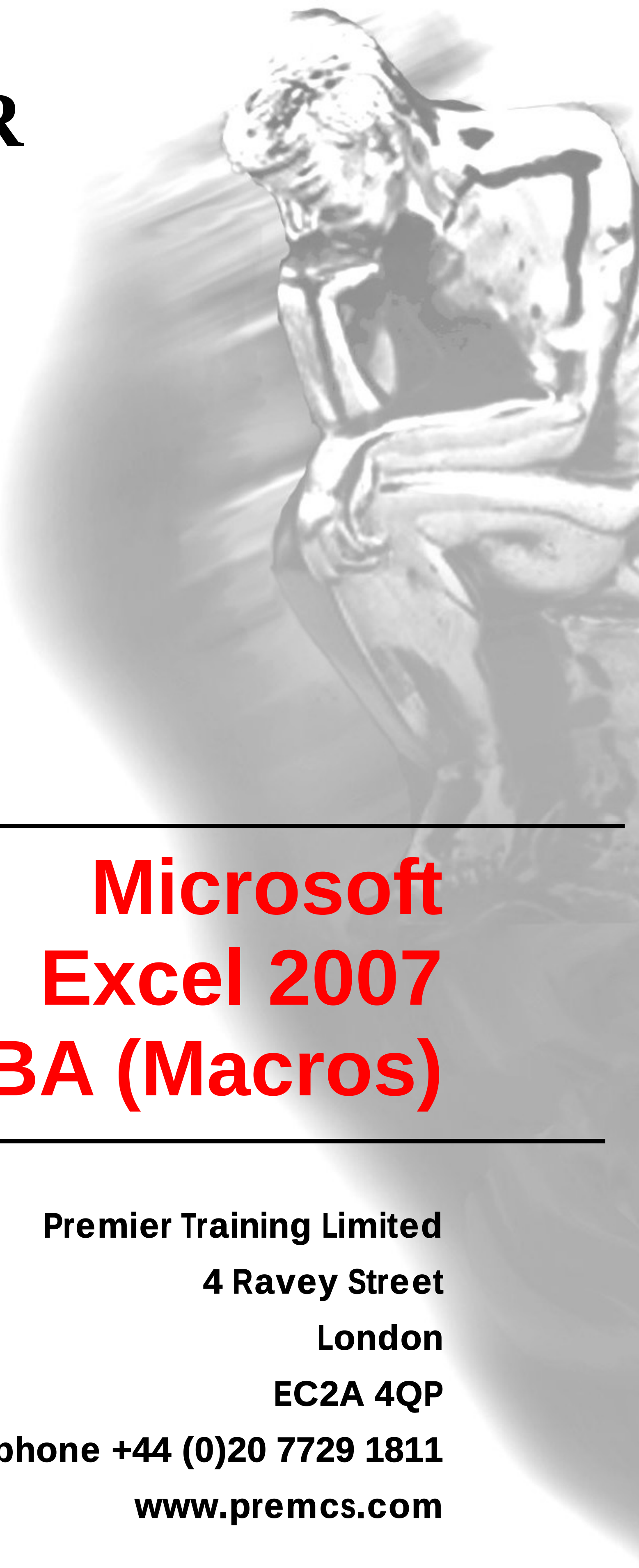
PREMIER



Microsoft Excel 2007 VBA (Macros)

Premier Training Limited

4 Ravey Street

London

EC2A 4QP

Telephone +44 (0)20 7729 1811

www.premcs.com

TABLE OF CONTENTS

INTRODUCTION

This manual is designed to provide information required when using *Excel 2007*. This documentation acts as a reference guide to the course and does not replace the documentation provided with the software.

The documentation is split up into modules. Within each module is an exercise and pages for notes. There is a reference index at the back to help you to refer to subjects as required.

These notes are to be used during the training course and in conjunction with the *Excel 2007* reference manual. *Premier Computer Solutions* holds the copyright to this documentation. Under the copyright laws, the documentation may not be copied, photocopied, reproduced or translated, or reduced to any electronic medium or machine readable form, in whole or in part, unless the prior consent of *Premier Computer Solutions* is obtained.

Module 1 - Overview of Visual Basic

A macro is a sequence of instructions that can be automatically executed in order to automate frequent or complicated tasks. Macros are written in a programming language called Visual Basic and can be created by recording a task or by writing the Visual Basic program or by a combination of the two.

Macros can be added to menus, assigned to keys or buttons or made to run automatically.

Objects and Hierarchies

When developing applications in Excel, it's helpful to think in terms of objects, excel elements that you can manipulate through a macro.

On the corresponding pages you will see a collection of objects available in Excel. The hierarchy comprises Excel's object model.

Excel's object model, exposes very powerful data analysis objects, such as worksheets, charts, pivot tables, scenarios, and financial functions amongst others. Objects act as containers for other objects such as Workbook and Command Bar objects. A worksheet object can contain objects such as Range objects and so on.

Methods

Objects have Methods, and a method is an action that is performed with the object. To clear the contents of a cell, we can use the method ClearContents, which is a method for a range object.

Another key concept of Excel VBA is collections. A collection is a group of objects of the same class. For example Worksheets is a collection of all Worksheet objects in a Workbook object.

Properties

Every object has properties. An example would be that a range object has a property called Value. You can display the value or set the property value to a specific value.

The following code uses the Msgbox function to display a value;

```
Sub ShowValue()

Msgbox Worksheets(“Sheet1”).Range(“A1”).Value

End Sub
```

Microsoft Excel Objects

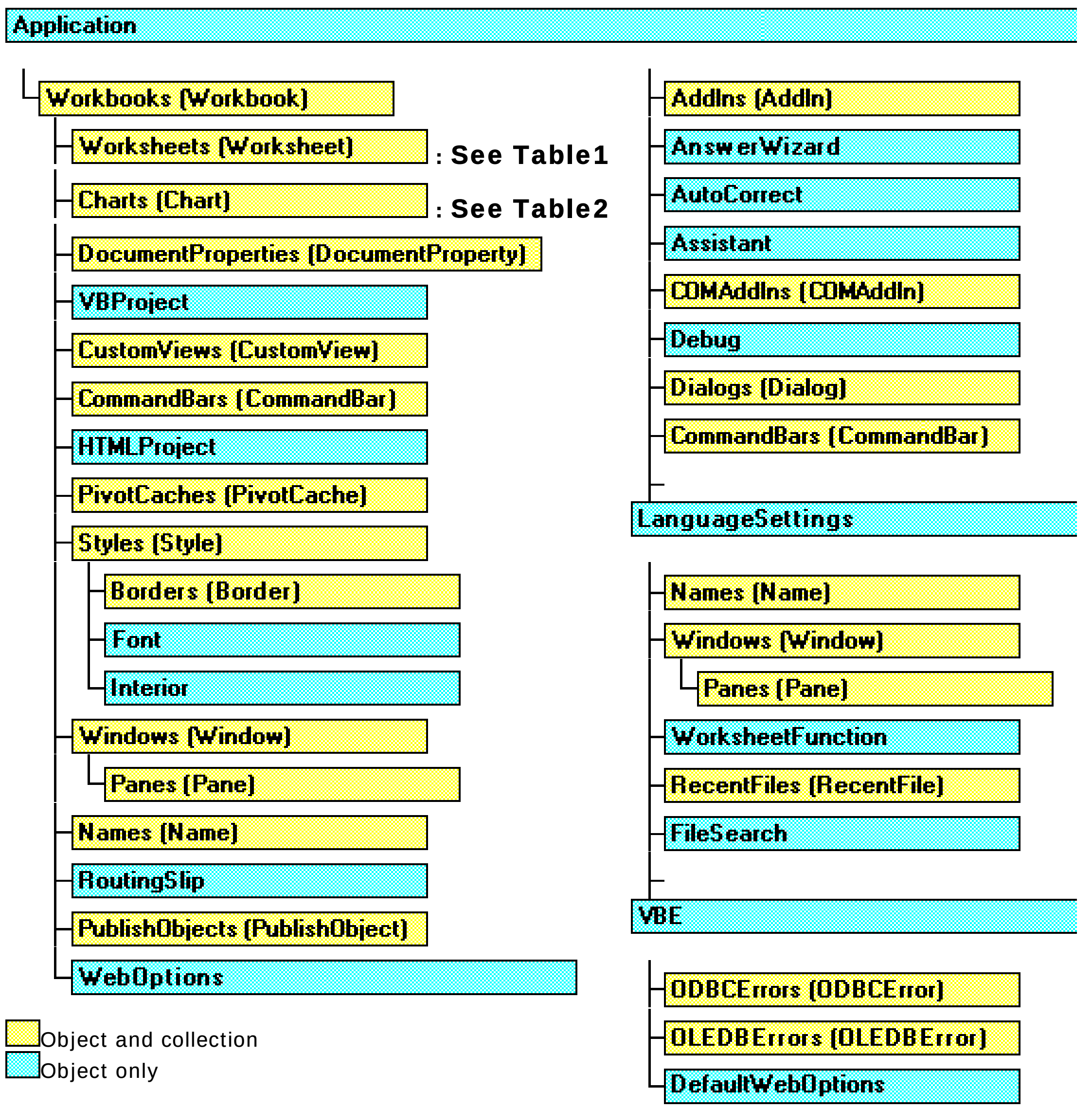


Table 1: Microsoft Excel Objects (Worksheet)

