err.Number & ". Thank you."
```
It is true to say that this message box does not say much more than the normal, run-time error window, but it is rather less scary and upon clicking OK, does not potentially leave the procedure hanging in break mode.
EXERCISE

DEALING WITH POTENTIAL ERRORS IN PROCEDURES

Task 1: To effectively handle errors using a simple IF statement.

1. Open the file Error Handle.
2. Run the sub procedure CircData (click the Circle worksheet button), which calculates the circumference and area of a circle.
3. Enter a value of 10 for the radius of the circle.
4. A message box should appear giving you the area and circumference.
5. Run the sub procedure again but just click the Cancel button.
6. Note the run-time error message and click End.
7. Repeat running the sub procedure but this time enter a text value as the radius of the circle.
8. Note the run-time error message and click End.
9. Add a simple IF statement to deal with the above-demonstrated errors.
10. Repeat 5 and/or 7 above and check that the procedure now works correctly.
11. Save and close the file.

Task 2: Trapping errors in a sub procedure.

1. Open the file Trap Error.
2. This workbook contains a macro that is designed to delete a file.
3. The macro already contains a simple IF statement to handle the error generated if the user clicks the Cancel button.
4. Run the macro by clicking the worksheet button on Sheet1 and enter the filename, Remove Me (not case sensitive) into the input box.
5. You should receive confirmation that the file has been successfully deleted.
6. Repeat 4 above. Note the error message and write down the error number.
7. Add code to deal with the error in a more user-friendly way. The code should:
   a. display a message explaining the error (eg. File not found);
   b. offer options to either try again or cancel the task;
   c. return control to where the error occurred, if applicable.
8. Run and test the sub procedure again. Correct any errors.
9. Open the file Remove Me Too.
10. Leave this file open and switch back to the **Trap Error** workbook.

11. Run the **Delete Old File** macro and attempt to delete **Remove Me Too**. Is your previous code handling the error correctly?

12. De-activate the error handling code by commenting out the **On Error GoTo** statement at the top of your procedure.

13. Run the **Delete Old File** macro again.

14. Note the error message. How does it differ from the one displayed in 4 above? (Tip: note the error number)

15. Edit the **DeleteOldFile** sub procedure with some additional error-trapping code to deal with both potential errors in a more user-friendly way.

16. Re-activate the error handling code at the top of the procedure.

17. Test the macro by attempting again to delete **Remove Me Too**.

18. The file should be successfully deleted.

19. Save and close all open files.

**BEWARE** - files deleted by a macro **cannot** be recovered.
LESSON 2 -
BUILT-IN DIALOG BOXES AND CUSTOM USERFORMS

In this lesson, you will learn how to:

- Display built-in dialog boxes
- Create and display user defined forms
- Use controls on user forms
- Add code to create event procedures for the user form
**Excel Dialog Boxes**

**Discussion**

Dialog boxes allow applications to interact with their users. A built-in Excel dialog box can be used in a procedure giving a quick, easy way to request information from, or display information to, the user.

Excel contains approximately 200 built-in dialog boxes. Each dialog box is identified by a constant (referred to as enumerators). These constants are all prefixed with xlDialog. Use the Object Browser to browse the list of dialog box constants or pick from the list of constants displayed when typing in the VB Editor.

The *Show* method of the *Dialogs* property displays and executes any action taken in a built-in Excel dialog box. To access a particular built-in Excel dialog box, specify an xlDialog constant with the *Dialogs* property of the *Application* object. For example, the following line of code displays the Save As dialog box, e.g:

```
Application.Dialogs(xlDialogSaveAs).Show
```

**Procedures**

1. Launch or switch to the *VB Editor*.
2. Identify in the *Project Explorer* pane, the workbook (VBA project) that you want to add code to.
3. Open the module sheet containing the code or, insert a new module.
4. Position the cursor in the procedure where you want to show the built-in dialog box.
5. Type *Application*.
6. Type a full stop.
7. Type *Dialogs*.
8. Type an opening bracket (.
9. Type or select from the list the dialog box required.
10. Type a closing bracket ).
11. Press *Enter*.

**User-Defined Forms**

**Discussion**

As with built-in dialog boxes, *User-Defined Forms* (or just *User Forms*) can be created to allow an applications to interact with the user. UserForms are, in essence, dialog boxes that you design yourself for a special purpose that is not met by the built-in ones.
Creating a functioning UserForm can be broken down into five steps:

1. Inserting a blank UserForm into a workbook
2. Adding controls to the form.
3. Giving the controls necessary properties.
4. Adding VBA code “behind” the form in order to make it respond to the user input.
5. Adding code to display the form.

Controls can also be added directly on to a worksheet or a chart. This topic is covered in the Microsoft Excel Level 4 Course.

**INSERTING A USERFORM INTO A WORKBOOK**

Before a UserForm can be created, a blank form has to be inserted into the workbook that the form applies to. It is a good idea to have clearly in your mind what you want the UserForm to achieve before making a start as this will determine the controls that need to be placed on it and how they are setup and programmed.

**Procedures**

1. Launch the **VB Editor**.
2. Right-click the workbook that you want to insert the UserForm into.
3. Point to **Insert**.
4. Select **UserForm**.

**ADDING CONTROLS TO A FORM**

Controls are the objects that you can add to a user form so that the user can “talk” with it (hence, dialog box). These appear on the Toolbox toolbar when a UserForm is inserted into a workbook in the VB Editor, and are quite literally drawn onto the form.

![The UserForm Toolbox toolbar](image)
These are as follows:

<table>
<thead>
<tr>
<th>Name of Control</th>
<th>Button Image</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select Objects</td>
<td><img src="select_objects.png" alt="Select Objects" /></td>
<td>Selects objects on a form</td>
</tr>
<tr>
<td>Label</td>
<td><img src="label.png" alt="Label" /></td>
<td>Static text that is displayed to inform the user</td>
</tr>
<tr>
<td>Text Box</td>
<td><img src="text_box.png" alt="Text Box" /></td>
<td>Mostly used when a typed-in input is requested from the user</td>
</tr>
<tr>
<td>Combo Box</td>
<td><img src="combo_box.png" alt="Combo Box" /></td>
<td>Displays a dropdown list of values to choose from</td>
</tr>
<tr>
<td>List Box</td>
<td><img src="list_box.png" alt="List Box" /></td>
<td>Displays a fixed list of values to choose from (possibly with scroll bar)</td>
</tr>
<tr>
<td>Check Box</td>
<td><img src="check_box.png" alt="Check Box" /></td>
<td>A box allowing the user to set a yes/no, true/false value</td>
</tr>
<tr>
<td>Option Button</td>
<td><img src="option_button.png" alt="Option Button" /></td>
<td>A button (often referred to as radio button) allowing the user to set a yes/no, true/false value. Usually used in groups (see below)</td>
</tr>
<tr>
<td>Toggle Button</td>
<td><img src="toggle_button.png" alt="Toggle Button" /></td>
<td>A button allowing the user to set a yes/no, true/false value. The button appears pressed in when “on” and out when “off.”</td>
</tr>
<tr>
<td>Frame</td>
<td><img src="frame.png" alt="Frame" /></td>
<td>Used to create group of option buttons. This allows only one option button at a time to be active.</td>
</tr>
<tr>
<td>Command Button</td>
<td><img src="command_button.png" alt="Command Button" /></td>
<td>A button for running a command. Most commonly used to <strong>OK</strong> or <strong>Cancel</strong> the UserForm.</td>
</tr>
<tr>
<td>Tab Strip</td>
<td><img src="tab_strip.png" alt="Tab Strip" /></td>
<td>A TabStrip is used to view different sets of information for related controls</td>
</tr>
<tr>
<td>MultiPage</td>
<td><img src="multi_page.png" alt="MultiPage" /></td>
<td>Used generally for large amounts of data that needs to be shown in separate tabs</td>
</tr>
<tr>
<td>Spin Button</td>
<td><img src="spin_button.png" alt="Spin Button" /></td>
<td>Allows the user to select a numerical value by clicking up and down buttons. The Spin Button needs a text box to display (return) its values.</td>
</tr>
<tr>
<td>Scroll Bar</td>
<td><img src="scroll_bar.png" alt="Scroll Bar" /></td>
<td>Similar to a Spin Button but in the form of a bar with a sliding “lever.”</td>
</tr>
<tr>
<td>Image</td>
<td><img src="image.png" alt="Image" /></td>
<td>Displays a picture. File formats vary with Excel versions but most common are: <strong>.bmp</strong>, <strong>.jpg</strong>, <strong>.wmf</strong>, <strong>.gif</strong>, <strong>.ico</strong>.</td>
</tr>
<tr>
<td>RefEdit</td>
<td><img src="refedit.png" alt="RefEdit" /></td>
<td>Similar to a text box control but contains a button at the right that collapses the form thus allowing easier selection of cells behind it.</td>
</tr>
</tbody>
</table>
It is advisable to have clear in your mind what controls the form should contain before you start. Design the form on paper before you start.

Procedures

1. Click in the Toolbox, the button for the control that you want to draw on the form.
2. Click the mouse pointer (black cross) on the form where you want the top left corner of the control to be placed.
3. Point to the grey border on the edge of the control (mouse pointer changes to a crosshair).
4. Click and drag to move the control, if necessary.
5. Point to one of the white square on the corners or edges of the control (mouse pointer changes to a double headed arrow).
6. Click and drag to resize the control, if necessary.
7. In the case of:
   - Labels
   - Command buttons
   - Check boxes
   - Options buttons
   Click on any visible text (Caption property) and replace it with a description of what the control stores or does, eg. a command button might have the text **OK** or **Cancel** on it.

8. Continue drawing, moving and resizing controls as necessary.

---

**In cases where text cannot be written directly on the control (eg. Frames, MultiPage and the form title), use the Caption property for the control.**

**Double-clicking a button on the Toolbox toolbar allows multiple controls of that type to be drawn on the form. Clicking the button again (or pressing ESC) cancels this operation.**
FORM CONTROLS DESIGN TOOLS AND TECHNIQUES

**Grouping**

Grouping controls, temporarily joins them together so that they can be moved, resized, aligned and formatted simultaneously.

Controls can also be grouped together permanently.

**Procedures**

1. Click the first control that you want to include in the group.
2. Hold down the SHIFT key.
3. Click the next control that you want to include in the group.
4. Continue holding down the SHIFT key and clicking control to include in the group.
5. Release the SHIFT key when all the controls have been selected.
6. Move or resize any of the grouped control.
7. Click the **Format** menu.
8. Point to the appropriate command to format and layout the grouped controls.

   ![Format Menu](image)

   - Align
   - Make Same Size
   - Size to Fit
   - Size to Grid
   - Horizontal Spacing
   - Vertical Spacing
   - Center in Form
   - Arrange Buttons
   - Group
   - Ungroup
   - Order

9. Click away from the grouped controls to cancel the selection.

![Grouped Labels and Text Boxes on a UserForm](image)
To permanently group controls together, select them as described above and then click the **Format** menu, **Group** command in the VB Editor.

To gain quicker and easier access to control layout and formatting commands, the **UserForm** toolbar can be displayed in the VB Editor.

Click the **View** menu, point to **Toolbars** and select **UserForm** from the side menu.

---

**CONTROL PROPERTIES**

Each form control has a list of properties that can be displayed and (if necessary) changed in the **Properties** pane.

Many properties can be modified by directly formatting the control on the form; others are set from the properties window. Properties can also be set or modified at run-time, i.e. when the form is displayed.

A comprehensive list of all properties for all form control would be too long and unnecessary for this booklet, but help can be sought by clicking onto a property in the properties pane and pressing **F1**.
If the Properties pane is not visible, click View – Properties, press F4 or click the Properties button on the VB Editor toolbar.

For the purposes of a UserForm, the property that would usually need to be set for all controls on the form is the Name property. Giving the controls a clear, consistent and descriptive name makes them easier to identify when it comes to programming the form.

Adding a Name Property to a Text Box Control

Control Naming Convention

While there seem to be no hard and fast rules to naming form controls, the use of prefixes to describe the type of control is a recommended approach. For example, naming a control `txtDate` or `chkMkItalic` makes it clear what type of control it is (text box / check box), and what it holds or does (date / makes italic).

The following list can be used as a guide to prefixes for the more common form controls:

<table>
<thead>
<tr>
<th>Control Type</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label</td>
<td>lbl</td>
</tr>
<tr>
<td>Text Box</td>
<td>txt</td>
</tr>
<tr>
<td>Combo Box</td>
<td>cbo</td>
</tr>
<tr>
<td>List Box</td>
<td>lst</td>
</tr>
<tr>
<td>Check Box</td>
<td>chk</td>
</tr>
<tr>
<td>Option Button</td>
<td>opt</td>
</tr>
<tr>
<td>Command Button</td>
<td>cmd or btn</td>
</tr>
<tr>
<td>Spin Button</td>
<td>spin</td>
</tr>
<tr>
<td>Scroll Bar</td>
<td>scr</td>
</tr>
<tr>
<td>Image</td>
<td>img</td>
</tr>
</tbody>
</table>

The form itself should also be named. The prefix to use is `frm`, eg. `frmDataEntry`.

Procedures (naming a control)

1. Launch the VB Editor.
2. Identify the workbook containing the UserForm that you want to modify control properties for.

3. Display the form.

4. Select a control that you want to name.

5. Click the **Name** box in the **Properties** pane.

6. Type a name for the control, bearing in mind the naming convention described above.

7. Press **Enter**.

   To add a name property for a form, click the grey form background or its blue title bar.

---

**PROGRAMMING A USERFORM**

**Discussion**

UserForms and controls have a predefined set of events. For example a command button has a “**click**” event, a procedure that runs when the user clicks the command button. UserForms have an “**initialise**” event that runs when the form is loaded.

In order for a UserForm to work in the expected way (ie. as a dialog box), it has to have at least two event procedure – one for when the OK button is pressed and a second for when the Cancel button is pressed.

To write a control or form event procedure, open a module by double clicking the form or control. A default event will appear on the module sheet but this can be changed by selecting a different one from the procedure drop-down list at the top right of the window.

   A good analogy to the module sheet that appears when you double click a form or control is to imagine the back of the form where the instructions on how to use it are written.

Event procedures include the name of the control. For example, the name of the click event procedure for a command button named **cmdOK** is **Private Sub cmdOK_Click**. “Private” because the sub procedure should not show in the *Macro* list in Excel, it does not need to because it is purely for use by the OK button on the form.

   Beware of renaming a form or control after the event procedures have been written. The name of the procedure will not be automatically renamed and hence, the event procedure will not run as expected.
It is good practice to name controls as described in the previous topic before writing event code.

Procedures

1. Double click the **Cancel** button on the form design.
2. Add code as necessary (see examples below).
3. Double-click the form in the **Project Explorer** pane.
4. Double-click the **OK** button on the form design.
5. Add code as necessary (see examples below).
6. Double click the form in the **Project Explorer** pane.

All the possible ways of programming a form and its controls' events are too numerous and varied to mention in any document. The examples given below, therefore, apply to the more commonly used controls and are given as a guide only. Additional examples and help can be obtained from the VB Editor help, and on various websites including the Microsoft Developer Network at:


**The Cancel Command Button**

Only one thing would normally happen when the cancel button is clicked – the form is removed from the screen and anything typed or set in its controls discarded. The code for this would be (assuming the form was named frmDataEntry):

```
Unload frmDataEntry
```

**The OK Command Button**

Many things will happen when the OK button is clicked! For example:

a) Anything typed into text boxes will have to be acted on.

b) Items selected in combo boxes and list boxes will need to be interpreted and made to carry out commands.

c) Spin button and scroll bars will have to have their values converted into meaningful actions.

d) Check boxes, toggle buttons, and options button on the form will need to be tested to ascertain which condition(s) apply.

e) The form itself must be removed from the screen after all the above tasks have been completed.

The following examples all apply to the “click event” of the OK button and can be written in any order. When referring to the control, its name as defined in the properties pane must always be preceded by the name of the form that it is on, eg:

```
frmDataEntry.txtName.Value
```
The word Me can be used in place of the full form name. Me meaning “me, the form that I am on!”

So instead of...