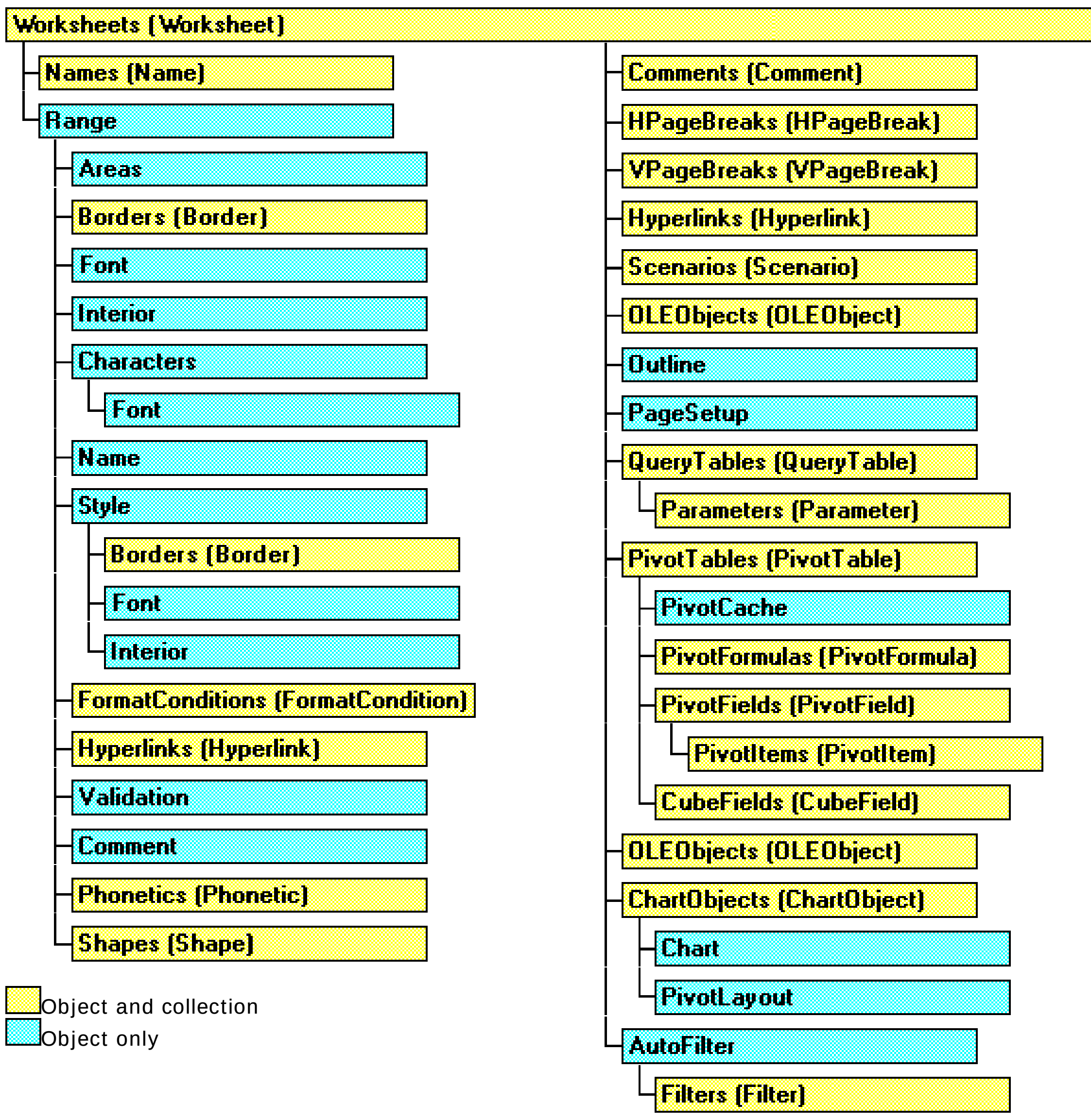
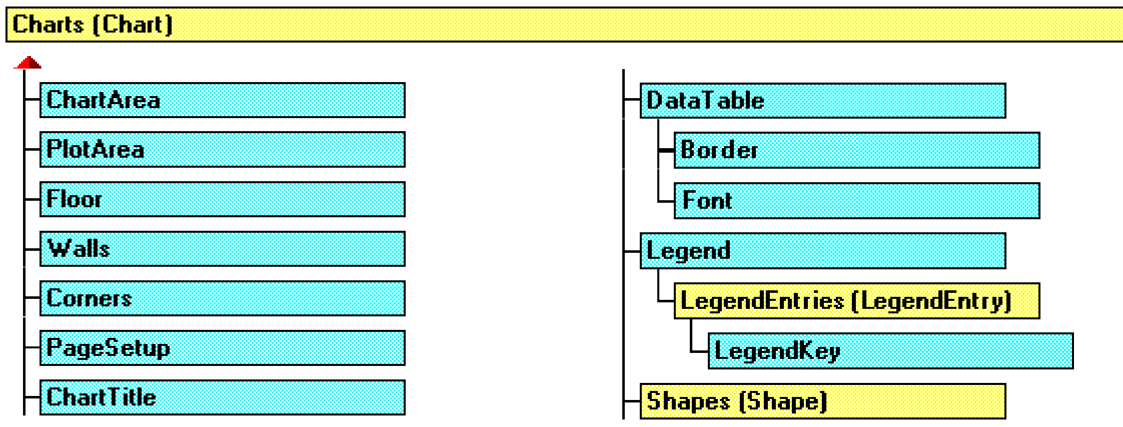
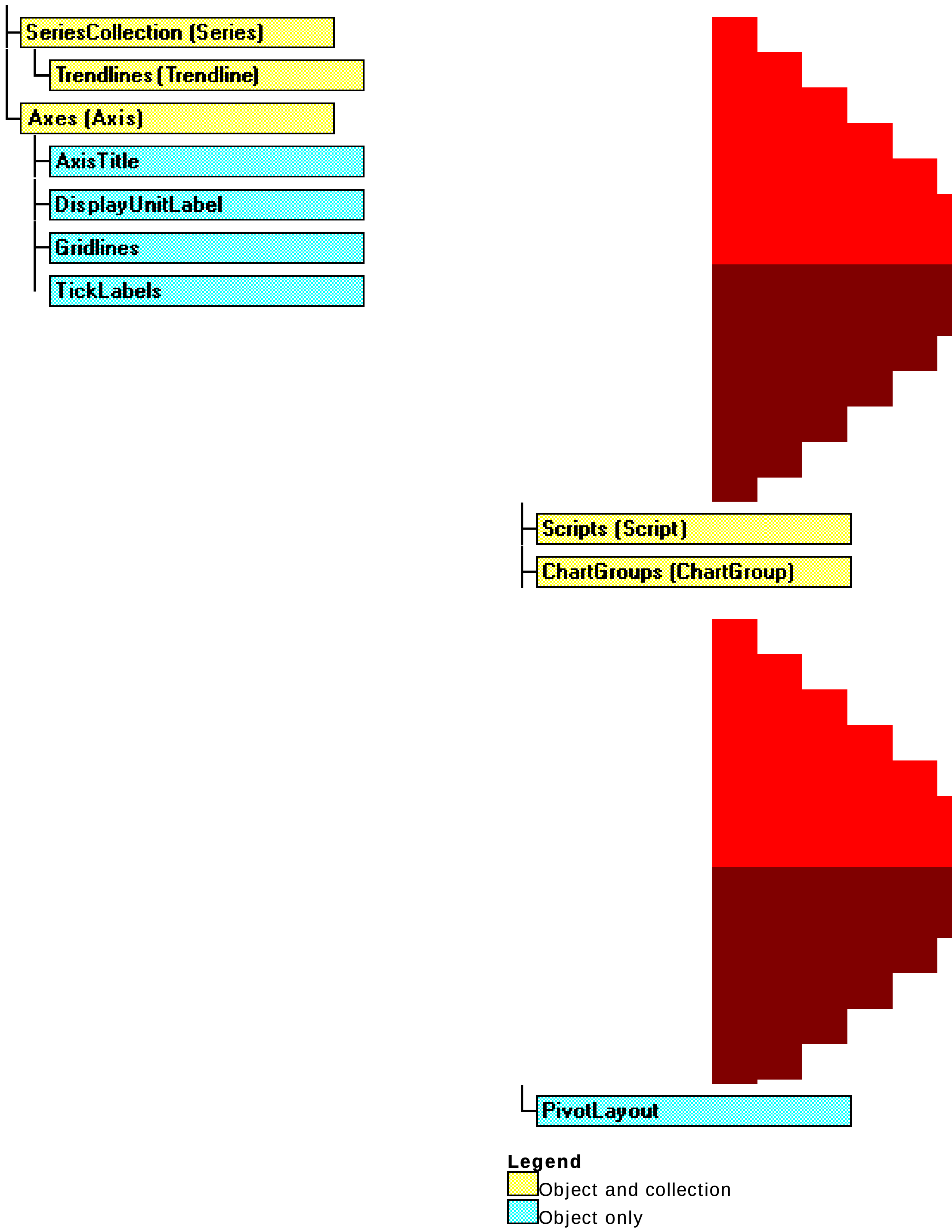


Table 2: Microsoft Excel Objects (Charts)




SeriesCollection (Series)
Trendlines (Trendline)
Axes (Axis)
AxisTitle
DisplayUnitLabel
Gridlines
TickLabels
Scripts (Script)
ChartGroups (ChartGroup)
PivotLayout
Legend
Object and collection
Object only

Objects, Properties and Methods

An **object i**s something that is controlled by Visual Basic, for example a worksheet or a range of cells. An object is controlled using **properties** and **methods**.

A **property** is a characteristic of the object that can be set in a certain way. For example a worksheet has the visible property indicating whether or not the worksheet is visible; a range of cells has the height property indicating the height of the rows in the range.

A **method** is an action that can be performed by the object. For example, a worksheet can recalculate formulae; a range of cells has the copy method.

Properties have values that are set or returned. Methods are actions that an object can perform. Most properties have a single value, methods can take one or more arguments and may return a value.

One of the following is usually performed by statements in a procedure:

Set the value of one of the properties of an object.

Return the value of one of the properties of an object.

Perform a task on an object by using a method of the object.

For example, a cell can be tested to see if it is empty by returning the **value property**. If it is not empty the cell can be cleared using the **ClearContents method**. A new formula can entered into the cell by setting the **value property**.

```
Sub TestEmpty()
    If IsEmpty(ActiveCell.Value) Then
            ActiveCell.Value = 10
    Else
            ActiveCell.ClearContents
    End If
End Sub
```

Controlling Objects with their Properties

An object is changed by changing it’s property. The current property value can be tested and a new value for the property can be set.

A property is usually one of the following:

A numeric value

A character string

A True or False value

A property is referred to in a statement as:

Object.Property

Setting a Property Value

To set a new value for the property, use the following statement:

*Object***.***Property* **=** *expression*

For example to set properties for the current active cell:

```
ActiveCell.Rowheight = 14
ActiveCell.Value = "Annual Totals"
ActiveCell.Locked = True
```

Returning a Property Value

The current property of a value can be returned and is usually assigned to a variable for further testing using the following statement:

variable = Object.Property

For example:

row_height = ActiveCell.RowHeight

A property can be set using the value of another property without assigning to a variable in between.

For example, to assign the value in the current active cell into cell C1 on the current worksheet:

Cells(1, 3).Value = ActiveCell.Value

Performing Actions with Methods

A method performs an action on an object. For example, **Clear** is a method that can be applied to clear the contents of a range of cells. Certain methods can also return values, for example the CheckSpelling method which performs a spell check on a range of cells, also returns the value True or False depending on whether the text is spelt correctly.

Using Methods

As well as possibly returning values a method may require certain arguments, this is referred to as the method *taking* arguments. The syntax of the statement depends on whether a method takes arguments:

A method that doesn’t take arguments is written:

*Object***.***Method*

For example, to justify a range of cells called Costs:

Costs.Justify

A method that does take arguments is written:

Object.Method argument list

For example to name a selection of cells using the labels in the top cell of the selection:

Selection.CreateNames True, False, False, False

As the arguments are optional this could also be written as:

Selection.CreateNames True

If a method returns a result and the result is to be saved, then the arguments that the method takes must be enclosed in parenthesis.

For example to save the result of the InputBox method in a variable called SalesData:

```
SalesData =InputBox ("Enter the sales data")
```

Using Named arguments

Some methods take several arguments some of which are optional and can be omitted. For example the CreateNames method has four arguments most of which can be omitted in normal use.

The method can be written to just include the relevant arguments but this is not always very clear:

```
Selection.CreateNames True
```

It may be preferable to write the method statement using named arguments:

```
Selection.CreateNames Top:=True
```

Named arguments allow the method to be written with any of the required arguments in any order. The value is assigned to the named argument using the := operator. They make the code much easier to understand and are less prone to error than an argument list where the arguments must be in the correct order and may be accidentally omitted.

Workbooks and Sheets

Activating a Workbook

Activating a workbook using the Activate method puts the workbook in the active window. The following procedure activates the open workbook named “MyBook.xls”.

```
Workbooks("MyBook.xls").Activate
```

Activating a Worksheet

The following example activates Sheet1 in the workbook "MyBook.xls":

```
Workbooks("MyBook.xls").Worksheets("Sheet1").Activate
```

The following example selects cells A1:C3 on Sheet1 and then makes cell B2 the active cell:

```
Worksheets("Sheet1").Activate
Range("A1:C3").Select
Range("B2").Activate
```

The following example inserts "Hello" in the cell B1 on the first sheet:

```
Worksheets(1).Range("B1").Value = "Hello"
```

ThisWorkBook

The **ThisWorkBook** property refers to the workbook containing the macro code. **ThisWorkbook** is the only way to refer to an add-in workbook from inside the add-in itself. The **ActiveWorkbook** property does not return the add-in workbook (it returns the workbook calling the add-in), and the Workbooks method may fail because the workbook name probably changed when the add-in was created. **ThisWorkbook** always returns the workbook where the code is running.

For example, use the following code to activate a dialog sheet stored in the add-in workbook:

ThisWorkbook.UserForm1.Show

The following example closes the workbook that contains the example code:

```
ThisWorkbook.Close
```

Performing Multiple actions on an Object

It may be necessary for a procedure to perform several different actions on the same object. For example, the following may need to be performed to the active cell:

```
ActiveCell.Formula = "=NOW()"
ActiveCell.NumberFormat = "dd/mm/yy"
ActiveCell.Font.Name = "Arial"
ActiveCell.Font.Bold = True
ActiveCell.Font.Size = 14
```

Because all the statements refer to the same object, it is possible to use the **With** statement to indicate that a series of statements are all to be performed to one object:

```
With ActiveCell
    .Formula = "=NOW()"
    .NumberFormat = "dd/mm/yy"
    .Font.Name = "Arial"
    .Font.Bold = True
    .Font.Size = 14
End With
```

The With statement makes the code easier to write, easier to read and is more efficient when running the macro.

With statements can be nested and the above example could be written as:

```
With ActiveCell
    .Formula = "=NOW()"
    .NumberFormat = "dd/mm/yy"
    With .Font
        .Name = "Arial"
        .Bold = True
        .Size = 14
    End With
End With
```

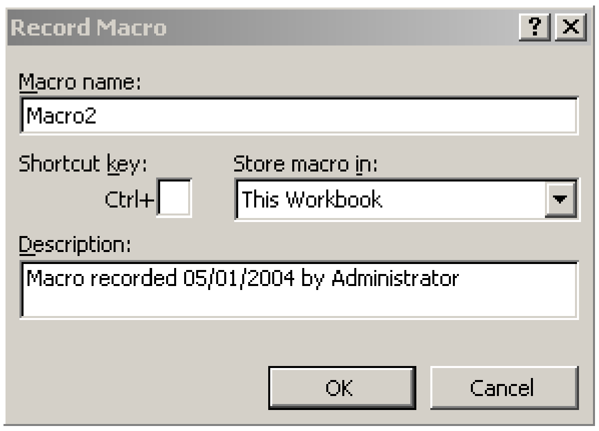
Module 2 - Introduction to VBA

Recording a Macro

A macro can be easily created by instructing Excel to record a series of routine actions. This is very useful for repetitive tasks.

- To record a macro:

1. Choose **Tools** ⇨ **Macro** ⇨ **Record Ne.w Macro**.



Record Macro

Macro name: Macro2

Shortcut key: Ctrl+

Store macro in: This Workbook

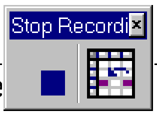
Description: Macro recorded 05/01/2004 by Administrator

2 In the Macro name text box, type a name for the macro.

   The macro name can contain letters, numbers and underscores; it must begin with a letter. The name cannot contain spaces or punctuation marks.

   Type a brief description of the macro in the Description box, Excel creates it's own default description.

3. To run the macro with a shortcut key type a letter in the Shortcut key box. The shortcut will be assigned to [CTRL]+[letter]. Hold [SHIFT] while typing the letter in the Shortcut key box to assign the shortcut to [CTRL]+[SHIFT]+[letter].
4. Choose the **OK** button to start recording the macro.The Stop Recording toolbar appears on the screen, and the word Recording appears in the status bar.
5. Perform the actions that are to be recorded.
6. Click the **Stop Recording** button from the Stop



Recording toolbar or choose **Tools** ⇨ **Macro** ⇨ **Stop Recording,** when the macro is finished.

Running a Macro

To run a macro:

If a shortcut key was assigned to the macro, then press the shortcut key.

Otherwise:

1. Choose **Tools** ⇨ **Macros** or press [ALT]+[F8].

2. Type the macro name or select it from the list.

3. Choose the **Run** button.

A macro can be assigned to a menu item, a button or another graphic object providing a simple and convenient way of running the macro.