```
frmDataEntry.txtName.Value
```

you can use...

```
Me.frmDataEntry.txtName.Value
```

**Text Boxes**

Typically, when the OK button is clicked, text entered into a text box will need to be transferred to a cell on a sheet, eg:

```
ActiveSheet.Range("A2").Value = frmDataEntry.txtName.Value
```

“Value” can be omitted because VBA always defaults to the value property for a range object if the method or property is omitted. In addition, if the form must write the text into cell A1 of the sheet that the form is being run from, there is no need to include ActiveSheet. Hence, the code can be abbreviated to:

```
Range("A2") = Me.txtName
```

**Combo Boxes and List Boxes**

These can be used in a similar way to a text box, to enter the selected text into a cell on a sheet. The code, therefore, would be very similar, eg:

```
Range("A3") = Me.cmbDepartment
```

or

```
Range("A4") = Me.lstLocation
```

Under other circumstances, it may be that the combo box or list box is designed to give the user options to choose from. In the following example, a list box has been used to give the user a choice of files to open. The code, therefore, would be:

```
Workbooks.Open Filename:= Me.lstFileNames
```

The above code assumes that the file to open is in the active, default folder. It may be safer, therefore, to concatenate the correct path to the value returned by the control, eg:

```
Workbooks.Open Filename:= s:\Excel\Stats\ & Me.lstFileNames
```

or

```
ChDir "s:\Excel\Stats\"
Workbooks.Open Filename:= Me.lstFileNames
```
In the following example, a combo box has been placed on the form to give the user a selection of fonts to use. Assuming that the range to format is A1 to E1, the code would be as follows:

```
Range("A1:E1").Font.Name = Me.cmbFontName
```

**Option Buttons**

Although not necessary if only one set of option buttons are used on a form, it is good practice to draw them inside a frame. Only one option button can be selected at a time so when one button is “pressed in,” another is “pushed out” (hence, radio buttons).

```
If Me.optMale = True Then
    Range("B4").Value = "M"
Else
    Range("B4").Value = "F"
End If
```

**Check Boxes**

Because a check box has two conditions – true or false, an If statement has to be used to evaluate them. In the following example, a check box has been used for the user to state whether to print the workbook or not.

```
If Me.chkPrint.Value = True Then
    ActiveDocument.PrintOut
End If
```

**Spin Buttons and Scroll Bars**

A spin button or scroll bar control does not have a built in way of viewing its value. Under usual circumstances, a text box has be placed beside the control to show its current value.
The picture below, a spin button and a text box have been placed on a UserForm side by side.

The code for making the value of the spin button visible in the text box would be added to the spin button’s On_Change event. As the value in the spin button changes, it is transferred to and made visible in the text box.

Procedures

1. Double-click the spin button on the form design.
2. Add the following code to the button’s On_Change event procedure.
   a. Private Sub spnCopies_Change()
      i. Me.<text box name> = Me.<spin box name>
   b. End Sub
3. Double-click the form in the Project Explorer pane.

A spin button’s values (ie. how high and low they can go) can be controlled by using its Max and Min properties.

The above procedure is purely to make the value of the spin button visible to the user while using the form. Additional code has to be added to the OK button’s on click event to make use of the spin button’s value.

In the following example, a spin button has been created on a form to prompt the user for how many copies to print. The code uses the spin button’s value to print the requested number of copies.

    Activesheet.PrintOut Copies:=Me.spnCopies

**FORM EVENTS**

The topic above deals with events associated with form controls. The form itself also has “events” that can be assigned a procedure. As with control events, the number and variety of examples is too great to cover in this booklet. A couple of commonly used examples, however, are given below to demonstrate use of the Initialize event. The initialize event occurs when the form loads (just before it becomes visible on the screen) and is often used to set default values for its controls and for populating combo and list boxes.

Example A populates a combo box with four cities and Example B applies default values to a text box and check box.
Example A

Private Sub UserForm_Initialize()

    With Me.cmbLocations
        .AddItem “Belfast”
        .AddItem “Cardiff”
        .AddItem “Edinburgh”
        .AddItem “London”
    End With

End Sub

A simpler way of populating a combo or list box is to type the list in a column on a worksheet. Create a named range from the list and use the range name in the **Row Source** property of the control.

To limit a combo box so that users can only select from the list, set the **Style** property to **frmStyleDropDownList**.

Example B

Private Sub UserForm_Initialize()

    Me.txtLocation.Value = “London”
    Me.chkPrint.Value = True

End Sub

DISPLAYING A **USERFORM**

To display a user form, use the Show method for the form in question. This would be written on a normal module sheet in the same workbook that the form has been created in. The following example displays the UserForm named frmDataEntry:

Sub DisplayForm()
    frmDataEntry.Show
End Sub
Once the procedure has been written, a method of running it from Excel needs to be chosen. This would be the same as for any other macro, i.e. button, menu or keystroke. The code could also be included as part of a larger procedure, or called into one as a separate sub procedure.

**Procedures**

1. Launch the VB Editor.
2. Identify the workbook containing the form that you want to show.
3. Insert a module sheet in the workbook.
4. Type the following code:
   ```vba
   Sub ShowForm
       <formname>.Show
   End Sub
   ```
5. Run the userform
**EXERCISE**

**CREATING A USERFORM**

**Task - To create a UserForm to prompt for Text, ComboBox and CheckBox information.**

1. Open a Blank Workbook and save it as User Form Practice.
2. Insert a new UserForm into the workbook and create controls on it as shown below:

![UserForm Diagram]

3. Set properties to the objects as follows:

<table>
<thead>
<tr>
<th>Control</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>CommandButton 1</td>
<td>Name: btnOK</td>
</tr>
<tr>
<td></td>
<td>Caption: OK</td>
</tr>
<tr>
<td>CommandButton 2</td>
<td>Name: btnCancel</td>
</tr>
<tr>
<td></td>
<td>Caption: Cancel</td>
</tr>
<tr>
<td>Label 1</td>
<td>Caption: Enter your name</td>
</tr>
<tr>
<td>Text Box</td>
<td>Name: txtInput</td>
</tr>
<tr>
<td>Label 2</td>
<td>Caption: Select a colour for your name</td>
</tr>
<tr>
<td>ComboBox</td>
<td>Name: comColour</td>
</tr>
<tr>
<td>CheckBox 1</td>
<td>Name: chkBold</td>
</tr>
<tr>
<td></td>
<td>Caption: Bold</td>
</tr>
<tr>
<td>CheckBox 2</td>
<td>Name: chkItalic</td>
</tr>
<tr>
<td></td>
<td>Caption: Italic</td>
</tr>
<tr>
<td>UserForm</td>
<td>Name: frmDataEntry</td>
</tr>
<tr>
<td></td>
<td>Caption: Data Entry</td>
</tr>
</tbody>
</table>
4. Assign an "on-click event" (what will happen when the control is clicked) to the Cancel buttons by double clicking it and typing the following code:

   **Unload frmDataEntry**

5. Double-click the form in the Project Explorer to return to its design window.

6. Assign an “initialize event” to the form (what happens when the form starts up) by double clicking the form background and typing the following code. This is necessary to load (“populate”) the combo box:

   ```vba
   With frmDataEntry.comColour
     .AddItem "Red"
     .AddItem "Blue"
     .AddItem "Green"
   End With
   ```

7. Return to the form design window.

8. Double-click the OK button and add the following code. This is necessary to “implement” the userform:

   ```vba
   Range("A1").Value = frmDataEntry.txtInput.Value
   Select Case frmDataEntry.comColours
     Case Is = "Red"
     Range("A1").Font.Color = vbRed
     Case Is = "Blue"
     Range("A1").Font.Color = vbBlue
     Case Is = "Green"
     Range("A1").Font.Color = vbGreen
   End Select

   If frmDataEntry.chkBold = True Then
     Range("A1").Font.Bold = True
   Else
     Range("A1").Font.Bold = False
   End If

   If frmDataEntry.chkItalic = True Then
     Range("A1").Font.Italic = True
   Else
     Range("A1").Font.Italic = False
   End If