ASSIGNING MACROS TO BUTTONS AND OBJECTS

Assigning a Macro to a Button on a Sheet

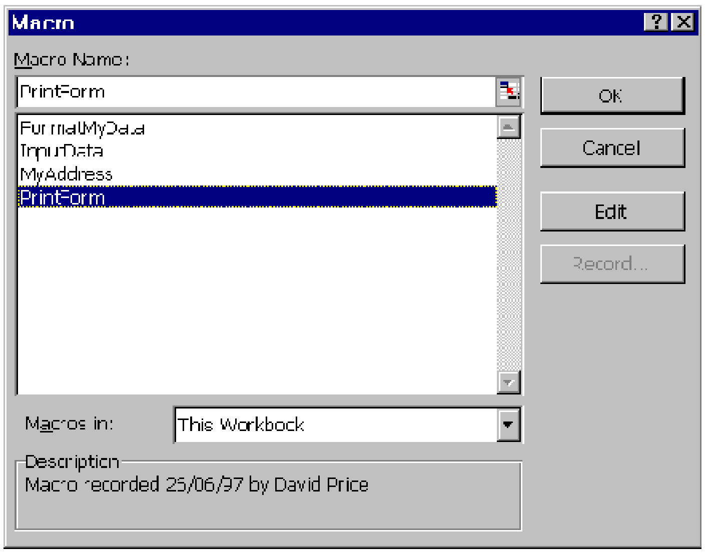
A button can be created on a worksheet and a macro assigned to it. The macro will be available whenever the workbook is open.

To create a button on a sheet:

1. Choose the Button tool on the Forms toolbar. The mouse pointer is replaced by a cross-hair.

2. Move the cross-hair to where the corner of the button should be.

3. Drag until the button is the correct size and shape and release the mouse button. The Assign Macro dialog box appears:



Macro recorded 25/06/97 by David Price

To assign an existing macro to the button:

1. Type the macro name or select it from the Macro Name list.
2. Choose the **OK** button.
3. To assign a new macro to the button:
4. Choose the **Record** button and follow the steps for recording a new macro.
5. To run the macro:
6. Click the button to which the macro is assigned.
7. To alter the text or appearance of a button:
8. Select the button by holding down [CTRL] and clicking the left mouse button.
9. Alter the button text by typing new text.
10. Change the font or alignment of the button text by choosing **Format** ⇨ **Control** from the shortcut menu.

Assigning a Macro to a Drawn Object on a Sheet

Macros can be assigned to any worksheet object by choosing **Assign Macro** from the object’s shortcut menu.

Assigning a Macro to a Button on a Toolbar

If a macro is assigned to a button on a toolbar, the macro is available at any time and for every sheet in the workbook, provided the toolbar is displayed. The workbook containing the macro is automatically opened, if it is not already open, when the button is clicked from another workbook.

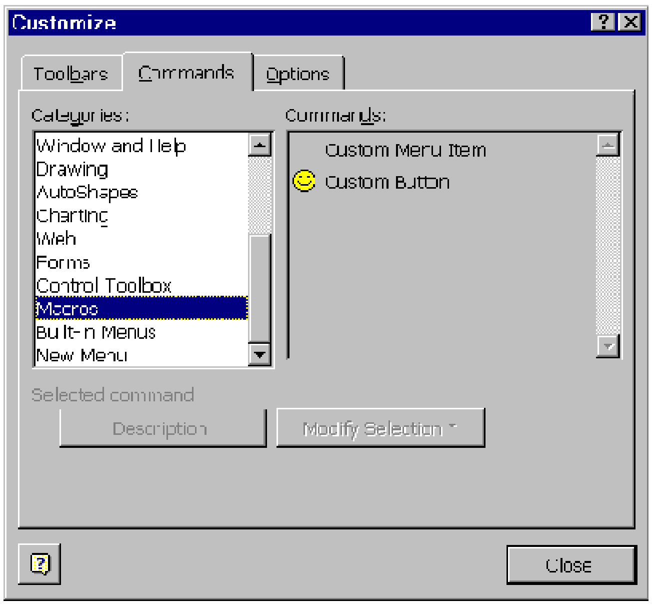
A macro is usually assigned to an unused custom button, but can be assigned to a built in button overriding the button's normal function.

Customising a Toolbar

Change, or customise, a toolbar so it contains buttons for most often used commands or macros.

To add a button to the toolbar:

1. Choose the Commands tab of **Tools** ⇨ **Customize**.



2. From the Categories list box, select Macros.
3. Drag the Custom Button from the Commands list box over the toolbar and then release the mouse.
4. To select a different button image for the new button, right-click the new button and choose **Change Button Image.** Select a button image.
5. Choose the **Close** button.

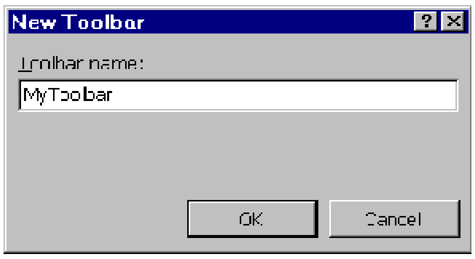
Creating a Toolbar

Buttons can only be added to existing toolbars. It may be necessary to create a new toolbar before adding a button to it.

To create a new toolbar:

1. Choose the Toolbars tab of **Tools** ⇨ **Customize**.

2. Click the **New** button and enter a name for the new toolbar.



MyToolbar

3. Choose the **OK** button.

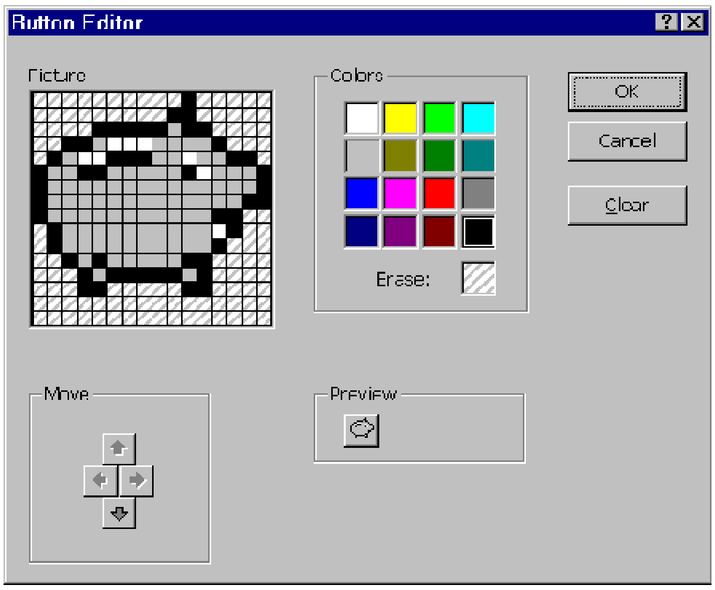
Button Image Editor

Excel is supplied with a small selection of "spare" button images. New button images can be created by editing the existing images or by drawing new images.

To edit a button image:

1. Choose **Tools** ⇨ **Customize**.

2. Right-click the button image to be edited and choose **Edit Button Image**.



3. To edit the existing image, select a colour and fill the pixels of the image.

4. To draw a new button image, choose the **Clear** button and then follow step 3.

Images can also be copied and pasted from other buttons or graphics applications.

Changing a Button's ScreenTip

By default, a new custom button is given the name "&Custom Button". The name of the button appears as the button's ScreenTip when the mouse is positioned over it.

To change a button's ScreenTip:

1. Choose **Tools** ⇨ **Customize**.

2. Right-click the button image to be edited and type the button's name in the **Name** box.

Relative References

If the relative reference option is not chosen the macro records the actual cell references and will always operate on those fixed cells, regardless of which cell is active when the macro is initiated. The following macro selects the range B3 to C5:

```
Sub SelectAbsolute()
    Range("B3:C5").Select
```

```
End Sub
```

If a macro is recorded using relative references, cell references are recorded relative to the previously active cell. The macro then operates on cells relative to the cell that is active when the macro is initiated. The following macro selects a single cell, one row beneath the starting cell:

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

- To set the reference type:

When recording a macro, click the **Relative References** button on the Stop Recording toolbar. By default, Excel records in absolute mode. Recording can be switched from absolute to relative, and vice versa, during the same recording process.

See also The Range Object page **Error! Bookmark not defined.**.

Editing A Macro

You can use ALT + F11 to enter the Visual Basic Code Editor…

Entering and editing the text in a VBA module works just as you would expect. You can select text and copy it as you would in any Microsoft Application.

A single instruction can be as long as you require it to be, however for readability, it's easier to read if you break it onto two or more lines. To do this, end the line with a space followed with and underscore, continuing the code on the next line e.g.

```
Msgbox ("The name entered into the Cell A1 on Worksheet 1 _

was " & MyName)
```

As you enter code in the VB editor, you'll notice that Excel makes some adjustments to your text, for example, if you omit the space before or after the equals sign (=), Excel enters it for you. Also various words in your code will become Title case. This is normal and will assist you in your code later on.

Auto Indent Option (Tools-Options)

The Auto Indent Option setting determines whether VBA automatically indents each new line of code by the same amount as the previous line. You can specify the number of characters, the default is four.

Don’t use the space bar to indent your code, use the Tab key. This also works if you select more that one line of code.

Like Excel the VB Editor has many levels of Undo and Redo.

Exercise 1

1. Record a macro named SystemDate to enter the system date in the active cell, format it to a suitable date format, change the font to Arial 10 and AutoFit the column width.

2. Edit the SystemDate macro to include better use of the **With** statement.

Exercise 2

1. Record the command **Edit ⇨ Clear ⇨ Formats** as a macro named “ClearFormat” to clear the formatting from a selection of cells.

   Assign the macro to the shortcut key combination [CTRL]+[SHIFT]+[f].

   Assign the macro to a custom button on a customised toolbar.

   Assign the macro to a button on a worksheet.

2. Record a macro named “Address” to enter the first three lines of an address in consecutive cells in a column. Record the macro using relative cell references.

3. Save the workbook containing the macros as EXERCISE 1.XLS.

Answers to Exercise 1

Question 1

```
Sub SystemDate()
    ActiveCell.FormulaR1C1 = "=NOW()"
    With Selection.Font
        .Name = "Arial"
        .FontStyle = "Regular"
        .Size = 10
        .Strikethrough = False
        .Superscript = False
        .Subscript = False
        .OutlineFont = False
        .Shadow = False
        .Underline = xlNone
        .ColorIndex = xlAutomatic
    End With
    Selection.NumberFormat = "dd-mmm-yy"
    Selection.EntireColumn.AutoFit
End Sub
```

Question 2

```
Sub SystemDate()
    ActiveCell.Formula = "=NOW()"
    With Selection
        With .Font
            .Name = "Arial"
            .Size = 10
        End With
        .NumberFormat = "dd-mmm-yy"
        .EntireColumn.AutoFit
    End With
End Sub
```

Answers to Exercise 2

Question 1

```
Sub ClearFormat()
      Selection.ClearFormats
End Sub
```

Question 2

```
Sub Address()
      ActiveCell.FormulaR1C1 = "XYZ Company"
      ActiveCell.Offset(1, 0).Range("A1").Select
      ActiveCell.FormulaR1C1 = "Any Street"
      ActiveCell.Offset(1, 0).Range("A1").Select
      ActiveCell.FormulaR1C1 = "Any Town"
End Sub
```

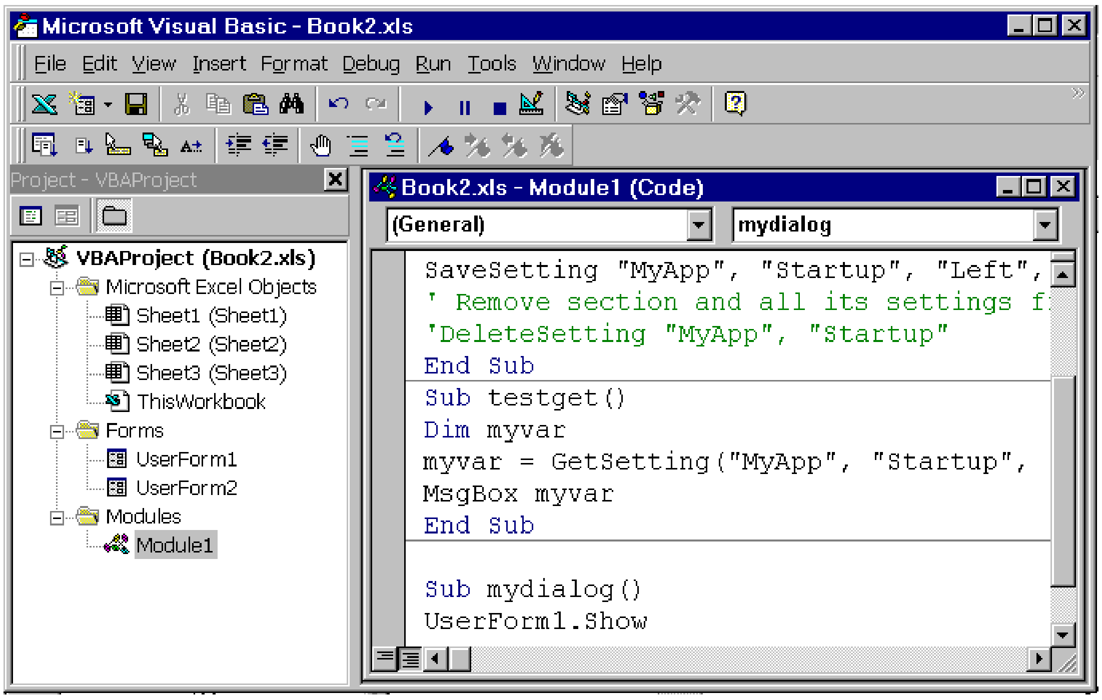
Module 3 - Overview of Visual Basic Editor

Visual Basic Editor

Excel is supplied with the Visual Basic Editor for modifying and creating macros and for creating user defined forms.

The Visual Basic Editor is common to all the Microsoft Office 97 applications.

To start with, the Visual Basic Editor contains two important windows: the Project Explorer and the Code window.



```
SaveSetting "MyApp", "Startup", "Left",
' Remove section and all its settings f
'DeleteSetting "MyApp", "Startup"
End Sub
Sub testget()
Dim myvar
myvar = GetSetting("MyApp", "Startup",
MsgBox myvar
End Sub

Sub mydialog()
UserForm1.Show
```

Creating Modules

Standard Code Modules, also called simply Code Modules or just Modules, are where you put most of your VBA code. Your basic macros and your custom function (User Defined Functions) should be in these modules.

Your workbook's VBA Project can contain as many standard code modules as you want. This makes it easy to split your procedure into different modules for organization and ease of maintenance.

Workbook And Sheet Modules are special modules tied directly to the Workbook object and to each Sheet object. The module for the workbook is called ThisWorkbook, and each Sheet module has the same name as the sheet that it is part of. These modules should contain the event procedures for the object, and that's all.

User Form Modules are part of the UserForm object, and contain the event procedures for the controls on that form. For example, the Click event for a command button on a UserForm is stored in that UserForm's code module. Like workbook and sheet modules, you should put only event procedures for the UserForm controls in this module.

Class Modules are used to create new objects.

VBA Modules

VBA Modules are stored in an Excel Workbook, but you view or edit a module using the Visual Basic Editor.

A VBA Module consists of Procedures. A Procedure is a unit of code that performs some action. Below is an example of a Sub Procedure,

```
Sub Example()

    Answer = 1+1

    MgsBox “The result is “ & Answer

End Sub
```

A VBA module can also have Function procedures.

Function Procedures

Introduction

There are in excess of 300 functions available in formulae on an Excel worksheet, such as SUM, IF and AVERAGE. These functions can be entered into cells directly or via the Function Wizard.

Visual Basic also contains functions. Some Visual Basic functions are similar to worksheet functions such as, NOW and YEAR, others are standard to all forms of the BASIC language and some are unique to Visual Basic.

Despite the number and complexity of functions available there is often a need to create others. A user-defined function can be used on a worksheet or in a module. A user-defined function can be thought of as a “stored formula”.

User-Defined Functions

Function procedures return a value. A function procedure is enclosed by Function and End Function statements and must contain a statement that will return a value to the place where the function was called. Functions usually have data passed to them, this is specified in brackets after the function name.

```
Function Function1 (variable list)
    Commands
    Function1 = Expression
End Function
```

A function procedure can be used as a user-defined function provided it does not perform actions that alter the Excel environment for example inserting, deleting or formatting cells; moving, adding or deleting sheets. Functions can only return values, they can not perform actions with objects.

User-defined functions allow the user to type the procedure name directly into a cell on the worksheet, followed by the required data enclosed in brackets and separated by commas in the order expected by the function procedure.

In the following example, the function *Percentage* is expecting two variables to be passed. It uses the variables to calculate the percentage and the result is passed back:

```
Function Percentage (Amount, Percent)
    ‘Increase amount by percent
    Percentage = (Amount*Percent/100)+Amount   ‘Returns
the result
End Function
```

To use the function in the worksheet, type the function name followed by the data that is to go into the variables enclosed in parenthesis:

```
=Percentage (150,17.5)
=Percentage (B1,B2)
```

In the following example, the function *Age* is expecting one variable to be passed. It uses the variable to calculate the age and the result is passed back:

```
Function Age(DOB)

    If Month(DOB) > Month(Now) Then
            Age = Year(Now) - Year(DOB) - 1
    ElseIf Month(DOB) < Month(Now) Then
            Age = Year(Now) - Year(DOB)
    ElseIf Day(DOB) <= Day(Now) Then
            Age = Year(Now) - Year(DOB)
    Else
            Age = Year(Now) - Year(DOB) - 1
    End If
End Function
```

Function Wizard

Although a user-defined function can be used on the worksheet by typing the function name and variable(s) in brackets it is often better to use the Function Wizard.

Excel automatically adds user-defined functions to a User Defined category of the Function Wizard.

Excel Functions

In the Age user-defined function example above, the Excel functions Month(), Now and Year() were used. Not all worksheet functions are applicable to modules. However, any worksheet function can be used in a module if it is written "*Application*.FunctionName".

Project Window

The Project Explorer displays a hierarchical list of the projects and all of the items contained in and referenced by each of the projects.

- VBAProject

The name of an open workbook.

- Microsoft Excel Objects

The worksheets and chartsheets contained in the VBAProject. The event procedures (see page **Error! Bookmark not defined.**) for each sheet and the workbook are stored here.