   Unload frmDataEntry
   ```

9. Finally, write a short sub procedure on a new module sheet to show the form:

   ```vba
   Sub EnterData()
     frmDataEntry.Show
   End Sub
   ```
10. Create a custom button on your toolbar to run the `EnterData` macro and check correct data entry in cell A1. Correct any code, if necessary.

11. Save and close the file.
LESSON 3 –
ACTIVEX WORKSHEET CONTROLS

In this lesson, you will learn how to:

- Create ActiveX worksheet controls
- Modify the properties of worksheet controls
- Write VBA code behind ActiveX worksheet controls
OVERVIEW AND DEFINITION

Discussion

In general, worksheet controls are buttons, check boxes, drop-down lists, spinners, lists and scroll bars (to name a few) that you can create on a worksheet to enable a user to enter their data. Not only is it quicker to select data from a control, but you can restrict what the user can select or type in it, resulting in better quality and more accurate data entry. Worksheet controls are normally used to build forms on worksheets.

Microsoft Excel offers two types of controls:

- Form controls; and
- ActiveX controls.

In Excel 2000 – 2003 these controls are available from toolbars, viz. the Forms toolbar (Form controls) and the Control Toolbox toolbar (ActiveX controls).
Excel Level 7: VBA Advanced

Lesson 3 - Worksheet Controls

Excel 2007 Controls group on Developer Ribbon

Form Controls
Form controls are only available to use on an Excel worksheet. Because these controls don’t require any VBA code behind them to work (although a limited amount of code can be added if required in Excel 2000 - 2003), they are fairly easy to create using standard Excel methods and commands. This, however, means that they are not especially flexible or powerful.

The most common form control is the command button; drawn on a worksheet and set up so that when it is clicked, a macro runs.

![Worksheet macro buttons created with Form controls](image)

Using form controls is covered on the Microsoft Excel Level 4 training course.

ACTIVEX CONTROLS

Discussion
ActiveX controls are available for use directly on an Excel worksheet OR they can be used in the VB Editor to create user forms (see Lesson 2 - Built-in Dialog Boxes and custom UserForms on page 19).

ActiveX controls are more useful than the form controls because they can be programmed more extensively by making them respond to “events.”

An event occurs when a user or the computer itself makes something “happens;” ie. they initiate an action. The commonest way of creating an event is by clicking a button on the screen or pressing a key on the keyboard. The computer itself can initiate an event by using an internal timer or clock, eg. pop up a meeting reminder.

With ActiveX controls, as well as a simple click, other less common events (but ones that can be “trapped” and programmed) are: double click; change (eg. a new item is selected from a combo box or list box); user leaving the control and shifting focus to another control; or back to the Excel interface, to name but a few.
ActiveX controls also have many “Properties” that can be used to change the characteristics of a control (eg. size, font, colour). The properties of a control depend on its type although some - such as Name - are common to all.

CONTROL PROPERTIES

Discussion

Each control has its own distinct set of characteristic or “properties.” Properties define how the control looks and behaves.

Properties vary for different controls vary. For example, a Spin Button control has a “Max” and a “Min” property that restricts the highest and the lowest value that it can spin to. This property is irrelevant to, say, a Combo Box control in the same way as a Combo Box’s “ListFillRange” property (the data to use for populating the combo box) would be irrelevant to a Spin Button. As a result, properties that are not applicable to a control are omitted from the properties list.

Some properties, however, are common to many controls. The list below gives the more commonly used ones.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoSize</td>
<td>If True, the control resizes itself automatically based on the length of text entered into it.</td>
</tr>
<tr>
<td>BackColor</td>
<td>The background colour of a control.</td>
</tr>
<tr>
<td>BackStyle</td>
<td>The style of the background (either transparent or opaque).</td>
</tr>
</tbody>
</table>
**Caption**
The text that appears on the control.

**LinkedCell**
A worksheet cell that contains the current value of a control.

**ListFillRange**
A worksheet range that contains items displayed in a ListBox or ComboBox control.

**Value**
The control's value.

**Left and Top**
Values that determine the control’s position.

**Width and Height**
Values that determine the control's width and height.

**Visible**
If False, the control is hidden.

**Name**
The name of the control. By default, a control’s name is based on the control type followed by a number. You can change the name to any valid name. However, each control’s name must be unique on the worksheet.

**Picture**
Enables you to specify a graphic image to display.

**Procedures**

1. Right click the control that you want to change the properties of.
2. Select **Properties**.
3. Edit or add property value as necessary.

**Adding ActiveX Controls to a Worksheet**

**Discussion**

**Check Box Control**

A Check Box consists of a caption and a small box into which a check mark can be placed by clicking it with the mouse. A Check Box is used for setting Yes/No or True/False values.

![Check Box control](image)

**Procedures (Excel 2000 – 2003)**

1. Show the **Control Toolbox** toolbar.
2. Click the **Check Box** button on the toolbar. The mouse pointer changes to a small crosshair.
3. Drag the mouse pointer over the area on the worksheet where you want to create the check box.
4. Point at the control and right click.
5. Select Properties.
6. Replace the text in the Caption text box with what you want to display next to the check box.

7. Close the Properties window.

Procedures (Excel 2007)

1. If necessary, show the Developer tab on the Ribbon as follows:
   a) Click the Office button.
   b) Click Excel Options.
   c) Select Popular in the left-hand pane.
   d) Click the Show Developer tab in the Ribbon check box.
   e) Click OK.
2. In the Developer tab, click Insert in the Controls group.
3. Click the Check Box button under ActiveX Controls.
4. Drag the mouse pointer over the area on the worksheet where you want to create the check box.
5. Point at the control and right click.
7. Replace the text in the **Caption** text box with what you want to display next to the check box.

8. Close the **Properties** window.

---

To align a control to the worksheet gridlines, press the [Alt] key as you drag.

**MOVING A CONTROL**

**Discussion**

By pointing at an ActiveX control, you can re-position elsewhere on the worksheet. When you re-position a control from one location to another, its size stays the same size.

To copy a control, hold down the [Ctrl] key as you drag.

**Procedures**

1. Click on the control that you want to reposition.
2. Drag the control to another part of the worksheet.
3. Release the mouse button
SIZING A CONTROL

Discussion

By pointing at a sizing handle and clicking and dragging, you to change the size or proportions of a control. By dragging out from the centre of the control, the object becomes larger. When you drag towards to centre of a control, the control becomes smaller. When you resize a control, you change its dimensions, not the size of the text.

To resize a control proportionally, press the [Shift] key as you drag. To resize a control proportionally from its centre, press the [Ctrl] key as you drag.

Procedures

1. Click the control you want to resize.
2. Drag the sizing handles as required to make the control bigger or smaller.
3. Release the mouse button.

DELETING A CONTROL

Procedures

1. Select the control.
2. Press the [Delete] key.

RETURNING A CONTROL’S VALUE

Discussion

When an ActiveX control is added to a worksheet, Excel goes automatically into "Design Mode." This allows you to work on the control (size and position it, change its properties or add VBA code behind it) without the control attempting to carry out its normal actions.

For the control to work fully, it is necessary to de-activate Design Mode. This is achieved by clicking the Design Mode button on the Control Toolbox button (Excel 2000 – 2003) or clicking the Design Mode button in the Controls group of the Developer tab (Excel 2007).

Control Toolbox show Design Mode button (Excel 2000 – 2003)
**Design Mode button in Developer tab (Excel 2007)**

**Using a linked cell**

At the simplest level, an ActiveX control can be made to return its selected value into a cell anywhere in the workbook that it has been created in. This is simply achieved by giving the control a **LinkedCell** property.

**Procedure**

1. Hold the mouse pointer over the control and right click.
2. Select **Properties**.
3. Type into the **LinkedCell** control the cell reference (eg. A5, B16, D32) that you want the selected value to be returned to.

   If the cell is on a different sheet to the control, include the sheet name (as shown in the tab at the bottom) followed by an exclamation mark (!) before the cell reference, eg.

   **Regions!A10**

   …will return the control’s value to cell A10 on the sheet named Regions of the workbook.
4. Close the **Properties** window.
5. Click the **Design Mode** button on the **Control Toolbox** toolbar (Excel 2000-2003) OR click the **Design Mode** button in the **Controls** group of the **Developer** tab (Excel 2007).
6. Test/ use the control.

**Using VBA code**

It is rare for an ActiveX control to be used without some VBA code that is run as one of its events take place.

This is achieved by double clicking the control (while in design mode). This takes you straight to the module for the sheet that the control has been created on. Think of this module as being the back of the worksheet where instructions are written for how certain things on the sheet should work and behave.

The procedure name will be preceded by the word **“Private.”** This is essentially to hide it from the **Macros** window so that it cannot be seen or run from Excel. This procedure is, after all, only relevant to the control for which it is being written and hence, of no use as a stand-alone macro that can be run in Excel normally.
The second thing to note is the rest of the procedure name. This name will depend on the control that has been double clicked and describes the event that triggers the procedure. The default event usually given to most controls is “Click” or “Change”.

In the example below, a check box has been double clicked.

![Image of Visual Basic Editor showing procedure name](image)

Note the procedure name: `Private Sub CheckBox1_Click()`. The `CheckBox1_Click` part of the name indicates that any code written into this procedure will be triggered (run) when the check box is clicked (i.e., turned on or off).

By clicking the drop down list in the top right of the code window, other events that can occur to the check box are listed.

![Image of Visual Basic Editor showing drop down list](image)

By selecting a different event from the list, the check box can be made to do something under different circumstances. In the example below, the event has been changed to Change.
The Private Sub CheckBox1_Click() event procedure can therefore be ignored or deleted.

Procedures

1. Double click the control.
2. Select the appropriate event for which you want to write a procedure.
3. Enter the relevant code.
4. Return to Excel and test/use the control.

The table below lists the more commonly used events and the controls that they apply to.

<table>
<thead>
<tr>
<th>Event</th>
<th>Control</th>
<th>Occurs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change</td>
<td>Check Box, List Box, Combo Box, Spin Button, Option Button, Text Box, Scroll Bar</td>
<td>Occurs when the value property of a control changes.</td>
</tr>
<tr>
<td>Click</td>
<td>Check Box, List Box,</td>
<td>When the user points to a</td>
</tr>
<tr>
<td>Event</td>
<td>Control</td>
<td>Occurs</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DblClick</td>
<td>Check Box, List Box, Combo Box, Option Button, Text Box, Command Button</td>
<td>When the user points to an control and then dics a mouse button twice.</td>
</tr>
<tr>
<td>DropButtonClick</td>
<td>Combo Box</td>
<td>When the user clicks the drop-down button on the control.</td>
</tr>
<tr>
<td>SpinDown</td>
<td>Spin Button</td>
<td>When the user clicks the lower or left spin-button arrow.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Occurs before the Change event.</td>
</tr>
<tr>
<td>SpinUp</td>
<td>Spin Button</td>
<td>When the user clicks the upper or right spin-button arrow.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Occurs before the Change event.</td>
</tr>
</tbody>
</table>

**Creating a List Box**

**Discussion**

List boxes are used to enable users to select from a list of suitable entries be used in an INDEX function to show the appropriate data. Advantages of using a List Box are:

- Restricting users to authorised cell entries only
- Speedier data entry as no typing is involved.

![A List Box](image)

**Procedures**

**Excel 2000 - 2003**

1. Show the Control Toolbox toolbar.

**Excel 2007**

1. Show the Developer tab if necessary (see item 1 in the procedure on page 42). Click the Insert button in the Controls group.
2. Click the **List Box** button on the toolbar. The mouse pointer changes to a small crosshair.

3. Drag the mouse pointer over the area on the worksheet where you want to create the List Box.

4. Release the mouse button.

5. Create a named range on a worksheet, listing the items that you want to appear in the list box. The worksheet can be anywhere in the workbook that you are creating the control in. The list must be laid out vertically (i.e., down a column, NOT across columns).

<table>
<thead>
<tr>
<th></th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>North</td>
</tr>
<tr>
<td>2</td>
<td>South</td>
</tr>
<tr>
<td>3</td>
<td>East</td>
</tr>
<tr>
<td>4</td>
<td>West</td>
</tr>
</tbody>
</table>

6. Hold the mouse pointer over the spin button and right click.

7. Select **Properties**.

8. Type into the `ListFillRange` property the named range created in 5 above.

9. Press **Enter**.

10. Close the **Properties** window.

---

**CREATING OPTION BUTTONS**

**Discussion**

Option buttons represent mutually exclusive choices, so only one can be selected at a time.

**Procedures**

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Show the <strong>Control Toolbox</strong> toolbar.</td>
<td>1. Show the <strong>Developer</strong> tab if necessary (see item 1 in the procedure on page 42). Click the <strong>Insert</strong> button in the <strong>Controls</strong> group.</td>
</tr>
<tr>
<td>2. Click the <strong>Option Button</strong> button on the toolbar. The mouse pointer changes to a small crosshair.</td>
<td>2. Click the <strong>Option Button</strong> button under <strong>ActiveX</strong> controls. The mouse pointer changes to a small crosshair.</td>
</tr>
<tr>
<td>3. Drag the mouse pointer over the area on the worksheet where you want to create the option button.</td>
<td></td>
</tr>
</tbody>
</table>
4. Release the mouse button.

5. Point at the control and right click.


7. Replace the text in the Caption text box with what you want to display next to the option button box.

8. Repeat steps 2 to 8 for each option button that you need.


A group of option buttons

**CREATING A SPIN BUTTON**

**Discussion**

These are used to enable users to enter numeric values into a cell without having to type the data. Spinners are usually formatted to restrict values between a range and are set to a suitable incremental rate in order to select relevant values more quickly.

![Spin Button]

**Procedures**

<table>
<thead>
<tr>
<th>Excel 2000 - 20003</th>
<th>Excel 2007</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Show the Control Toolbox toolbar.</td>
<td>1. Show the Developer tab if necessary (see item 1 in the procedure on page)</td>
</tr>
</tbody>
</table>
Click the Insert button in the Controls group.

2. Click the Spin Button button on the toolbar. The mouse pointer changes to a small crosshair.

2. Click the Spin Button button under ActiveX controls. The mouse pointer changes to a small crosshair.

3. Drag the mouse pointer over the area on the worksheet where you want to create the spin button.

4. Hold the mouse pointer over the spin button and right click.

5. Select Properties.

6. If necessary, enter a Min and/ or a Max for the biggest and the smallest value(s) that you want the spin button to show.

7. If necessary, enter a SmallChange property for the interval up or down that you want the spin button to use every time it is clicked.

8. Close the Properties window.

**BEWARE -** Do not select the Scroll Bar button by mistake. It is similar in appearance to a Spin Button but works quite differently.

### CREATING A COMBO BOX

**Discussion**

Combo boxes (often called Drop-down Boxes) are similar to List Boxes in the sense that they enable users to select from a list of suitable entries. The difference being that the Combo Box shows only the currently selected item, whereas the List Box shows several items simultaneously.

**Procedures**

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Show the Control Toolbox toolbar.</td>
<td>1. Show the Developer tab if necessary (see item 1 in the procedure on page 42). Click the Insert button in the Controls group.</td>
</tr>
<tr>
<td>2. Click the Combo Box button on the toolbar. The mouse pointer changes to a small crosshair.</td>
<td>2. Click the Combo Box button under ActiveX controls. The mouse pointer changes to a small crosshair.</td>
</tr>
<tr>
<td>3. Drag the mouse pointer over the area on the worksheet where you want</td>
<td></td>
</tr>
</tbody>
</table>
4. Release the mouse button.

5. Create a named range on a worksheet, listing the items that you want to appear in the combo box. The worksheet can be anywhere in the workbook that you are creating the control in. The list must be laid out vertically (i.e. down a column NOT across columns).

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

6. Hold the mouse pointer over the control and right click.

7. Select Properties.

8. Type into the ListFillRange property the named range created in 5 above.


10. Close the Properties window.

Example of a Combo Box
EXERCISE

AUTOMATING A WORKSHEET WITH ACTIVEX CONTROLS

Task - Execute data entry by using a drop-down list, spinner and check box

1. Open the file Controls Practice.
2. Draw a ComboBox control on Sheet1 starting in cell D2 and continuing to cell G3.
3. Draw a Spin Button in cell D5.
4. Draw a Check Box in cell D8.

(use picture below as a guide).

5. Create a named range called StockList using the data in cells A1 to B7 of Sheet2 (this data is imported from the Web and automatically updated every 10 minutes).
6. Set up Properties and code for the three controls as follows:

<table>
<thead>
<tr>
<th>Control</th>
<th>Properties</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComboBox</td>
<td>ListFillRange</td>
<td>StockList</td>
</tr>
<tr>
<td></td>
<td>ColumnCount</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>ColumnWidths</td>
<td>125, 50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Value in column 1 to be entered into cell B2 when selected item changes.</td>
</tr>
<tr>
<td>SpinButton</td>
<td>Max</td>
<td>100000</td>
</tr>
<tr>
<td></td>
<td>Min</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>SmallChange</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Value to be entered into cell B5 when it changes.</td>
</tr>
<tr>
<td>CheckBox</td>
<td>Caption</td>
<td>Commission Yes/No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A zero to be entered in cell B8 when check box is changed to “off”, or 10% when changed to “on”.</td>
</tr>
</tbody>
</table>

7. Turn off Design Mode and test that the controls function as expected. Formulas already exist in cells B7 and B10 for calculation of the results.
8. Write code to the Workbook “Open” event that protects Sheet1 for UserInterfaceOnly.
9. Save and close the workbook.
10. Re-open the workbook and check that the controls and protection functions as expected. You should only be able to enter values using the ActiveX Controls.
11. Save and close the workbook.
LESSON 4 – EVENT PROCEDURES

In this lesson, you will learn how to:

- Identify different types of event
- Correctly locate and name event procedures
- Write VBA event procedures
- Use the OnTime method to schedule running a procedure
OVERVIEW

Discussion

In the preceding two chapters, a start was made at understanding “events” and how to associate VBA code to them.

Essentially, an event occurs when a user makes something “happens” on their computer. The commonest way of creating an event is by clicking a button on the screen or pressing a key on the keyboard, thus triggering a command. When you click a macro button, you are initiating an event (ie. the macro runs). Hence, an “event procedure” is a piece of code that automatically executes when an event is triggered.

Not all events are user initiated; some are triggered by the computer itself through a timer, (eg. an appointment or task reminder popping up in Outlook). Others are set off during, just before or just after certain actions take place in a workbook or on a worksheet, such as a file being opened or saved or a change made to a range. They can also occur when a procedure or macro carries out the actions above.

The events that we will be examining in this lesson are:

a) those associated with a worksheet, eg. when a cell is selected on a sheet or the sheet is recalculated;

b) those associated with the workbook itself, eg. when a new one is created, saved or activated; and

c) those that trigger automatically, eg. using a timer.

TYPES OF EVENT

Discussion

The table below list events associated with worksheets and workbooks and explains when they are triggered.

<table>
<thead>
<tr>
<th>Event</th>
<th>Triggers when...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activate</td>
<td>The worksheet is activated.</td>
</tr>
<tr>
<td>BeforeDoubleClick</td>
<td>The worksheet is double-clicked.</td>
</tr>
<tr>
<td>BeforeRightClick</td>
<td>The worksheet is right-clicked.</td>
</tr>
<tr>
<td>Calculate</td>
<td>The worksheet is calculated (or recalculated).</td>
</tr>
<tr>
<td>Change</td>
<td>Cells on the worksheet are changed by the user.</td>
</tr>
<tr>
<td>Deactivate</td>
<td>The worksheet is deactivated.</td>
</tr>
<tr>
<td>FollowHyperlink</td>
<td>A hyperlink on the worksheet is clicked.</td>
</tr>
<tr>
<td>PivotTableUpdate</td>
<td>A PivotTable on the worksheet has been updated.</td>
</tr>
<tr>
<td>SelectionChange</td>
<td>The selection on the worksheet is changed.</td>
</tr>
<tr>
<td>Event</td>
<td>Triggers when...</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Activate</td>
<td>The workbook is activated.</td>
</tr>
<tr>
<td>AddInInstall</td>
<td>A workbook is installed as an add-in.</td>
</tr>
<tr>
<td>AddInUninstall</td>
<td>A workbook is uninstalled as an add-in.</td>
</tr>
<tr>
<td>AfterXMLExport</td>
<td>Excel saves or exports data from the workbook to an XML data file.</td>
</tr>
<tr>
<td>AfterXMLImport</td>
<td>An existing XML data connection is refreshed or after new XML data is imported into the workbook.</td>
</tr>
<tr>
<td>BeforeClose</td>
<td>The workbook is about to be closed.</td>
</tr>
<tr>
<td>BeforePrint</td>
<td>The workbook (or anything in it) is about to be printed.</td>
</tr>
<tr>
<td>BeforeSave</td>
<td>The workbook is about to be saved.</td>
</tr>
<tr>
<td>BeforeXMLExport</td>
<td>Excel saves or exports data from the workbook to an XML data file.</td>
</tr>
<tr>
<td>BeforeXMLImport</td>
<td>An existing XML data connection is refreshed or before new XML data is imported into the workbook.</td>
</tr>
<tr>
<td>Deactivate</td>
<td>The workbook is deactivated.</td>
</tr>
<tr>
<td>NewSheet</td>
<td>A new sheet is created in the workbook.</td>
</tr>
<tr>
<td>Open</td>
<td>The workbook is opened.</td>
</tr>
<tr>
<td>PivotTableCloseConnection</td>
<td>After a PivotTable report closes the connection to its data source</td>
</tr>
<tr>
<td>PivotTableOpenConnection</td>
<td>After a PivotTable report opens the connection to its data source.</td>
</tr>
<tr>
<td>SheetActivate</td>
<td>Any sheet in the workbook is activated.</td>
</tr>
<tr>
<td>SheetBeforeDoubleClick</td>
<td>Any worksheet in the workbook is double-clicked. This event occurs before the default double-click action.</td>
</tr>
<tr>
<td>SheetBeforeRightClick</td>
<td>Any worksheet in the workbook is right-clicked. This event occurs before the default right-click action.</td>
</tr>
<tr>
<td>SheetCalculate</td>
<td>Any worksheet in the workbook is calculated (or recalculated).</td>
</tr>
<tr>
<td>SheetChange</td>
<td>Any worksheet in the workbook is changed by the user.</td>
</tr>
<tr>
<td>SheetDeactivate</td>
<td>Any sheet in the workbook is deactivated.</td>
</tr>
<tr>
<td>SheetFollowHyperlink</td>
<td>Any hyperlink in the workbook is clicked.</td>
</tr>
<tr>
<td>SheetPivotTableUpdate</td>
<td>After the sheet of a PivotTable report has been updated.</td>
</tr>
<tr>
<td>SheetSelectionChange</td>
<td>The selection on any worksheet in the workbook is changed.</td>
</tr>
<tr>
<td>Sync</td>
<td>The local copy of a worksheet that is part of a Document Workspace is synchronized with the copy on the server.</td>
</tr>
<tr>
<td>WindowActivate</td>
<td>Any window of the workbook is activated.</td>
</tr>
</tbody>
</table>
WindowDeactivate | Any workbook window is deactivated.
WindowResize | Any workbook window is resized.

**WRITING CODE TO EVENTS**

**Discussion**

Event procedures associated with a worksheet or a workbook cannot be written onto a standard module. Those associated with a worksheet must be added to the relevant worksheet module, and those associated with the workbook must be written into the ThisWorkbook module.

The Project Explorer pane in the VB Editor shows the location of these under Microsoft Excel Objects within the relevant VBAProject (Excel file).

![Project Explorer pane showing correct locations for event procedures](image)

As well as writing event procedures in the correct location, they must be given their correct, reserved name. The naming convention for an event procedure associated with a worksheet is as follows:

**Worksheet** followed by an underscore followed by the appropriate **event name** as listed in **Table A** above, eg.

Worksheet_Activate or Worksheet_SelectionChange

The naming convention for an event procedure associated with a workbook is as follows:

**Workbook** followed by an underscore followed by the appropriate **event name** as listed in **Table B** above, eg.

Workbook_Open or Workbook_BeforeSave

Although you can type the name of the procedure yourself, it is more convenient and safer (to prevent typos) to let the VB Editor enter the name for you.

This can be achieved by selecting in the Project Explorer the object that you want to write the procedure into, and then selecting the procedure name from the drop down list at the top of the code pane.
Procedures

1. Launch the VB Editor.
2. Identify in the Project Explorer pane the VBA Project containing the worksheet or workbook object that you want to write an event procedure for.
3. Double click the object.
4. Select **Worksheet** or **Workbook** from the drop down list at the top left of the code pane.
5. Select the appropriate event name from the drop down list at the top right of the code pane.
6. Write code as necessary.

**VB Editor showing event name list for a worksheet**

An example of a simple event procedure assigned to the Workbook_SheetActivate event might be:

```vba
Private Sub Workbook_SheetActivate(ByVal sh As Object)
    MsgBox “A different sheet has been activated”
End Sub
```

Or, a simple event procedure confirming that the correct workbook has been opened could be assigned to the Workbook_Open event as follows:

```vba
Private Sub Workbook_Open()
```

WWP Training Limited
MsgBox "This workbook contains sales data for Quarter 1 2010 only"
End Sub

In older versions of Excel before the VB Editor was introduced (and you have to go back about 13 years!), the same result could be achieved by creating a procedure on a standard module sheet and giving it the reserved name of Auto_Open. This is still supported in Excel and is included here in case you come across this form of coding in older workbooks (or indeed, wish to use it yourself).

You may also encounter Auto_Close, which is equivalent to the Workbook_BeforeClose event.

EVENT PROCEDURE ARGUMENTS

Discussion

Some events come with optional arguments that can be used to enhance the functionality of the event. For example, the Workbook_WindowActivate event comes with the following argument:

Workbook_WindowActivate(ByVal Wn As Window)

wn represents the window that has just been activated, so if you want your event procedure to carry out some action(s) to that window, you refer to it as wn.

In the following example, whenever a window is activated in the current workbook, it is maximised and given a caption (title) of Active Window.

Private Sub Workbook_WindowActivate(ByVal Wn As Window)
    Wn.WindowState = xlMaximized
    Wn.Caption = "Active Window"
End Sub

Conversely, in the example below, whenever a window is deactivated in the current workbook, it is minimised and the caption reset to the file name.

Private Sub Workbook_WindowDeactivate(ByVal Wn As Window)
    Wn.WindowState = xlMinimized
    Wn.Caption = ThisWorkbook.Name
End Sub
In the example below, whenever a new sheet is added to the workbook, it is automatically given a name consisting of a random number between 1 and 10000 and a red tab colour.

```vba
Private Sub Workbook_NewSheet(ByVal Sh As Object)
    Sh.Name = Int(Rnd() * 10000 + 1)
    Sh.Tab.Color = vbRed
End Sub
```

Some events come with a Cancel argument. This is normally used in conjunction with an IF statement to cancel the associated event procedure if unfavourable conditions are encountered. In the example below, the Workbook_BeforeClose event is cancelled if there is no data entered in cell 1 of sheet 1.

```vba
Private Sub Workbook_BeforeClose(Cancel As Boolean)
    If IsEmpty(Sheets("Sheet1").Range("A1")) Then
        MsgBox "Cannot close this file unless data is entered in cell A1 of Sheet 1"
        Cancel = True
    End If
End Sub
```

**COMMON WORKBOOK EXAMPLES**

**Discussion**

**ThisWorkbook**

Probably the two most common event procedures associated with a workbook are Open and BeforeClose.

The Open event is useful for displaying introductory message boxes, opening other files and activating a specific sheet and/ or cells. Eg:

```vba
Private Sub Workbook_Open()
    ' Maximises the Excel window
    Application.WindowState = xlMaximized

    ' Opens another workbook named Data Analysis and maximises it
    Workbooks.Open "Data Analysis"
    ActiveWindow.WindowState = xlMaximized

    ' Activates the sheet named Summary and select cell A1
    Workbooks("Data Analysis").Activate
    Worksheets("Summary").Activate
    Range("A1").Select
```
End Sub

The **BeforeClose** is equally useful for cleaning up and/ or restoring data or settings when closing a file.

Other useful workbook events are those associated with sheets such as, **SheetActivate** and **NewSheet**. The following example ensures that cell A1 is always selected when a different sheet in the current workbook is activated.

**Private Sub Workbook_SheetActivate(ByVal Sh As Object)**

```vba
    On Error Resume Next
    Range(“A1”).Select
End Sub
```

The **On Error Resume Next** statement prevents a runtime error occurring if the sheet being activated does not contain cell A1 (eg. it is a chart sheet).

The following enters and formats a heading in cell A1 of all new sheets added to the workbook.

**Private Sub Workbook_NewSheet(ByVal Sh As Object)**

```vba
    If TypeName(Sh) = “Worksheet” Then
        Range(“A1”).Value = “Expenditure Figures for ” & Date
        Range(“A1”).Font.Bold = True
    End If
End Sub
```

On this occasion, a simple IF statement is used to evaluate the type of sheet being inserted to ensure that a runtime error is not generated if, for example, a chart sheet is inserted.

**Worksheet**

A popular event associated with a sheet is **SelectionChange**. The following example, colours the cell being selected, thus making it easier to identify.

**Private Sub Worksheet_SelectionChange(ByVal Target As Range)**

```vba
    ' Removes any current interior colour from the current sheet
    Cells.Interior.ColorIndex = xlNone

    ' Adds a yellow interior to the selected cell (Target)
    Target.Interior.ColorIndex = 6
End Sub
```
NON OBJECT-RELATED EVENTS (ONTIME)

Discussion

Although the majority of event procedures are associated with an object, it is worth looking at the OnTime method. This acts like an event in as much as it can be used to schedule a procedure to run at a specified time in the future (either at a specific time of day or after a specific amount of time has passed).

Unlike the events covered previously in this lesson, the OnTime method is made use of in a standard procedure on a general module sheet.

The OnTime method is a member of the Application object, hence the syntax for its use is:

```
Application.OnTime
```

It has arguments as follows:

```
Application.OnTime (Earliest Time, Procedure, Latest Time, Schedule)
```

**EarliestTime: Required** - The time when you want this procedure to be run.

**Procedure: Required** - The name of the procedure to be run.

**LatestTime: Optional** - The latest time at which the procedure can be run. For example, if LatestTime is set to EarliestTime + 30 and Microsoft Excel is not in Ready, Copy, Cut, or Find mode at EarliestTime because another procedure is running, Microsoft Excel will wait 30 seconds for the first procedure to complete. If Microsoft Excel is not in Ready mode within 30 seconds, the procedure won’t be run. If this argument is omitted, Microsoft Excel will wait until the procedure can be run.

**Schedule: Optional** - True to schedule a new OnTime procedure. False to clear a previously set procedure. The default value is True.

**EarliestTime** and **LatestTime** can be set using the time expressed as a fraction of a day. For example:

```
Application.OnTime 0.5, "MyProcedure"
```

... would run MyProcedure at 12 noon (half way through the day!).

```
Application.OnTime 0.75, "MyProcedure"
```

... would run MyProcedure at 6pm.

Because it is difficult to calculate times out as fractions of a day, the easiest way of entering the times is by using the TimeValue function. This allows you to enter the time in a more understandable hours, minutes and seconds format. For example, 3pm could be expressed as TimeValue(“15:00:00”). Hence:

```
Application.OnTime TimeValue("15:00:00), "MyProcedure"
```

... would run MyProcedure at 3pm.
The examples above allow you to run a procedure at a specific time in the future. By using the \texttt{NOW} function in conjunction with \texttt{TimeValue}, you can run a procedure after a specific time has elapsed. For example, \texttt{NOW + TimeValue(“00:01:00”)} will calculate a time of one minute from when the procedure is initiated.

Hence:

\texttt{Application.OnTime NOW + TimeValue(“00:01:00”), “MyProcedure”}

... will run MyProcedure one minute from when the statement is executed. So if the statement is executed at 13:45, MyProcedure will run at 13:46.

You also can use the \texttt{OnTime} method to schedule a procedure on a particular day but to do this, you must keep your computer turned on with Excel active and, of course, you must remember to run the procedure that contains the \texttt{OnTime} method before leaving!
EXERCISE

HANDLING EVENTS WITH VBA

Task - Run procedures automatically when certain changes occur in a workbook or on a worksheet

1. Open the file Event Triggers.
2. Create a “Splash Screen” (a form that appears when you open a file and then closes automatically after a few seconds) by firstly creating a user form as shown below.

3. Add code to the appropriate event (Tip: it’s a workbook one) to display the splash screen when the workbook opens.
4. Add code to another event to close the form after 3 seconds (Tip: it’s one of the form events - 3 seconds after what does the form have to close? Look through the list of form events).
5. Save and close the file.
6. Re-open it and check if the Splash Screen works correctly. Debug if necessary.
7. Write a procedure against the appropriate event for Sheet1 to highlight the cell being selected by giving the entire row and the entire column a yellow fill colour.
8. Test and debug.
9. Write a procedure to the appropriate event on Sheet2 in order to prevent the entering of text values or numbers over 100 into the range E1 to E20.
10. Test and debug.
11. Finally, write a procedure against the appropriate event so that cell A1 in Sheet1 is always selected when the workbook closes (and hence, selected when the workbook opens!).
12. Test and debug.
13. Save and close the file.
LESSON 5 –
WORKING WITH CHARTS

In this lesson, you will learn how to:

- Create a chart using VBA
- Move and size a chart
- Format a chart
- Add a series to a chart
- Create a Pivot Table
- Refresh a Pivot Table
- Format and structure a Pivot Table
OVERVIEW

Discussion

While it is true that the macro recorder does a pretty good job of creating and formatting charts, it does not always do it in the most effective and efficient way. This lesson therefore will look at some simple and useful ways of enhancing the code created by the recorder and adding code to carry out additional tasks that the recorder cannot manage.

There are so many chart objects, that a comprehensive explanation of them all would stretch to dozens of pages. The examples here, therefore, gives a few simple VBA techniques to create charts, move and size them and add or remove series and add/ edit their formatting and structure. The code shown is also useful because it can be used to program Microsoft Chart; the applet used by other Office applications to create charts for them (eg. in a PowerPoint slides).

CREATING CHARTS

Discussion

The recorder uses quite a roundabout method to create an embedded chart (one that is located on a worksheet alongside the data). It first creates a default chart on a separate chart sheet, makes changes to it as necessary and then relocates it to the required worksheet. It places it in the default location (centre of work area) and uses the default size (approximately a quarter of the work area).

The recorded code below creates an embedded clustered bar chart using just the first step of the Chart Wizard (ie. clicking **Finish** after selecting the chart type):

```
Sub CreateChart()
    Charts.Add
    ActiveChart.ChartType = xlBarClustered
    ActiveChart.SetSourceData Source:=Sheets("Sheet1").Range("A2:E5")
    ActiveChart.Location Where:=xlLocationAsObject, Name:="Sheet1"
End Sub
```

Although the code does the job, it is rather inflexible in as much as it will always create the chart using the same data (Sheet1 A2:E5) and locate it on the same sheet (Sheet1).

To make the procedure create the chart from ANY data on ANY sheet and always place it on the same sheet as the data, you need to create a couple of variables to:

- a) Store the selected data as a range object; and
- b) Store the active sheet name as a string.

The variables can then be used as arguments for the **SetDataSource** **Source:=** and **Location Name:=** arguments in place of the absolute references created by the recorder. Eg:
Sub CreateChart()

Dim chtData As Range
Dim chtSheet As String

Set chtData = Selection
chtSht = ActiveSheet.Name
Charts.Add
ActiveChart.SetSourceData Source:=chtData
ActiveChart.ChartType = xlBarClustered
ActiveChart.Location Where:=xlLocationAsObject, Name:=chtSht

End Sub

This is a good way of creating a procedure that acts like a template, always giving you a consistent chart in a “house” style.
The procedure also works if chtData is a non-contiguous range.

SIZING AND POSITIONING CHARTS

Discussion

The recorded procedure below repeats creating the embedded chart above, but then moves it to the top left hand corner of the worksheet and resizes it to a width of six standard columns and to a height of 20 rows.

Sub CreatePositionAndSizeChart()

Charts.Add
ActiveChart.ChartType = xlColumnClustered
ActiveChart.SetSourceData Source:=Sheets("Sheet1").Range("A2:E5")
ActiveChart.Location Where:=xlLocationAsObject, Name:="Sheet1"
ActiveSheet.Shapes("Chart 1").IncrementLeft -183#
ActiveSheet.Shapes("Chart 1").IncrementTop -115.8
ActiveSheet.Shapes("Chart 1").ScaleWidth 0.91, msoFalse, msoScaleFromTopLeft
ActiveSheet.Shapes("Chart 1").ScaleHeight 0.85, msoFalse, msoScaleFromTopLeft

End Sub

Not only does this use absolute references for the sheet and the range (although this can be rectified as outlined in the previous topic), it also refers to the chart as Chart 1; again, an absolute reference lacking flexibility should you wish to run this macro on a different chart. Also, the methods used to move and size the charts are difficult to understand and edit.
Hence, the solution is NOT to use the same code as the recorder but to write your own VBA using the **ChartObjects** object.

This is because **ChartObjects.Add** produces a chart equally as well as **Charts.Add**, but allows you to enter arguments for positioning and sizing the chart as it is created. This is because an embedded chart is contained in a **ChartObject** and it is the **ChartObject** that has the size and location properties. The chart inside the ChartObject is what you access to change most of the other properties.

And because a ChartObject can be created on an active sheet and a chart then put directly inside it, there is no need to create the additional variables in the previous examples. Hence, it produces much cleaner, more understandable code.

The example below, creates a clustered column chart on the same sheet that the data is selected from. It positions the chart in the top left hand corner of the work area (**Left**: 0 and **Top**: 0) and sizes it to 375 pixels wide by 225 pixels high. The actual size of the chart will depend on the resolution set on the computer monitor.

```vba
Sub AddPositionAndSizeChart()
    With ActiveSheet.ChartObjects.Add(Left:=0, Width:=375, Top:=0, Height:=225)
        .Chart.SetSourceData Source:=Selection
        .Chart.ChartType = xlColumnClustered
    End With
End Sub
```

To size a chart in inches for printing purposes (approximately), multiply the number of inches that you want by 72 pixels. Eg:

- **Height**: = 4 * 72 will create a chart 4 inches high
- **Width**: = 6 * 72 will create a chart 6 inches wide

You may find it more convenient, in a longer procedure, to define an **object variable** for the new chart created. Thus, if you need to refer to this chart later in the procedure, you can conveniently use the variable name.

The following example does the same as the two above, but sets a ChartObject variable for the new chart added.

```vba
Sub AddPositionAndSizeChart()
    Dim myCht As ChartObject
    Set myCht = ActiveSheet.ChartObjects.Add (Left:=100, Width:=375, Top:=75, Height:=225)
    myCht.Chart.SetSourceData Source:=Selection
    myCht.Chart.ChartType = xlColumnClustered
End Sub
```
In the same way as you can set a chart object’s size and position properties at its creation, you can position or size the chart at any time by changing the same properties. The following identifies a specific chart object and moves and sizes it:

```vba
Sub ResizeAndRepositionChart()
    With ActiveSheet.ChartObjects("Chart 1")
        .Left = 100
        .Width = 375
        .Top = 75
        .Height = 225
    End With
End Sub
```

The following does the same as above but to an active chart. Remember, it is not the chart that has the sizing and positioning properties, it is its container - the chart object. Hence, in this case, you have to refer to the chart object as the parent of the chart. Confusing?........yes!

```vba
Sub ResizeAndRepositionChart()
    With ActiveChart.Parent
        'The ChartObject is the chart's parent
        .Left = 100
        .Width = 375
        .Top = 75
        .Height = 225
    End With
End Sub
```

Rather than calculating a chart’s position using pixels, you can cover a specified range with a chart. The following procedure places a chart over the range D5:K25:

```vba
Sub CoverRangeWithChart()
    Dim cht As Chart Object
    Dim rng As Range

    Set cht = ActiveChart.Parent
    Set rng = ActiveSheet.Range("A1:G20")

    cht.Left = rng.Left
    cht.Width = rng.Width
    cht.Top = rng.Top
    cht.Height = rng.Height
End Sub
```
SIZING AND POSITIONING MULTIPLE CHARTS

Discussion

When working in Excel normally, if you wish to print several charts together on one sheet of paper, you need to create them as embedded charts on the same worksheet and then position and size them suitably to fit onto the page. This can be tedious and time consuming work so the example below illustrates positioning and sizing two named charts neatly and quickly one on top of the other.

The first part of the procedure measures the size (height and width) of the first chart and the second part arranges and positions the second chart directly underneath, and sizes it to identical proportions.

Sub AddChartBelow()

    Dim chtWdt As Integer
    Dim chtHgt As Integer
    Dim chtPosx As Integer
    Dim chtPosy As Integer

    chtWdt = ActiveSheet.ChartObjects("Chart 1").Width
    chtHgt = ActiveSheet.ChartObjects("Chart 1").Height
    chtPosx = ActiveSheet.ChartObjects("Chart 1").Top
    chtPosy = ActiveSheet.ChartObjects("Chart 1").Left

    ActiveSheet.ChartObjects("Chart 2").Top = chtPosx + chtHgt
    ActiveSheet.ChartObjects("Chart 2").Left = chtPosy
    ActiveSheet.ChartObjects("Chart 2").Width = chtWdt
    ActiveSheet.ChartObjects("Chart 2").Height = chtHgt

End Sub

If you have many charts on a worksheet, you may like to arrange them ALL neatly. The following procedure loops through any number of charts, resizes them the same and arranges them in regular rows and 3 columns:

Sub ArrangeMyCharts()

    Dim iChart As Long
    Dim nCharts As Long
    Dim dTop As Double
    Dim dLeft As Double
    Dim dHeight As Double
    Dim dWidth As Double
    Dim nColumns As Long

    dTop = 75 ' top of first row of charts
    dLeft = 100 ' left of first column of charts
dHeight = 225 ' height of all charts
dWidth = 375 ' width of all charts
nColumns = 3 ' number of columns of charts
nCharts = ActiveSheet.ChartObjects.Count

For iChart = 1 To nCharts
    With ActiveSheet.ChartObjects(iChart)
        .Height = dHeight
        .Width = dWidth
        .Top = dTop + Int((iChart - 1) / nColumns) * dHeight
        .Left = dLeft + ((iChart - 1) Mod nColumns) * dWidth
    End With
Next
End Sub

NAMING A CHART

Discussion
Charts created on their own sheet are easily identified by their sheet name. For example, Charts("Chart1").Activate will activate the chart on the sheet named Chart1. Any changes that you wish to make to that chart can then be referred to as ActiveChart, eg.
ActiveChart.PlotArea.Interior = vbCyan
ActiveChart.ChartArea.Interior = vbYellow
ActiveChart.Name = “South RegionSales”

Excel, however, gives embedded charts a unique name when they are created. The first chart on a sheet is named “Chart 1”; the next one “Chart 2” and so forth (there is a space between Chart and the number). The names are based on a sheet-by-sheet basis hence, you may have several Chart 1s in a workbook.

This can make it difficult to identify a specific chart that you want your VBA to act upon. As a result, it may be advantageous to give it a name of your own.

It is also important to note that it is NOT the embedded chart itself that is named but its container - the ChartObject. The examples below name embedded charts under various circumstances

Naming an active chart:
ActiveChart.Parent.Name = "Name of chart"

Naming a specific existing chart:
ActiveSheet.ChartObjects(3).Name = "Name of chart"
or...
ActiveSheet.ChartObjects("Old name of chart").Name = "New name of chart"
When creating a new embedded chart:

```vba
With ActiveSheet.ChartObjects.Add (Left:=100, Width:=375, Top:=75, Height:=225)
  .Chart.ChartType = xlXYScatterLines
  .Chart.SetSourceData Source:=Selection
  .Chart.Parent.Name = "Name of chart"
End With
```

Adding a Series to a Chart

Discussion

As with most structural and formatting changes in charts, the macro recorder is the easiest way of writing the code.

The following example adds a new series (a 2nd one) to an existing (active) chart

```vba
Sub AddNewSeries()
  ActiveChart.SeriesCollection.NewSeries
  ActiveChart.SeriesCollection(2).Values = "=Sheet1!R3C3:R5C3"
  ActiveChart.SeriesCollection(2).Name = "=Sheet1!R2C3"
End Sub
```

This is a good and effective piece of code but like many things that the macro recorder does, is not particularly flexible for two reasons:

1. It always adds a 2nd series to a chart irrespective of how many the chart already contains. This means that any current series are “pushed” out of place and hence, not plotted correctly.

2. It uses the R1C1 notation for the data.

The following piece of code has been adapted from the example above to add a new series in sequence and uses the more familiar A1 type notation (or any range reference, eg. Cells) for the data.

```vba
Sub AddNewSeries()
  With ActiveChart.SeriesCollection.NewSeries
    .Name = ActiveSheet.Range("G3")
    .Values = ActiveSheet.Range("G4:G14")
  End With
End Sub
```

Of course, you don’t have to specify absolute references. You could use a string for the name of the series or a named range for the values, eg.

```vba
Sub AddNewSeries()
  With ActiveChart.SeriesCollection.NewSeries
    .Name = "April"
    .Values = "=Sheet1!Apr_Sales"
  End With
End Sub
```
End Sub

As with other examples in this lesson, you can set an object variable for the new chart series being added. The following procedure assigns the variable myNewSer to the new chart series created.

Sub AddNewSeries()
    Dim MyNewSrs As Series
    Set MyNewSrs = ActiveChart.SeriesCollection.NewSeries
    With MyNewSrs
        .Name = "April"
        .Values = "=Sheet1!Apr_Sales"
    End With
End Sub

DELETING SERIES FROM A CHART

Discussion

The basic code for deleting a series is to identify it by its name and delete it. The name of the series will be as shown in the legend of the chart.

ActiveChart.SeriesCollection(“January”).Delete

To clear out ALL of a chart’s series, your code needs to loop through the entire series collection, each time deleting the first one until there are none left, eg.

Sub RemoveAllSeries()
    Do Until ActiveChart.SeriesCollection.Count = 0
        ActiveChart.SeriesCollection(1).Delete
    Loop
End Sub

The chart now appears completely blank and contains only the chart area. You can now add series as detailed in the previous topic.
EXERCISE

CHARTING WITH VBA

Tasks - To create and modify charts using VBA and ActiveX controls

1. Open the file Practice Charts.
2. The workbook already contains a procedure (CreateCharts) that adds a chart for the London data to Sheet1 and positions it and sizes it in the top left corner.
3. Add code to the procedure to create a separate chart from the Bristol data (use the code already in the procedure as a guide).
4. Add code to position the chart immediately below the first one and give it the same height and width.
5. Assign this procedure to the ActiveX command button on the worksheet.
6. Test and debug if necessary.
7. Add properties to the ActiveX combo box so that it is populated with the items in range A1 to A2 of Sheet2.
8. Add code to the “change” event for the combo box so that when an item is selected from it, both charts change type based on the item picked. Make use of the existing code for this event as a guide to what is required.
9. Test and debug the code.
10. Add code to the Plot by columns option button so that both charts are plotted by column if it is selected. Use the existing code for the Plot by rows button as a guide.
11. Test and debug the code.
12. Finally, assign to the “OnClick” event for the ActiveX check box, a procedure named FormatCharts that formats each chart as follows. Use the Object Browser or the macro recorder to help you find the correct properties to use:
13. Chart Area interior light blue (ColorIndex = 34).
14. Plot Area interior dark blue (ColorIndex = 49).
15. Series 1 interior yellow (ColorIndex = 6).
16. Series 2 interior lilac (ColorIndex = 39).
17. X and Y axis fonts 10 points.
   (Tip: You will need to add some error handling code that ignores the run time error generated if the chart doesn’t contain a Series 2).
18. Test the code and debug if necessary.
19. Save and close the workbook.
LESSON 6 –
WORKING WITH PIVOT TABLES

In this lesson, you will learn:

- Concepts of Pivot Table design
- How to create a Pivot Table using VBA
- How to structure the layout of a Pivot Table Using VBA
- How to refresh Pivot Tables using VBA
The Pivot Table Object

Discussion

Like many other objects in Excel, Pivot Tables come in two VBA forms:

1. as a “collection” - `PivotTables` - representing ALL the Pivot Tables on a worksheet, or
2. as individual objects - `PivotTables(<name>)` - representing a single Pivot Table.

There are few practical examples of using the `PivotTables` collection but one might be:

```
numPt = ActiveSheet.PivotTables.Count
```

... stores in the variable `numPt` the number of Pivot Tables on the active sheet.

There are, however, many practical examples of using the Pivot Table object singly, eg.

```
ActiveSheet.PivotTables("MyPivot").RefreshTable
```

... updates the named Pivot Table with any changes made to the source data

```
ActiveSheet.PivotTables("MyPivot1").RowGrand = False
ActiveSheet.PivotTables("MyPivot1").ColumnGrand = False
```

... removes row and column grand totals from the named Pivot Table (TRUE will add the grand totals).

```
ActiveSheet.PivotTables("MyPivot1").Format xlReport4
```

... adds the Report 4 AutoFormat style to the named Pivot Table.

```
ActiveSheet.PivotTables("MyPivot1").PivotSelect ""
```

... selects the entire named Pivot Table. That could then be followed by:

```
Selection.Clear       or
Selection.Copy       or
Selection.Font.Name = "Calibri"
```

... or any one of many other properties and methods applicable to a Pivot Table.
Creating a Pivot Table

Discussion

While it is easier to use the macro recorder to write the VBA code needed to create a Pivot Table, it’s useful to have an understand of the programming behind the recorded procedure in order to automate the various tasks that you may later want to perform.

Writing your own code is also preferable because it will invariably be more concise and understandable than that produced by the macro recorder.

There are several ways of creating a Pivot Table using VBA and the following follows a three step approach.

1. Set a “Pivot Cache” to store the source data for the Pivot Table. A “cache” is a temporary store for a copy of the original data. This cache can later be used in the same procedure for producing other Pivot Tables from the same source. The code for creating a Pivot Cache is as follows:

   **ActiveWorkbook.PivotCaches.Add (arguments)**

   The `Add` method above takes two arguments:

<table>
<thead>
<tr>
<th>SourceType</th>
<th>The type of data being used, viz.</th>
</tr>
</thead>
<tbody>
<tr>
<td>xlConsolidation</td>
<td>xlDatabase</td>
</tr>
<tr>
<td>xlDatabase</td>
<td>xlExternal</td>
</tr>
<tr>
<td>xlExternal</td>
<td>xlPivotTable</td>
</tr>
<tr>
<td>xlPivotTable</td>
<td>xlScenario</td>
</tr>
</tbody>
</table>

   In most cases, where an Excel list or table is being used, this would be `xlDatabase`.

   SourceData | The actual data being used.
   In most cases, where an Excel list or table is being used, this would something like `Range("B2:H20")`.

   Hence, the statement below sets a Pivot Cache named `pvCsh` that contains data from an Excel list or table that is stored on the sheet named `Data` in cells `B2` to `H20`:

   ```vba
   Set pvCsh = ActiveWorkbook.PivotCaches.Add(SourceType:=xlDatabase, SourceData:=Sheets("Data").Range("B2:H21"))
   ```

2. Create the basic Pivot Table “skeleton”. The code, assuming the Pivot Table is being created on the sheet that the macro is being run from, is as follows:

   **ActiveSheet.PivotTables.Add (arguments)**

   The `Add` method above takes four arguments, although only the first two are required.

   | PivotCache (Required) | The source data for the Pivot Table. This will be the |

   Hence, the statement below sets a Pivot Table on the sheet named `Sheet1` that uses the Pivot Cache created above:

   ```vba
   ActiveSheet.PivotTables.Add(PivotCache:=pvCsh, TableName:="Pivot1")
   ```
Hence, the statement below will add from the cache named `pvCsh`, a Pivot Table starting in cell A30 of the active sheet, and name it `MyPivot1`.

```vba
ActiveSheet.PivotTables.Add PivotCache:=pvCsh, TableDestination:=Range("A30"), TableName:="MyPivot1"
```

3. Add columns, rows and data to the Pivot Table.

The `PivotFields` method is generally used to add and/ or modify the columns, rows and data of a Pivot Table.

The `PivotFields` method has many additional methods and properties but in cases where it is used for placing or locating fields on a Pivot Table, the `Orientation` property is used.

Hence, in the example below, the `Region` field is grouped in the rows of the Pivot Table, the `Products` field into the columns and the `Sales` field used to provide the summarised data at the intersection of each column and row.

```vba
ActiveSheet.PivotTables("MyPivot1").PivotFields("Region").Orientation = xlRowField
ActiveSheet.PivotTables("MyPivot1").PivotFields("Product").Orientation = xlColumnField
ActiveSheet.PivotTables("MyPivot1").PivotFields("Sales").Orientation = xlDataField
```

These three statements could be abbreviated using a `With... End With` block as follows:

```vba
With ActiveSheet.PivotTables("MyPivot1")
  .PivotFields("Region").Orientation = xlRowField
  .PivotFields("Product").Orientation = xlColumnField
  .PivotFields("Sales").Orientation = xlDataField
End With
```
The example below creates a Pivot Table using the data in cells B2 to H20 and places the Pivot Table in the cell A30 on the same sheet. The entire procedure, therefore, for creating the Pivot Table described above is:

```vbnet
Sub CreatePivotTable()
    Dim pvCsh
    Set pvCsh = ActiveWorkbook.PivotCaches.Add(SourceType:=xlDatabase, SourceData:=Sheets("Data").Range("B2:H21"))
    ActiveSheet.PivotTables.Add PivotCache:=pvCsh, TableDestination:=Range("A30"), TableName:="myPivot1"
    With ActiveSheet.PivotTables("MyPivot1")
        .PivotFields("Region").Orientation = xlRowField
        .PivotFields("Product").Orientation = xlColumnField
        .PivotFields("Sales").Orientation = xlDataField
    End With
End Sub
```

While the code above may seem tedious to write, it compares favourably to the example below that creates the same Pivot Table using the macro recorder.