- Forms

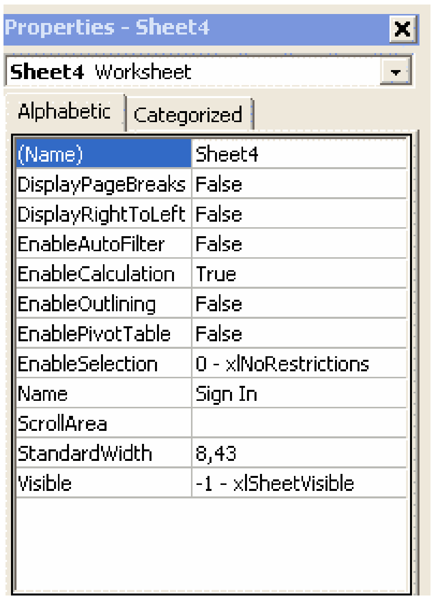
Any user defined forms.

- Modules

Recorded and written procedures.

Properties Window

This is the window in the Visual Basic programming environment that describes the physical properties of each object on the form, and allows the programmer to change those properties.



(Name)	Sheet4
DisplayPageBreaks	False
DisplayRightToLeft	False
EnableAutoFilter	False
EnableCalculation	True
EnableOutlining	False
EnablePivotTable	False
EnableSelection	0 - xlNoRestrictions
Name	Sign In
ScrollArea	
StandardWidth	8,43
Visible	-1 - xlSheetVisible

Code Window

Use the Code window to write, display and edit Visual Basic code. Open as many Code windows as there are modules; view the code in different forms or modules and copy and paste between them.

Open a Code window from the Project window, by double-clicking a form or module.

Drag selected text to:

A different location in the current Code window.

Another Code window.

The Immediate and Watch windows.

The Recycle Bin.

- Object Box

Displays the name of the selected object. Click the arrow to the right of the list box to display a list of all objects associated with the form.

- Procedures/Events Box

Lists all the events recognised by Visual Basic for a form or control displayed in the Object box. When an event is selected, the event procedure associated with that event name is displayed in the Code window.

If (General) is displayed in the Object box, the Procedure box lists any declarations and all of the general procedures that have been created for the form. If editing module code, the Procedure box lists all of the general procedures in the module. In either case, the procedure selected in the Procedure box is displayed in the Code window.

All the procedures in a module appear in a single, scrollable list that is sorted alphabetically by name. Selecting a procedure using the drop down list boxes at the top of the Code window moves the cursor to the first line of code in the procedure selected.

- Split Bar

Dragging the Split bar down, splits the Code window into two horizontal panes, each of which scrolls separately. View different parts of code at the same time. The information that appears in the Object box and Procedures/Events box applies to the code in the pane that has the focus. Dragging the bar to the top or the bottom of the window or double-clicking the bar closes a pane.

- Margin Indicator Bar

A grey area on the left side of the Code window where margin indicators are displayed. Margin indicators provide visual cues during code editing.

Procedures

A procedure is a series of statements of Visual Basic code stored in a module of a Visual Basic project. When a macro is recorded Excel writes a procedure. The code can be modified and special Visual Basic commands can be included to allow user interaction, conditional testing, looping and other options.

Each procedure is identified by a name that is written at the top of the procedure.

There are two types of procedures, Sub procedures and Function procedures.

Sub Procedures

Sub procedures perform actions. A macro is a recorded Sub procedure.

A Sub procedure is enclosed by Sub and End Sub statements, the Sub statement contains the macro name:

```
Sub Macro1 ()
    Commands
End Sub
```

The macro name is followed by brackets. The brackets may contain variable data called **arguments**.

Inserting Comments

It is very important to put explanatory comments into a macro as it will make the macro easier to read and debug.

Comment text can be placed in a line on its own or following a macro statement.

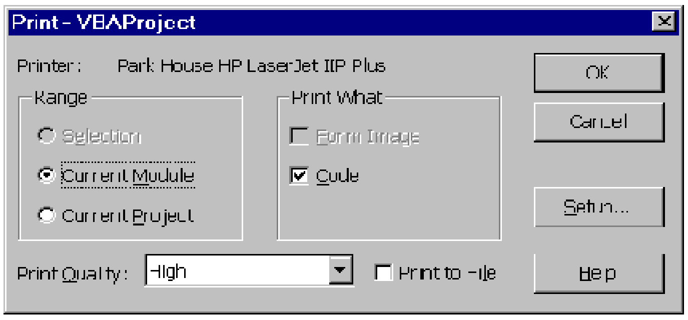
Comment text is prefixed by an apostrophe. The apostrophe tells Visual Basic to ignore the text that follows it and to continue execution with the statement on the next line.

Comment text normally appears in a green font, but this style can be altered.

Printing a Visual Basic Module

- To print the contents of a Visual Basic module:

1. Display the Visual Basic module that is to be printed.
2. Choose **File ⇨ Print**.
3. Set the required print options.


Print - VBAProject

Printer: Park House HP LaserJet IIP Plus

Range: Selection; [✓ Current Module]; Current Project

Print What: Form Image; [✓ Code]

Print Quality: High

Print to File

4. Choose the **OK** button.

Exercise 3

1. Open the workbook USER FUNCTIONS.XLS and investigate the Age(), Celsius() and Fahrenheit() functions.

2. Write a function procedure named “YdsToMts” to convert a length in yards to one in metres, where 1 yard equals 0.914 metres.

3. Write a function procedure named “MtsToYds” to convert a length in metres to one in yards, where 1 metre equals 1.094 yards.

4. Resave the workbook (this workbook will be needed again later in the course).

Answers to Exercise 3

Question 2

```
Function YdsToMts(Yards)
      YdsToMts = Yards * 0.914
End Function
```

Question 3

```
Function MtsToYds(Metres)
      MtsToYds = Metres * 1.094
End Function
```

Module 4 - Inputs and Outputs

Selecting Cells, Ranges and Objects

A range object can consist of:

- A cell.
- One or more selections of cells.
- A 3-D range.
- A row or column.

Working with range objects is fundamental to most Excel macros. There are several important methods and properties for controlling a range object.

Cells Method

The Cells method returns a single cell. All cells are numbered on a worksheet from left to right, top to bottom. Hence, A1 is cell number 1 and IV65536 is cell number 16,777,216.

However, because cells always occur in rows and columns a cell can also be specified by its row and column index. Cells(1) can also be referred to as Cells(1,1) and Cells(16777216) can be referred to as Cells(65336,256). Referring to a cell by its row and column index is usually easier than referring to it simply by its position number unless working within a virtual worksheet (see below).

Using the Cells method with no argument returns all the cells on a worksheet.

Range Method

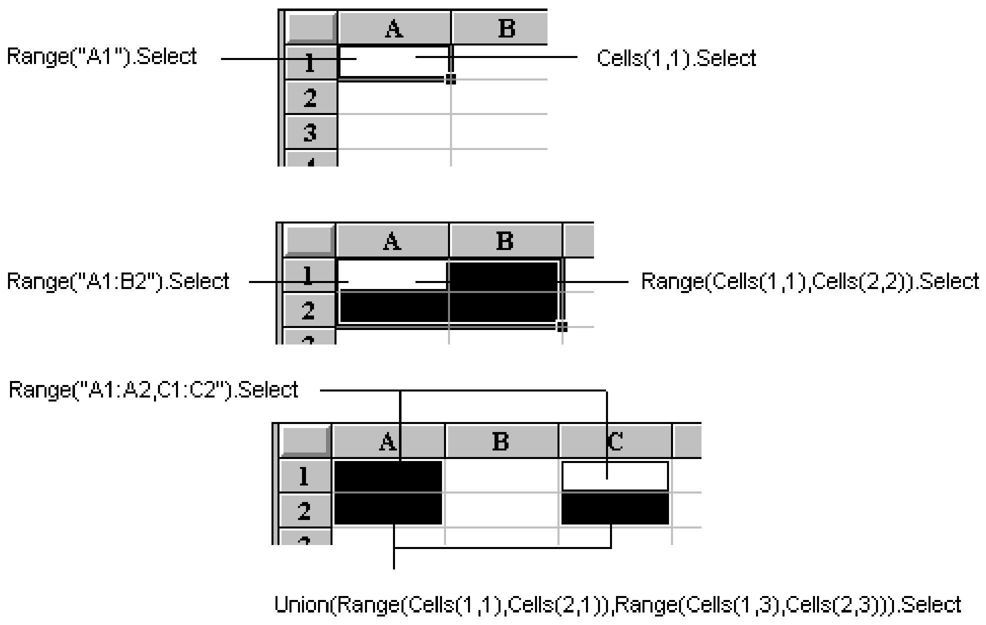
The range method returns a rectangular range or cells.

The range can be specified by a text string that is a cell address or a defined range name or it can be specified with the Cells method.