```vbnet
Sub MakePivotTable()
    ActiveWorkbook.PivotCaches.Add(SourceType:=xlDatabase, SourceData:="Sheet1!A1:G60").CreatePivotTable TableDestination:="", TableName:="PivotTable1", DefaultVersion:=xlPivotTableVersion10
    ActiveSheet.PivotTableWizard TableDestination:=ActiveSheet.Cells(3, 1)
    ActiveSheet.Cells(3, 1).Select
    With ActiveSheet.PivotTables("PivotTable1").PivotFields("Salesperson")
        .Orientation = xlRowField
        .Position = 1
    End With
    With ActiveSheet.PivotTables("PivotTable1").PivotFields("Product")
        .Orientation = xlColumnField
        .Position = 1
    End With
    ActiveSheet.PivotTables("PivotTable1").AddDataField ActiveSheet.PivotTables("PivotTable1").PivotFields("Sales"), "Sum of Sales", xlSum
End Sub
```

**NAMING PIVOT TABLES**

**Discussion**

Like many other Excel objects, Pivot Tables can be referred to in two ways:

1. **Numerically, using their index number.** Excel determines this number automatically based on the order that the Pivot Tables have been created.
Hence, if there are two Pivot Tables on a sheet, the “oldest” (the one created first) will have an index number of two. The “newest” Pivot Table will always have an index number of 1.

To add to the confusion, if a sheet contains, lets say, three Pivot Tables, index numbers 1, 2, and 3 and the second one to have been created is removed, the one that had index number 3 gets renumbered to index number 2.

This can make it very difficult to identify individual Pivot Tables but proves useful in looping through them using a For... Next statement block.

2. **By a name expressed as a string** (ie. as text). When you create a Pivot Table using the Wizard, you are not given the opportunity of naming it. Excel, therefore, gives it a default name - **PivotTable** followed by a **sequential number**.

This is in sharp contrast to the index number, because the FIRST Pivot Table created is given a name of **PivotTable1**, the second Pivot Table is named **PivotTable2** and so on.

Names of Pivot Tables are retained in the file that they were created, so even if you remove ALL the Pivot Tables on a sheet, the name of the next one created will always be named in sequence to the last one. Hence, you may have only one Pivot Table on a sheet, but its name could be PivotTable12. And to add to the confusion further, Pivot Tables names are numbered on a workbook basis, whereas index numbers are added on a sheet-by-sheet basis.

All this goes to show that it is advantageous when writing code that creates Pivot Tables, to give each one an easily identifiable name.

As shown in the example on page 96, a Pivot Table can be named when using VBA to create it. But in order to rename an existing Pivot Table, the Pivot Table has to be identified individually and then given a **Name** property, eg.

```
ActiveSheet.PivotTables("PivotTable1").Name = "Sales by Year and Month"
```

A useful piece of code for referring to a Pivot Table that has had a cell selected on it is:

```
ActiveCell.PivotTable
```

Hence,

```
ActiveCell.PivotTable.Name = "Sales by Year and Month"
```

... would rename a selected Pivot Table.

**MANIPULATING PIVOT TABLE FIELDS**

**Discussion**

Once a Pivot Table has been created either with the Wizard or by using VBA, there may come a need to automate “pivoting” the fields, hiding fields, removing fields, adding new ones or changing the calculation being performed.

In VBA, Pivot Table contains Pivot Fields that can have their methods used and properties changed. In the example on page 79 that creates a Pivot Table, we saw how **PivotField** was used to place fields from the source data into the required section, ie.
rows, column or data. PivotField can be used in a similar way to move, remove or add fields to a Pivot Table structure.

The examples below give the necessary code to undertake a variety of tasks on Pivot Fields.

**Move a field from columns to rows:**

```vba
ActiveSheet.PivotTables("name" or index).PivotFields("field name ").Orientation = xlRowField
```

**Move a field from rows to columns:**

```vba
ActiveSheet.PivotTables("name" or index).PivotFields("field name ").Orientation = xlColumnField
```

**Move a field to a different level**

```vba
ActiveSheet.PivotTables("name" or index).PivotFields("field name ").Position = 2
```

This results in the Year field (currently in position 1) being moved to become a sub-grouping of Region (position 2 in the hierarchy of levels). The pictures below show the effect of moving the Year field to position 2.

- **Remove a field from a Pivot Table**

  ```vba
  ActiveSheet.PivotTables("name" or index).PivotFields("field name ").Orientation = xlHidden
  ```

- **Add a field to a Pivot Table**

  ```vba
  ActiveSheet.PivotTables("name" or index).PivotFields("field name ").Orientation = xlRowField
  ```

  If the above code is applied to a Pivot Table already containing row fields, the new field is added at the right (the lowest level) of the existing fields. It can then be moved to a higher level by using the Position property for the field as shown above.
**Change the summary calculation of the data field**

`ActiveSheet.PivotTables("name" or index).PivotFields("Sum of <field name> ").Function = xlCount`

The above example changes the calculation to count. The value of the Function property can be any of the following:

- `xlAverage`
- `xlCountNums`
- `xlMin`
- `xlStDev`
- `xlSum`
- `xlVar`
- `xlCount`
- `xlMax`
- `xlProduct`
- `xlStdDevP`
- `xlUnknown` (applies only to OLAP data sources)
- `xlVarP`

**Deleting a Pivot Table**

`ActiveSheet.PivotTables("name" or index).PivotSelect ""
Selection.Clear`

---

**Refreshing Pivot Tables**

**Discussion**

A drawback to Pivot Tables is that you can only refresh them automatically by setting the *Refresh on open* option via the *PivotTable Options* dialog box. The *Refresh every <x> minutes* option is only available in rare cases where the Pivot Table is created from data that is being accessed by Excel from an external source such as an Access table, a SQL server or an Oracle data management system, etc.

This is not usually an issue with seasoned Excel users who are familiar with the “Refresh” button on the Pivot Table toolbar. But it is not unknown for less experienced people to change the source data of a Pivot Table and then submit the Pivot Table without realising that its values have not been recalculated.

It would be useful, therefore, to refresh Pivot Tables using VBA code whenever a worksheet is activated (using the *Worksheet_Activate* event), or by giving users clearly labelled worksheet buttons that run macros to refresh Pivot Tables.

**Refresh a Single Pivot Table by Name**

`ActiveSheet.PivotTables("name" or index).RefreshTable`
Refresh all Pivot Tables on a Worksheet

For Each pvTab In ActiveSheet.PivotTables
    pvTab.RefreshTable
Next pvTab

Refresh Specific Pivot Tables on a Worksheet

For Each pvTab In ActiveSheet.PivotTables
    Select Case pvTab.Name
        Case "PivotTable1", "PivotTable4", "PivotTable8"
            pvTab.RefreshTable
        Case Else
            End Select
    Next pvTab

The example above used a Case Select statement block to evaluate the Pivot Table names. If preferred, an If… Then… Else statement block can be used, ie.

If pvTab.Name = "PivotTable1" Or pvTab.Name = "PivotTable4" Or pvTab.Name = "PivotTable8" Then
    pvTab.RefreshTable
End If

Refresh All Pivot Tables in a Workbook

For Each ws In ActiveWorkbook.Worksheets
    For Each pt In ws.PivotTables
        pt.RefreshTable
    Next pt
Next ws

NOTE: When refreshing a Pivot Table, ALL Pivot Tables in a workbook that are based on the same data source (Pivot Cache) are simultaneously refreshed. It may not be necessary, therefore, to loop through them all as shown above.
**EXERCISE**

**WORKING WITH PIVOT TABLES IN VBA**

**Task - Automate the production and modification of a Pivot Table.**

Open the file **Practice Pivot**.

The **SurveyData** sheet contains the results of a survey that need to be summarised using Pivot Tables.

Your task is to create a sub procedure that will create four Pivot Tables on a new sheet named **Summary**, each totalling the count of one of the responses received to the four questions.

| Table 1 | Name: “Convenient”  
|         | Table Destination: A1  
| Rows: Store locations are convenient  
| Columns: Sex  
| Data: Store locations are convenient  
| Function: Count  
| Table 2 | Name: “Friendly”  
|         | Table Destination: A12  
| Rows: Employees are friendly  
| Columns: Sex  
| Data: Employees are friendly  
| Function: Count  
| Table 3 | Name: “Pricing”  
|         | Table Destination: A24  
| Rows: Pricing is competitive  
| Columns: Sex  
| Data: Pricing is competitive  
| Function: Count  
| Table 4 | Name: “Recommend”  
|         | Table Destination: A36  
| Rows: I would recommend your company  
| Columns: Sex  
| Data: I would recommend your company  
| Function: Count  

Part of the code is already written for you. The sub procedure, **CreatePivotAnalysis**, creates a PivotCache from the data and on the **SurveyData** sheet and produces the first Pivot Table for the “Store locations are convenient” question. Expand this procedure.
using the information in the table above, to produce Pivot Tables for the other three questions as shown in the screen shot above.

Test and debug the code.

Add code to replace 1, 2, 3, 4 and 5 with Strongly Disagree, Disagree, Undecided, Agree, Strongly Agree. (Tip: Use ActiveSheet.Range("A:A").Replace "1", "Strongly Disagree" and repeat for each option).

Test and debug the code.

The ChangeToPercent sub procedure on module sheet 2 of this workbook is designed to change the type of calculation performed in the data area of a Pivot Table to a percentage of the column.

Modify this sub procedure so that it runs on ALL Pivot Tables on an active sheet (Tip: Use a loop).

Repeat 6 above for the ChangeToCount sub procedure.

Test and debug the code.

Finally, call into the CreatePivotAnalysis sub procedure, the sub procedure named MakeButtons. This creates two macro buttons to run the ChangeToPercent and ChangeToCount.

Assign the CreatePivotAnalysis sub procedure to the button on the SurveyData sheet.

Run the CreatePivotAnalysis sub procedure.

Test and debug the code.

Save and close the workbook.
LESSON 7 –
COMMUNICATING WITH OTHER OFFICE APPLICATIONS

In this lesson, you will learn how to:

- Create a reference to another Office application
- Write VBA code to interact with other Office applications
OVERVIEW

Discussion

Automation (formerly OLE Automation) is a feature that programs use to expose their objects to development tools, macro languages, and other programs that support Automation. For example, a spreadsheet program may expose a worksheet, chart, cell, or range of cells, each as a different type of object. A word processor might expose objects such as an application, a document, a paragraph, a sentence, a bookmark, or a selection.

When a program supports Automation, you can use Visual Basic for Applications to access the objects it exposes. You manipulate these objects in Visual Basic by invoking methods on the object or by getting and setting the object's properties. We will look at working with other Microsoft Office programs.

There are four main steps to automating another Office application (where * is the name of the application, eg. Word, Excel, PowerPoint, Access, Visio or Outlook).

- Add a reference to the application’s object library.
- Declare a variable as a * object type.
- Assign the object returned by the CreateObject function to the object variable (application) you declared in step 2.
- Use the properties and methods of the object variable to automate *

ADDING A REFERENCE TO THE APPLICATION’S OBJECT LIBRARY

This will help by displaying a members list from where you can identify and select appropriate methods and properties for the relevant object. For example, adding the Microsoft Word Object Library reference allows your program to access Microsoft Word Online Help and the Microsoft Word Visual Basic for Applications constants, properties, and methods. Note that the Word Object Library reference is required to automate the Word object types directly.

Adding a reference to the Object Library is called early binding.

Procedures

1. Launch the VB Editor.
2. Select the Tools menu.
3. Select References... .
4. In the list of Available References; click to select the relevant check box.
5. Click OK.
**DECLARING THE OBJECT VARIABLE**

To declare an object variable, you dimension the variable just as you `DIM` any variable, except that you specify the type when declaring the object. For example, `Word.Application`, `Document`, and `Paragraph` are separate Word Objects.

The following example declares the variable `wrdApp` as an object of type `Word.Application`:

```vba
Dim wrdApp as Word.Application
```

It would be similar for other Office applications, eg.

```vba
Dim pptApp as PowerPoint.Application
Dim accApp as Access.Application
Dim olkApp as Outlook.Application
```

**Procedures**

1. Launch the VB Editor.
2. Identify in the Project Explorer pane the VBAProject that you wish to write the procedure in.
3. Add a module to the project or activate an existing module sheet.
4. Start a new sub procedure.
5. Declare (dimension) the object variable that you intend using in the sub procedure. Use the examples above as a guide.
## Setting the Variable

### Discussion

There are two VBA functions you can use to "bind" the already declared object variable to the application: **CreateObject** and **GetObject**. The primary differences are that the CreateObject function creates a new instance of the program, while the GetObject function uses an existing, or already running instance of it. You can also use GetObject to bind your object variable to a specific Word document.

The following example binds the `wrdApp` variable to Word by using the CreateObject function.

```vba
Dim wrdApp as Word.Application
Set wrdApp = CreateObject("Word.Application")
```

It would be similar for other Office applications, eg.

```vba
Dim pptApp as Powerpoint.Application
Set pptApp = CreateObject("Powerpoint.Application")
```

Or…

```vba
Dim olkApp as Outlook.Application
Set olkApp = CreateObject("Outlook.Application")
```

The following example binds the `wrdApp` variable to a specific Word document.

```vba
Dim wrdApp As Word.Document
Set wrdApp = GetObject("s:\SharedDocs\Report.doc")
```

The following example binds the `wrdApp` variable to an existing instance of Word

```vba
Dim wrdApp As Word.Document
Set wrdApp = GetObject( , "Word.Application")
```

---

It is recommended to use only the **CreateObject** function to automate Word. The **GetObject** function can cause unpredictable behaviour if WordMail is running or if a Word document is embedded inside of another program.
USE METHODS AND PROPERTIES TO AUTOMATE THE OTHER APPLICATION

Discussion

When you complete the steps above, you can use the object variable to automate Word. This does, however, require knowledge of and experience at using the VBA language of the other Office application. Suggestions for obtaining help and guidance are:

- Recording macros in the other program (though not all Office applications have a macro recorder, eg. Outlook and Access).
- Seeking examples by searching the Internet.
- Using the VB Editor Help.
- Using the Members List.

EXAMPLES

Below are some simple examples for automating Word, PowerPoint and Outlook.

1. The following sample macro uses automation to create a Word object, create a new document, add some text, and save the document.

```vba
Sub AutomateWord()

' Declare the variable.
Dim objWD As Word.Application

' Set the variable (runs new instance of Word)
Set objWD = CreateObject("Word.Application")

' Add a new document
objWD.Documents.Add

' Add some text
objWD.Selection.TypeText "This is some text."

' Save the document
objWD.ActiveDocument.SaveAs filename:="mydoc.doc"

' Quit Word
objWD.Quit

' Clear the variable from memory
Set objWD = Nothing

End Sub
```
EXERCISE

CONTROLLING AND COMMUNICATING WITH OTHER APPLICATIONS

To use automation to carry out actions across MS Office applications

1. Open the file named, Talk to Word.
2. Add code to the sub procedure named, TakeToWord so that it:
   - copies the range A1 to E5 on Sheet1;
   - launches Word;
   - adds a document;
   - formats the selection to Bold, font size 18;
   - adds a heading, Quarter 1 Sales at the top of the Word document;
   - centrally aligns the heading
   - adds two blanks lines below;
   - copies the clipboard contents into the Word document;
   - closes Word with a prompt to save the file.
3. Run and test your procedure. Debug if necessary.
4. Create another sub procedure that copies the chart on Sheet2 of the workbook and pastes it into Slide6 of the PowerPoint presentation named, Annual AGM.
5. Run and test your procedure. Debug if necessary.
6. Save and close all open files.

Optional extra (if time)
Add additional code as necessary to prompt the user with an input box what heading to use in the Word document.
# APPENDIX I – LIST OF TRAPPABLE ERRORS AND THEIR CODES

Source:


<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Return without GoSub</td>
</tr>
<tr>
<td>5</td>
<td>Invalid procedure call</td>
</tr>
<tr>
<td>6</td>
<td>Overflow</td>
</tr>
<tr>
<td>7</td>
<td>Out of memory</td>
</tr>
<tr>
<td>9</td>
<td>Subscript out of range</td>
</tr>
<tr>
<td>10</td>
<td>This array is fixed or temporarily locked</td>
</tr>
<tr>
<td>11</td>
<td>Division by zero</td>
</tr>
<tr>
<td>13</td>
<td>Type mismatch</td>
</tr>
<tr>
<td>14</td>
<td>Out of string space</td>
</tr>
<tr>
<td>16</td>
<td>Expression too complex</td>
</tr>
<tr>
<td>17</td>
<td>Can't perform requested operation</td>
</tr>
<tr>
<td>18</td>
<td>User interrupt occurred</td>
</tr>
<tr>
<td>20</td>
<td>Resume without error</td>
</tr>
<tr>
<td>28</td>
<td>Out of stack space</td>
</tr>
<tr>
<td>35</td>
<td>Sub, Function, or Property not defined</td>
</tr>
<tr>
<td>47</td>
<td>Too many code resource or DLL application clients</td>
</tr>
<tr>
<td>48</td>
<td>Error in loading code resource or DLL</td>
</tr>
<tr>
<td>49</td>
<td>Bad code resource or DLL calling convention</td>
</tr>
<tr>
<td>51</td>
<td>Internal error</td>
</tr>
<tr>
<td>52</td>
<td>Bad file name or number</td>
</tr>
<tr>
<td>53</td>
<td>File not found</td>
</tr>
<tr>
<td>54</td>
<td>Bad file mode</td>
</tr>
<tr>
<td>55</td>
<td>File already open</td>
</tr>
<tr>
<td>57</td>
<td>Device I/ O error</td>
</tr>
<tr>
<td>58</td>
<td>File already exists</td>
</tr>
<tr>
<td>59</td>
<td>Bad record length</td>
</tr>
<tr>
<td>61</td>
<td>Disk full</td>
</tr>
<tr>
<td>62</td>
<td>Input past end of file</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>63</td>
<td>Bad record number</td>
</tr>
<tr>
<td>67</td>
<td>Too many files</td>
</tr>
<tr>
<td>68</td>
<td>Device unavailable</td>
</tr>
<tr>
<td>70</td>
<td>Permission denied</td>
</tr>
<tr>
<td>71</td>
<td>Disk not ready</td>
</tr>
<tr>
<td>74</td>
<td>Can't rename with different drive</td>
</tr>
<tr>
<td>75</td>
<td>Path/File access error</td>
</tr>
<tr>
<td>76</td>
<td>Path not found</td>
</tr>
<tr>
<td>91</td>
<td>Object variable or With block variable not set</td>
</tr>
<tr>
<td>92</td>
<td>For loop not initialized</td>
</tr>
<tr>
<td>93</td>
<td>Invalid pattern string</td>
</tr>
<tr>
<td>94</td>
<td>Invalid use of Null</td>
</tr>
<tr>
<td>97</td>
<td>Can't call Friend procedure on an object that is not an instance of the defining class</td>
</tr>
<tr>
<td>98</td>
<td>A property or method call cannot include a reference to a private object, either as an argument or as a return value</td>
</tr>
<tr>
<td>298</td>
<td>System resource or DLL could not be loaded</td>
</tr>
<tr>
<td>320</td>
<td>Can't use character device names in specified file names</td>
</tr>
<tr>
<td>321</td>
<td>Invalid file format</td>
</tr>
<tr>
<td>322</td>
<td>Can't create necessary temporary file</td>
</tr>
<tr>
<td>325</td>
<td>Invalid format in resource file</td>
</tr>
<tr>
<td>327</td>
<td>Data value named not found</td>
</tr>
<tr>
<td>328</td>
<td>Illegal parameter; can't write arrays</td>
</tr>
<tr>
<td>335</td>
<td>Could not access system registry</td>
</tr>
<tr>
<td>336</td>
<td>Component not correctly registered</td>
</tr>
<tr>
<td>337</td>
<td>Component not found</td>
</tr>
<tr>
<td>338</td>
<td>Component did not run correctly</td>
</tr>
<tr>
<td>360</td>
<td>Object already loaded</td>
</tr>
<tr>
<td>361</td>
<td>Can't load or unload this object</td>
</tr>
<tr>
<td>363</td>
<td>Control specified not found</td>
</tr>
<tr>
<td>364</td>
<td>Object was unloaded</td>
</tr>
<tr>
<td>365</td>
<td>Unable to unload within this context</td>
</tr>
<tr>
<td>368</td>
<td>The specified file is out of date. This program requires a later version</td>
</tr>
<tr>
<td>371</td>
<td>The specified object can't be used as an owner form for Show</td>
</tr>
<tr>
<td>380</td>
<td>Invalid property value</td>
</tr>
<tr>
<td>381</td>
<td>Invalid property-array index</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>382</td>
<td>Property Set can't be executed at run time</td>
</tr>
<tr>
<td>383</td>
<td>Property Set can't be used with a read-only property</td>
</tr>
<tr>
<td>385</td>
<td>Need property-array index</td>
</tr>
<tr>
<td>387</td>
<td>Property Set not permitted</td>
</tr>
<tr>
<td>393</td>
<td>Property Get can't be executed at run time</td>
</tr>
<tr>
<td>394</td>
<td>Property Get can't be executed on write-only property</td>
</tr>
<tr>
<td>400</td>
<td>Form already displayed; can't show modally</td>
</tr>
<tr>
<td>402</td>
<td>Code must close topmost modal form first</td>
</tr>
<tr>
<td>419</td>
<td>Permission to use object denied</td>
</tr>
<tr>
<td>422</td>
<td>Property not found</td>
</tr>
<tr>
<td>423</td>
<td>Property or method not found</td>
</tr>
<tr>
<td>424</td>
<td>Object required</td>
</tr>
<tr>
<td>425</td>
<td>Invalid object use</td>
</tr>
<tr>
<td>429</td>
<td>Component can't create object or return reference to this object</td>
</tr>
<tr>
<td>430</td>
<td>Class doesn't support Automation</td>
</tr>
<tr>
<td>432</td>
<td>File name or class name not found during Automation operation</td>
</tr>
<tr>
<td>438</td>
<td>Object doesn't support this property or method</td>
</tr>
<tr>
<td>440</td>
<td>Automation error</td>
</tr>
<tr>
<td>442</td>
<td>Connection to type library or object library for remote process has been lost</td>
</tr>
<tr>
<td>443</td>
<td>Automation object doesn't have a default value</td>
</tr>
<tr>
<td>445</td>
<td>Object doesn't support this action</td>
</tr>
<tr>
<td>446</td>
<td>Object doesn't support named arguments</td>
</tr>
<tr>
<td>447</td>
<td>Object doesn't support current locale setting</td>
</tr>
<tr>
<td>448</td>
<td>Named argument not found</td>
</tr>
<tr>
<td>449</td>
<td>Argument not optional or invalid property assignment</td>
</tr>
<tr>
<td>450</td>
<td>Wrong number of arguments or invalid property assignment</td>
</tr>
<tr>
<td>451</td>
<td>Object not a collection</td>
</tr>
<tr>
<td>452</td>
<td>Invalid ordinal</td>
</tr>
<tr>
<td>453</td>
<td>Specified code resource not found</td>
</tr>
<tr>
<td>454</td>
<td>Code resource not found</td>
</tr>
<tr>
<td>455</td>
<td>Code resource lock error</td>
</tr>
<tr>
<td>457</td>
<td>This key is already associated with an element of this collection</td>
</tr>
<tr>
<td>458</td>
<td>Variable uses a type not supported in Visual Basic</td>
</tr>
<tr>
<td>459</td>
<td>This component doesn't support the set of events</td>
</tr>
<tr>
<td>460</td>
<td>Invalid Clipboard format</td>
</tr>
<tr>
<td>Error Code</td>
<td>Error Description</td>
</tr>
<tr>
<td>------------</td>
<td>------------------</td>
</tr>
<tr>
<td>461</td>
<td>Method or data member not found</td>
</tr>
<tr>
<td>462</td>
<td>The remote server machine does not exist or is unavailable</td>
</tr>
<tr>
<td>463</td>
<td>Class not registered on local machine</td>
</tr>
<tr>
<td>480</td>
<td>Can't create AutoRedraw image</td>
</tr>
<tr>
<td>481</td>
<td>Invalid picture</td>
</tr>
<tr>
<td>482</td>
<td>Printer error</td>
</tr>
<tr>
<td>483</td>
<td>Printer driver does not support specified property</td>
</tr>
<tr>
<td>484</td>
<td>Problem getting printer information from the system. Make sure the printer is set up correctly</td>
</tr>
<tr>
<td>485</td>
<td>Invalid picture type</td>
</tr>
<tr>
<td>486</td>
<td>Can't print form image to this type of printer</td>
</tr>
<tr>
<td>520</td>
<td>Can't empty Clipboard</td>
</tr>
<tr>
<td>521</td>
<td>Can't open Clipboard</td>
</tr>
<tr>
<td>735</td>
<td>Can't save file to TEMP directory</td>
</tr>
<tr>
<td>744</td>
<td>Search text not found</td>
</tr>
<tr>
<td>746</td>
<td>Replacements too long</td>
</tr>
<tr>
<td>31001</td>
<td>Out of memory</td>
</tr>
<tr>
<td>31004</td>
<td>No object</td>
</tr>
<tr>
<td>31018</td>
<td>Class is not set</td>
</tr>
<tr>
<td>31027</td>
<td>Unable to activate object</td>
</tr>
<tr>
<td>31032</td>
<td>Unable to create embedded object</td>
</tr>
<tr>
<td>31036</td>
<td>Error saving to file</td>
</tr>
<tr>
<td>31037</td>
<td>Error loading from file</td>
</tr>
</tbody>
</table>
APPENDIX II – ADDING INTERACTIVITY TO A MESSAGE BOX

Discussion

Examples of where an interactive message box might be required are where confirmation is required to proceed with the next part of a procedure, or in error handling, eg.

Example A

Example B

To make the message box interactive, the arguments must be put inside brackets. The following code will display Example A above.

**MsgBox** ("Do you want to continue deleting the data", vbYesNo, "Delete Confirm")

The **buttons** argument consists of constants or values from each of the following three groups:

**Number and type of button:**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbOKOnly</td>
<td>0</td>
<td>OK button only</td>
</tr>
<tr>
<td>vbOKCancel</td>
<td>1</td>
<td>OK and Cancel buttons</td>
</tr>
<tr>
<td>vABortRetryIgnore</td>
<td>2</td>
<td>Abort, Retry and Ignore buttons</td>
</tr>
<tr>
<td>vbYesNoCancel</td>
<td>3</td>
<td>Yes, No and Cancel buttons</td>
</tr>
<tr>
<td>vbYesNo</td>
<td>4</td>
<td>Yes and No buttons</td>
</tr>
<tr>
<td>vbRetryCancel</td>
<td>5</td>
<td>Retry and Cancel buttons</td>
</tr>
</tbody>
</table>

**Icon style:**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Display</th>
<th>Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbCritical</td>
<td>16</td>
<td>Critical Message icon.</td>
<td>![Critical Message Icon]</td>
</tr>
<tr>
<td>vbQuestion</td>
<td>32</td>
<td>Warning Query icon.</td>
<td>![Warning Query Icon]</td>
</tr>
</tbody>
</table>
Default Button:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbDefaultButton1</td>
<td>0</td>
<td>First button is default</td>
</tr>
<tr>
<td>vbDefaultButton2</td>
<td>256</td>
<td>Second button is default</td>
</tr>
<tr>
<td>vbDefaultButton3</td>
<td>512</td>
<td>Third button is default</td>
</tr>
</tbody>
</table>

The **buttons** argument of the message box function can hold three pieces of information separated by a plus (+) symbol. Using the following code as the **buttons** argument will produce Example C below.

```
vbYesNo + vbQuestion + vbDefaultButton2
```

A more concise method of writing the above sample of code would be to use the numeric values for the arguments, eg.

```
4 + 32 + 256
```

It is easier, however, to remember the vb constants, viz. vbYesNo + vbQuestion + vbDefaultButton2.

**Procedures**

1. Position the cursor in the sub procedure code where you want to place the statement.
2. Type a variable name to store the value of whichever button is clicked in the message box, eg. response.
3. Type **=**.
4. Type **MsgBox**.
5. Type an opening bracket (**(`**).
6. Type a speech mark (**Shift** 2).
7. Type a prompt for the message box, i.e. the message that you want it to display, e.g. Do you want to continue?

8. Type a speech mark (Shift 2).

9. Type a comma.

10. Type the necessary value to indicate which buttons you want the message box to display, e.g. vbYesNo.

11. If you wish to add an icon and/or default button to the message box, type a plus symbol (+). If you do not wish to add an icon and/or a default button to the message box, go to 15 below.

12. Type the necessary value to indicate which icon to display, e.g. vbQuestion

13. Type a plus symbol (+).

14. Type the necessary value to indicate which default button you wish to set, e.g. vbDefaultButton2

15. Type a comma.

16. If you wish to add a title to the message box, type a comma. If you do not wish to add a title to the message box, go to 20 below.

17. Type a speech mark (Shift 2).

18. Type the title text.

19. Type a speech mark (Shift 2).

20. Type a closing bracket ().


22. Add additional code as necessary.

**Responding to an Interactive Message Box**

The value returned by the function depends upon which button is pressed. The value is returned as a constant, which is equal in value to a number. The constant or the value can be tested by the procedure, usually by means of an IF statement:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Button Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbOK</td>
<td>1</td>
<td>OK</td>
</tr>
<tr>
<td>vbCancel</td>
<td>2</td>
<td>Cancel</td>
</tr>
<tr>
<td>vbAbort</td>
<td>3</td>
<td>Abort</td>
</tr>
<tr>
<td>vbRetry</td>
<td>4</td>
<td>Retry</td>
</tr>
<tr>
<td>vbIgnore</td>
<td>5</td>
<td>Ignore</td>
</tr>
</tbody>
</table>
In the following example, the message box offers a yes/no response. If the user clicks yes, then the procedure will delete all the data from the sheet. If the user clicks no, the procedure will terminate.