Selecting Range Objects

Range objects are selected using the Select method:

Range(“A1:B2”).Select
Range(Cells(1,1),Cells(2,2)).Select


Range("A1").Select
Cells(1,1).Select
Range("A1:B2").Select
Range(Cells(1,1),Cells(2,2)).Select
Range("A1:A2,C1:C2").Select
Union(Range(Cells(1,1),Cells(2,1)),Range(Cells(1,3),Cells(2,3))).Select

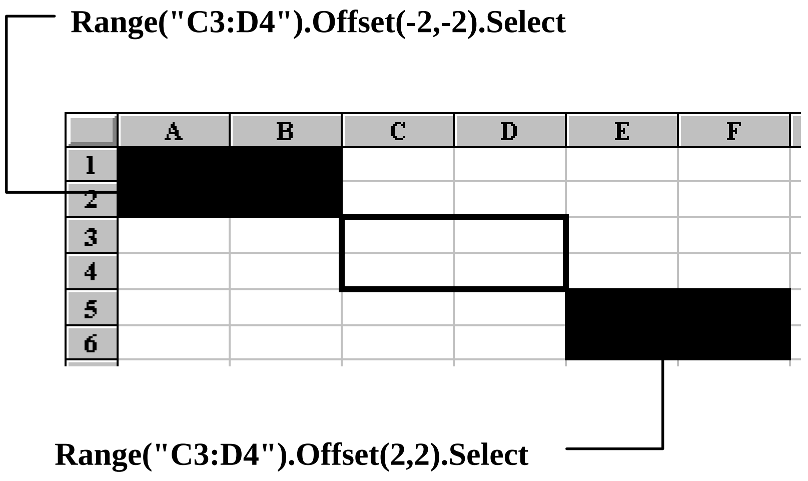
Multiple selections using the Cells method must also use the Union method.

Flexible Ranges

Offset Method

The Offset method takes one Range object and calculates a new Range object from it.

The Offset method has two arguments: the number of rows to move the selection down or up and the number of columns to move the selection right or left.


Range("C3:D4").Offset(-2,-2).Select
A
B
C
D
E
F
1
2
3
4
5
6
Range("C3:D4").Offset(2,2).Select

Resize Method

The Resize method also takes one Range object and calculates a new Range object from it. However, instead of moving the range, it changes the range's size.

The Resize method has two arguments: the number of rows for the new selection and the number of columns for the new selection.

Range("A1:B2").Resize(3,4).Select resize the range to A1:D4

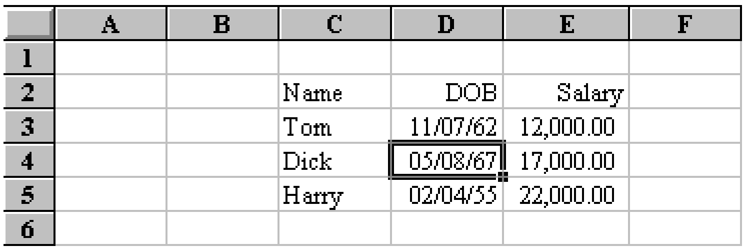
Selection, ActiveCell and Activate Methods

The Selection method returns the currently selected range. The ActiveCell method returns the active cell within the current selection. If the current selection consists of only one cell, then the Selection method and ActiveCell method return the same object.

The Activate method activates a single cell within the current selection.

Row and Column Properties

Returns the number of the first row or column of a range. In the following example, FirstRow would equal 2 and FirstColumn would equal 3 so long as the active cell was in the region C2:E5. CurrentRow would equal 4 and CurrentColumn would equal 4 if the active cell was D4:



	A	B	C	D	E	F
1						
2			Name	DOB	Salary	
3			Tom	11/07/62	12,000.00	
4			Dick	05/08/67	17,000.00	
5			Harry	02/04/55	22,000.00	
6						

```
Sub FirstRowColumn()
    FirstRow = ActiveCell.CurrentRegion.Row
    CurrentRow=ActiveCell.Row
    FirstColumn = ActiveCell.CurrentRegion.Column
    CurrentColumn=ActiveCell.Column
End Sub
```

Rows and Columns Methods

Returns a single row or column or a collection of rows and columns.

Rows(3).Delete Deletes the third row of the worksheet.

Selection.Columns.Count Returns the number or columns in the current selection.

Address Method

Returns the range reference, as a string. The following examples assume the active cell is A1:

ActiveCell.Address	$A$1
ActiveCell.Address(rowAbsolute:=False)	$A1
ActiveCell.Address(referenceStyle:=xlR1C1)	R1C1

Returning to the Starting Cell

If a procedure uses a method to offset or resize a selection it is often useful to be able to return to the cell which was active before the procedure was run. To do this use the Address method of the active cell:

```
Sub Test()
    ReturnCell = ActiveCell.Address

    ...procedure

    Range(ReturnCell).Select
```

End Sub

End Method

Returns a cell (a Range object) at the end of the region. Equivalent to pressing [END]+[↑], [END]+[↓], [END]+[→] or [END]+[←].

The direction to move is one of xlToLeft, xlToRight, xlUp, or xlDown.

e.g. **Range("C3").End(xlUp).Select**

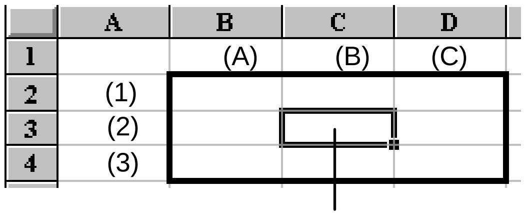
Range("C3").End(xlToRight).Select

Combining Methods

Statement:	**Description**
Selection.Offset(1,0).Select	Moves the current selection down one row.
Selection.Resize(Selection.Rows.Count+1).Select	Increases the current selection by one row.
Selection.Offset(0,3).Resize(4,5).Select	Move the current selection 3 columns to the right and resize it to 4 rows by 5 columns.
Selection.Offset(1,0).Activate	Move the active cell to the next row in the selection.
Range(ActiveCell , _ ActiveCell.End(xlDown).End(xlToRight)).Select	Extends the selection from the active cell to the bottom right corner of the region.

Virtual Worksheets

A Range object is a *virtual worksheet*. Cells can be referred to relative to the Range object, which need not necessarily coincide with the real worksheet.



Range("B2:D4").Range("B2").Select or

Range("B2:D4").Cells(2,2).Select or

Range("B2:D4").Cells(5).Select

In the example above, Range("B2:D4").Range("B2").Select selects B2 on the virtual worksheet, which actually corresponds to C3 on the real worksheet. However, when referring to a range within a virtual worksheet it is often better to refer to the range using the Cells method:

e.g. **Range("B2:D4").Cells(2,2).Select** selects the cell in the second row and second column of the range.

Range Contents

Data can be assigned to or returned from Range objects using one of three properties: Value, Formula or FormulaR1C1. In practice, the Value property can be used for all assignments.

Ranges can be formatted with the NumberFormat property.

The Text property is a read-only property that returns the formatted contents of a range.

Value Property

Use the Value property to assign or return text or numbers to or from a range.

Statement:	**Description**
Range("A1").Value = "Ability"	Assigns the text Ability to the cell A1.
Range("A1").Value = 25	Assigns the number 25 to the cell A1.
Selection.Value = 10	Assigns the number 10 to each of the cells in the current selection.
ActiveCell.Value = "Ability"	Assigns the text Ability to the active cell.
AnyVariable = ActiveCell.Value	Returns the contents of the active cell to the variable AnyVariable.
ActiveCell.Value = Range("A1").Value	Returns the contents of cell A1 and assigns it to the active cell.

Entering Formulas and Calculations

Use the Formula property to return a formula from a range. Use either the Value or Formula property to assign a formula to a range.

Statement:	Description
AnyVariable = ActiveCell.Formula	Returns the formula in the active cell to the variable AnyVariable.
Selection.Value = “=A1*2”	Assign the formula =A1*2 to the current selection. N.B. As the cell reference is relative, then only the first cell in the selection will actually contain =A1*2.
Selection.Value = “=$A$1*2”	Assign the formula =$A$1*2 to the current selection. N.B. As the cell reference is absolute, each of the cells in the selection will actually contain =$A$1*2.

if a cell contains a formula and it is returned with the value property then the value of the formula is returned and not the formula itself.

FormulaR1C1 Property

Use the FormulaR1C1 property to return a formula from a range. Use either the Value or FormulaR1C1 property to assign a formula to a range.

Statement:	Description
Selection.Value = “=r1c1*2”	Same result as using Selection.Value =“=$A$1*2”
Selection.Value = “=r[-1]c*2”	Assigns a formula that multiplies the contents of the cell above in the selection, by 2.
Selection.Value = “=sum(rc[-5]:rc[-1])"	Assigns a formula that sums the five cells to the right of each cell in the selection.

NumberFormat Property

Use the NumberFormat property to set or return the numeric format of a range.