```
If MsgBox("Do you want to delete all data", vbYesNo + vbCritical) = vbYes Then
    ActiveSheet.Cells.Clear
End If
```

The result of the message box (ie. whether the yes or no button is clicked) can be stored in a variable and the code could also be written:

```
Dim response as Byte
response = MsgBox("Do you want to delete all data", vbYesNo + vbCritical)
If response = vbYes Then  
or  If response = 6 Then
    ActiveSheet.Cells.Clear
End If
```

**Procedures**

1. (The procedure below assumes a message box containing Yes and No buttons. It can be adapted, however, to respond to any set of buttons (eg. vbOKCancel, vbRetryCancel etc.))
2. Position the cursor in the sub procedure code where you want to place the statement.
3. Create an interactive message box as described in the previous topic of this lesson.
4. Press Enter.
5. Type If.
6. Type a space.
7. Type the variable name that you have used to store the response from the message box.
8. Type a space.
9. Type = vbYes.
10. Type a space.
11. Type Then.
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12. Type the statements that you want the sub procedure to perform if the user has clicked the <strong>Yes</strong> button.</td>
<td></td>
</tr>
<tr>
<td>13. Press <strong>Enter</strong>.</td>
<td></td>
</tr>
<tr>
<td>14. Type <strong>Else</strong>.</td>
<td></td>
</tr>
<tr>
<td>15. Press <strong>Enter</strong>.</td>
<td></td>
</tr>
<tr>
<td>16. Type the statements that you want the sub procedure to perform if the user has clicked the <strong>No</strong> button.</td>
<td></td>
</tr>
<tr>
<td>17. Press <strong>Enter</strong>.</td>
<td></td>
</tr>
<tr>
<td>18. Type <strong>End If</strong>.</td>
<td></td>
</tr>
<tr>
<td>19. Press <strong>Enter</strong>.</td>
<td></td>
</tr>
<tr>
<td>20. Add additional code as necessary.</td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX III – SOLUTIONS TO EXERCISES

Dealing with Potential Errors (Page 17)

Sub CircData()
Const PI = 3.142
Radius = InputBox("Type the radius of your circle in centimetres", "Circle Data")
If IsNumeric(Radius) Then
    CircArea = PI * (Radius ^ 2)
    CircCircumf = 2 * PI * Radius
    MsgBox "The area of the circle is: " & CircArea & vbCrLf & vbCrLf & 
        "The circumference of the circle is: " & CircCircumf
Else
    MsgBox "Input not valid"
End If
End Sub

Sub DeleteOldFile()
Dim fileToRemove As String
On Error GoTo errhandl
fileToRemove = InputBox("Type name of file to delete from the current folder", "Delete File")
If fileToRemove = "" Then
    Exit Sub
Else
    Kill ThisWorkbook.Path & "\" & fileToRemove & ".xls"
    MsgBox "File successfully deleted"
End If
Exit Sub
errhandl:
If Err.Number = 53 Then
    response = MsgBox("File not found. Do you want to try again?", vbYesNo)
    If response = vbYes Then
        fileToRemove = InputBox("Type name of file to delete from the current folder", "Delete File")
        Resume
    End If
End If
End If
End If
If Err.Number = 70 Then
    MsgBox "File is currently open. Close the file and run the macro again"
End If
End Sub
Automating a worksheet with ActiveX controls (Page 91)

Event procedures for controls

Private Sub CheckBox1_Change()
    If Me.CheckBox1 Then
        Range("B8").Value = 0.1
    Else
        Range("B8").Value = 0
    End If
End Sub

Private Sub ComboBox1_DropButtonClick()
    Me.ComboBox1.BoundColumn = 1
    Range("B1").Value = Me.ComboBox1.Value
    Me.ComboBox1.BoundColumn = 2
    Range("B3").Value = Me.ComboBox1.Value / 100
End Sub

Private Sub SpinButton1_Change()
    Range("B5").Value = Me.SpinButton1.Value
End Sub

Event procedures for workbook

Private Sub Workbook_BeforeClose(Cancel As Boolean)
    Application.EnableEvents = True
End Sub

Private Sub Workbook_Open()
    Sheets("Sheet1").Protect UserInterfaceOnly:=True
End Sub
Handling Events (Page 64)

Event procedure for Sheet1

Private Sub Worksheet_SelectionChange(ByVal Target As Range)
    Cells.Interior.ColorIndex = xlNone
    Target.EntireColumn.Interior.Color = vbYellow
    Target.EntireRow.Interior.Color = vbYellow
End Sub

Event procedures for Sheet2

Dim origVal
Dim origCel

Private Sub Worksheet_SelectionChange(ByVal Target As Range)
    origVal = Target.Value
    origCel = Target.Address(0, 0)
End Sub

Private Sub Worksheet_Change(ByVal Target As Range)
    If Target.Count = 1 Then
        Application.EnableEvents = False
        On Error GoTo ws_exit
        If Not IsNumeric(Target.Value) Or Target.Value > 100 Then
            Target.Value = origVal
            Range(origCel).Select
            MsgBox "Entry not valid", vbInformation
        End If
    ws_exit:
        Application.EnableEvents = True
    End If
End Sub

Event procedures for Sheet3
Private Sub Worksheet_Change(ByVal Target As Range)
    Dim chngCel As String
    Dim origVal As Variant
    On Error GoTo errhandle
    chngCel = Target.Address(False, False)
    origVal = Target.Value
    If Intersect(Target, Sheets("Sheet2").Range("E1:E20")) Is Nothing Then
        Exit Sub
    Else
        If Not IsNumeric(Target.Value) Or Target.Value > 100 Then
            Application.EnableEvents = False
            MsgBox "Entry not valid"
            Range(chngCel).Select
            Range(chngCel).Value = origVal
            Application.EnableEvents = True
        End If
    End If
    Exit Sub
errhandle:
    Application.EnableEvents = True
End Sub

Events procedures for Workbook

Private Sub Workbook_BeforeClose(Cancel As Boolean)
    Sheets("Sheet1").Activate
    Cells(1).Select
End Sub

Private Sub Workbook_Open()
    UserForm1.Show
End Sub

Event procedure for user form
Private Sub UserForm_Activate()
    Application.OnTime Now + TimeValue("00:00:03"), "CloseSplash"
End Sub
Charting with VBA (Page 74)

Event procedures for Sheet1

Private Sub butRun_Click()
    CreateCharts
End Sub

Private Sub chkColour_Change()
    If Me.chkColour = True Then
        For Each cht In ActiveSheet.ChartObjects
            If cht.Chart.ChartType = xlLineMarkers Then
                For i = 1 To numser
                Next i
            Else
                For i = 1 To numser
                Next i
            End If
        Next cht
    End If
End Sub

Private Sub cmbType_Change()
    If Me.cmbType.Value = "Column" Then
        For Each cht In ActiveSheet.ChartObjects
            cht.Chart.ChartType = xlColumnClustered
        Next cht
    End If
End Sub
If Me.cmbType.Value = "Bar" Then
   For Each cht In ActiveSheet.ChartObjects
      cht.Chart.ChartType = xlBarClustered
   Next cht
   End If
If Me.cmbType.Value = "Line" Then
   For Each cht In ActiveSheet.ChartObjects
      cht.Chart.ChartType = xlLineMarkers
   Next cht
   End If
End Sub

Private Sub optCol_Click()
   For Each cht In ActiveSheet.ChartObjects
      cht.Chart.PlotBy = xlColumns
   Next cht
   End Sub

Private Sub optRows_Click()
   For Each cht In ActiveSheet.ChartObjects
      cht.Chart.PlotBy = xlRows
   Next cht
   End Sub

Sub temp()
End Sub

Sub procedures on separate module sheet

Sub CreateCharts()
   Dim chartRange1 As Range
   Dim chartRange2 As Range
Specify the ranges to be used
Set chartRange1 = Range("A1:B3") ' London
Set chartRange2 = Union(Range("A1:A3"), Range("C1:C3")) ' Bristol

Reset worksheet controls to default
ActiveSheet.OLEObjects("cmbType").Object.Value = "Column"
ActiveSheet.OLEObjects("optRows").Object.Value = False
ActiveSheet.OLEObjects("optCol").Object.Value = True
ActiveSheet.OLEObjects("chkColour").Object.Value = False

Delete any existing charts
For Each cht In ActiveSheet.ChartObjects
    cht.Delete
Next cht

Create the chart
Charts.Add
ActiveChart.ChartType = xlColumnClustered
ActiveChart.SetSourceData Source:=chartRange1
ActiveChart.Location where:=xlLocationAsObject, Name:="Sheet1"
ActiveChart.Parent.Name = "London"

Position and size the chart
With ActiveSheet.ChartObjects("London")
    .Left = 0
    .Top = 0
    .Height = 200
    .Width = 250
End With
Charts.Add
ActiveChart.ChartType = xlColumnClustered
ActiveChart.SetSourceData Source:=chartRange2
ActiveChart.Location where:=xlLocationAsObject, Name:="Sheet1"
ActiveChart.Parent.Name = "Bristol"
' Position and size the chart
    With ActiveSheet.ChartObjects("Bristol")
        .Left = 0
        .Top = 200
        .Height = 200
        .Width = 250
    End With
    ActiveSheet.Cells(1).Select
    End Sub
Sub CreatePivotAnalysis()
    Dim pvCsh As PivotCache
    Set pvCsh = ActiveWorkbook.PivotCaches.Add(SourceType:=xlDatabase, SourceData:=Sheets("SurveyData").Range("A1:F101"))

    ' Delete Summary sheet if one exists
    On Error Resume Next
    Application.DisplayAlerts = False
    Sheets("Summary").Delete
    On Error GoTo 0

    ' Cancel ignoring errors
    Application.DisplayAlerts = True
    Sheets.Add.Name = "Summary"
    ActiveSheet.PivotTables.Add PivotCache:=pvCsh, TableDestination:=Range("A1"), TableName:="Convenient"
    With ActiveSheet.PivotTables("Convenient")
        .PivotFields("Sex").Orientation = xlColumnField
        .PivotFields("Store locations are convenient").Orientation = xlRowField
        .PivotFields("Store locations are convenient").Orientation = xlDataField
        .DataFields(1).Function = xlCount
    End With
    ActiveSheet.PivotTables.Add PivotCache:=pvCsh, TableDestination:=Range("A12"), TableName:="Employees"
    With ActiveSheet.PivotTables("Employees")
        .PivotFields("Sex").Orientation = xlColumnField
        .PivotFields("Employees are friendly").Orientation = xlRowField
        .PivotFields("Employees are friendly").Orientation = xlDataField
        .DataFields(1).Function = xlCount
    End With
End Sub
ActiveSheet.PivotTables.Add PivotCache:=pvCsh, TableDestination:=Range("A24"), TableName:="Pricing"
With ActiveSheet.PivotTables("Pricing")
    .PivotFields("Sex").Orientation = xlColumn
    .PivotFields("Pricing is competitive").Orientation = xlRowField
    .PivotFields("Pricing is competitive").Orientation = xlDataField
    .DataFields(1).Function = xlCount
End With
ActiveSheet.PivotTables.Add PivotCache:=pvCsh, TableDestination:=Range("A36"), TableName:="Recommend"
With ActiveSheet.PivotTables("Recommend")
    .PivotFields("Sex").Orientation = xlColumn
    .PivotFields("I would recommend your company").Orientation = xlRowField
    .PivotFields("I would recommend your company").Orientation = xlDataField
    .DataFields(1).Function = xlCount
End With
With Range("A:A")
    .Replace "1", "Strongly Disagree"
    .Replace "2", "Disagree"
    .Replace "3", "Undecided"
    .Replace "4", "Agree"
    .Replace "5", "Strongly Agree"
End With
Call MakeButtons
End Sub

Sub ChangeToPercent()
    For Each PT In ActiveSheet.PivotTables
        With PT.DataFields(1)
            .Calculation = xlPercentOfColumn
            .NumberFormat = "0.00%"
        End With
    Next
End Sub
Sub ChangeToCount()
    For Each PT In ActiveSheet.PivotTables
        With PT.DataFields(1)
            .Calculation = xlNormal
            .Function = xlSum
            .NumberFormat = "General"
        End With
    Next
End Sub

Sub MakeButtons()
    Dim percBut As Button
    Dim countBut As Button
    Set percBut = Sheets("Summary").Buttons.Add(400, 25, 100, 40)
    percBut.OnAction = "ChangeToPercent"
    percBut.Characters.Text = "Change to percent"
    Set countBut = Sheets("Summary").Buttons.Add(400, 90, 100, 40)
    countBut.OnAction = "ChangeToCount"
    countBut.Characters.Text = "Change to count"
End Sub
Automation - communicating with other Office programs (Page 91)

Sub TalkToWord()
Dim WordApp As Object
Set WordApp = CreateObject("Word.Application")
Selection.Copy
With WordApp
 .Visible = True
 .Documents.Add
   With .Selection.Font
    .Bold = True
    .Size = 18
   End With
   With .Selection
    .TypeText "Quarter 1 Sales"
    .ParagraphFormat.Alignment = 1
    .typeparagraph
    .typeparagraph
    .Paste
   End With
   .Quit
End With
End Sub
INDEX

A
ActiveX Controls
  Adding to worksheet ............................42
  check box ........................................42
  combo box ......................................47, 52
  Control toolbox ................................39
  deleting .........................................45
  events ............................................47, 48
  list box .........................................49
  moving and sizing ..............................44, 45
  option buttons ...................................50
  Properties .......................................41
  returning a value ...............................46
  spin button ......................................51
Automation
  adding a reference to object library ........88
  declaring an MS Office object variable ....89
  definition ........................................88
  setting an MS object variable ...............90
C
Charts
  creating in VBA ..................................67
  moving and sizing ..............................68
  multiple .........................................71
  naming ...........................................72
  series adding ....................................73
  series, deleting ..................................74
D
Dialog Boxes
  using Excel dialog boxes in procedures ......21
E
Error Handling
  Err.Number ......................................13
  error descriptions ..............................16
  labels ............................................11
  On Error GoTo ..................................9, 11
  On Error GoTo 0 ................................9, 16
  On Error Resume Next ..........................9, 10
  Resume ...........................................9, 11
  trapping with error numbers ................13
  using interactive message boxes ............12
  using the IF statement .........................7
Error handling, IS... functions
  IsArray .........................................8
  IsDate ......................................88
  IsEmpty .........................................8
  IsError ..........................................8
  IsMissing .......................................8
  IsNull ..........................................8
  IsNumeric .......................................8
  IsObject .......................................8
Event Procedures
  definition .......................................56
Events
  arguments .......................................60
  cancel argument ...............................61
  examples .......................................61
  non object .......................................63
  OnTime .........................................63
  types ............................................56
F
Form Controls
  Definition .......................................39
  Forms toolbar ...................................39
I
IsNumeric .........................................8, 102
M
Message Box
  adding interactivity ............................97
  button types ....................................97
  default buttons ................................98
  icons .............................................97
  interactive use examples ....................99
  responding to ..................................99
P
Pivot Tables
  creating in VBA ..................................78
  moving fields ...................................81
  naming ...........................................80
  overview .......................................77
  pivot cache ....................................78
  refreshing ......................................83
  removing fields ...............................82
R
Run-time error ...................................7
U
Userforms, custom
  adding controls ................................22
  adding events to controls ....................28
  Cancel button ..................................29
  check boxes ....................................31
  combo boxes ....................................30
  control properties .............................26
  control types ..................................22
  creating ........................................21
  displaying ......................................33
  editing controls ...............................24
  form events - initialize ......................32
  grouping controls .............................25
  inserting to a workbook .......................22
  list boxes .....................................30
  naming ..........................................27
  OK button ......................................29
<table>
<thead>
<tr>
<th></th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>options buttons</td>
<td>31</td>
</tr>
<tr>
<td>scroll bars</td>
<td>31</td>
</tr>
<tr>
<td>spin buttons</td>
<td>31</td>
</tr>
<tr>
<td>text boxes</td>
<td>30</td>
</tr>
</tbody>
</table>

**W**

Worksheet controls

ActiveX ................................ 38
definition ................................ 39