Selection.NumberFormat = "£0.00"
Selection.NumberFormat = "d mmmm yyyy"

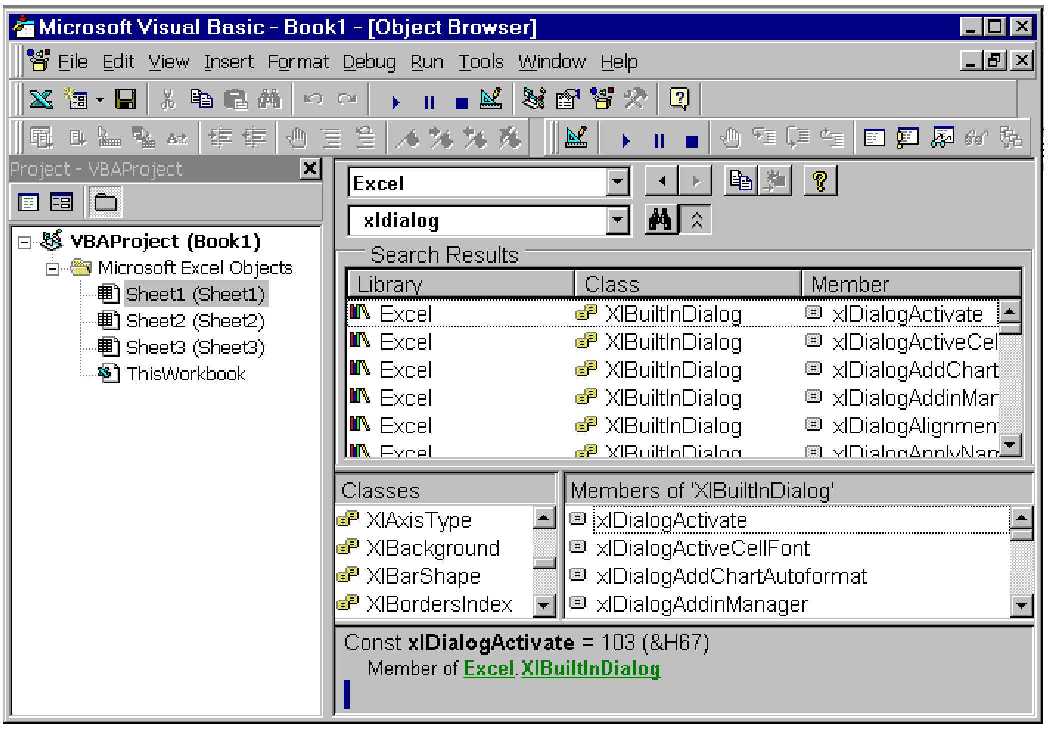
Input and Message Boxes

Built-In Dialog Boxes

Excel contains approximately 200 built-in dialog boxes. Each dialog box has a constant assigned to it; these constants all begin with xlDialog. Use the Object Browser to browse the list of dialog box constants.

- To view a list of dialog box constants:

1. In the Visual Basic Editor, choose **View** ⇨ **Object Browser** or press [F2].



Library	Class	Member
Excel	XlBuiltInDialog	xlDialogActivate
Excel	XlBuiltInDialog	xlDialogActiveCel
Excel	XlBuiltInDialog	xlDialogAddChart
Excel	XlBuiltInDialog	xlDialogAddinMar
Excel	XlBuiltInDialog	xlDialogAlignmen
Excel	XlBuiltInDialog	xlDialogApplyNar

Classes	Members of 'XlBuiltInDialog'
XlAxisType	xlDialogActivate
XlBackground	xlDialogActiveCellFont
XlBarShape	xlDialogAddChartAutoformat
XlBordersIndex	xlDialogAddinManager

Const **xlDialogActivate** = 103 (&H67)
Member of Excel.XlBuiltInDialog

2. Select the Excel library and type “xldialog” in the Search box.

3. Click the **Search** button

The constants correspond to dialog box names; for example, the constant for the Format Font dialog box is xlDialogFormatFont or 150.

This method may fail if an attempt is made to show a dialog box in an incorrect context. For example, to display the Format Charttype dialog box the active sheet must be a chart, otherwise the method fails.

The following example would display the built-in dialog box for Clear:

```
Sub ShiftF3()
     Application.Dialogs(xlDialogClear).Show
End Sub
```

Predefined Dialog Boxes

There are three ways to add predefined dialog boxes to an application, **InputBox function**, **InputBox method** and **MsgBox function**.

InputBox Function

The InputBox function displays a prompt in a dialog box and returns the text typed by the user in the supplied text box. It has an OK button and a Cancel button.

The syntax of the InputBox function is:

InputBox(prompt**,** title**,** default**,** xpos**,** ypos**,** helpfile**,** context**)**

Prompt is a string of characters displayed as the message in the dialog box. The maximum length is approximately 1024 characters.

Title is displayed in the title bar of the dialog box. It is optional and without it nothing is placed in the title bar.

Default is displayed in the text box as the default response if no entry is made. It is optional and without it the text box appears empty.

Xpos is the horizontal distance in twips of the left edge of the dialog box from the left edge of the screen. It is optional and without it the dialog box is horizontally centred.

20 twips equals 1 point. 72 points equals an inch.

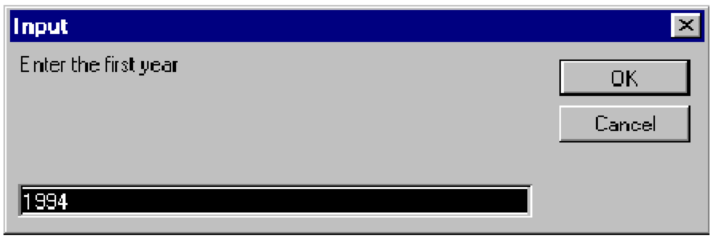
Ypos is the vertical distance in twips of the upper edge of the dialog box from the top of the screen. It is optional and without it the dialog box is positioned approximately one third of the way down the screen.

Helpfile identifies the file used to provide context sensitive help for the dialog box. If *helpfile* is specified a Help button is automatically added to the dialog box. *Context* must also be specified.

Context is a number indicating which help topic is required from the *helpfile*.

The following statement displays the dialog box pictured below. The default is set to 1994. The user's response is stored in a variable called FirstYear.

FirstYear = InputBox ("Enter the first year", , 1994)



If the user chooses OK or presses Enter the InputBox function returns whatever is in the text box.

If the user chooses Cancel or presses Escape the InputBox function returns a zero length string ("").

InputBox Method

The InputBox method displays a prompt in a dialog box and returns the data typed by the user in the supplied edit box. It has an OK button and a Cancel button.

The syntax of the InputBox method is:

*Object***.InputBox(**prompt**,** title**,** default**,** left**,** top**,** helpfile**,** context**,** type**)**

Prompt is displayed as the message in the dialog box. This can be a string, a number, a date or a Boolean value.

Title is displayed in the title bar of the dialog box. It is optional and without it nothing is placed in the title bar.

Default is displayed in the edit box as the default response if no entry is made. It is optional and without it the edit box appears empty. This value may be a range.

Left is the x position in points of the dialog box from the top left of the screen. It is optional and without it the dialog box is horizontally centred.

Top is the y position in points of the dialog box from the top left of the screen. It is optional and without it the dialog box is positioned approximately one third of the way down the screen.

Helpfile identifies the file used to provide context sensitive help for the dialog box. If *helpfile* is specified a Help button is automatically added to the dialog box. *Context* must also be specified.

Context is a number indicating which help topic is required from the *helpfile*.

Type specifies the type of data that can be returned from the dialog box. It is optional and without it the dialog box returns text.

The values of type may be:

Type	Returns
0	a formula
1	a number
2	text
4	a logical value (true or false)
8	a cell reference as a Range object
16	an error value such as #N/A
64	an array of values

A sum of the allowable values indicates that the input box can accept more than one type. For example setting type to 3 indicates that the input box can accept text or numbers.

The following statement displays the dialog box pictured below. The default is set to 1994, and only numbers will be accepted. The user’s response is stored in a variable called FirstYear.

```
FirstYear = Application.InputBox ("Enter the first year", ,
1994,,,"vba_xl.hlp",,1)
```

This could also be written as:

```
FirstYear = Application.InputBox (prompt:="Enter the first
year", _
default:=1994, helpfile:="vba_xl.hlp",type:=1)
```

The file “vba_xl.hlp” must be located in the Excel directory or a path specified.

If the user chooses the **OK** button or presses [RETURN] the InputBox method returns whatever is in the text box.

If the user chooses the **Cancel** button or presses [ESC] the InputBox method returns False.

MsgBox Function

The MsgBox function displays a prompt in a dialog box and waits for the user to choose a button. It returns a value indicating which button was chosen.

The syntax of the MsgBox function is:

MsgBox (prompt**,** buttons**,** title**,** helpfile**,** context**)**

If there is no return value, MsgBox acts as a statement and the arguments need not be enclosed in parenthesis.

e.g. MsgBox “Hello world!”

Prompt is a string of characters displayed as the message in the dialog box. The maximum length is approximately 1024 characters.

Buttons is the sum of the numbers used to identify: the number and type of buttons to display; the icon to use; the identity of the default button. See below for a list of the numbers.

Title is displayed in the title bar of the dialog box. It is optional and without it Microsoft Excel is placed in the title bar.

Helpfile identifies the file used to provide context sensitive help for the dialog box. If *helpfile* is specified a Help button is automatically added to the dialog box. *Context* must also be specified.

Context is a number indicating which help topic is required from the *helpfile*.

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

Number and type of button:

Constant	Value	Display:
vbOKOnly	0	OK button only.
vbOKCancel	1	OK and Cancel buttons.
vbAbortRetryIgnore	2	Abort, Retry, and Ignore buttons.
vbYesNoCancel	3	Yes, No, and Cancel buttons.
vbYesNo	4	Yes and No buttons.
vbRetryCancel	5	Retry and Cancel buttons.

Icon style:

Constant	Value	Display:	
vbCritical	16	Critical Message icon	🛑
vbQuestion	32	Warning Query icon	❓
vbExclamation	48	Warning Message icon	❗
vbInformation	64	Information Message icon	ℹ

Default button:

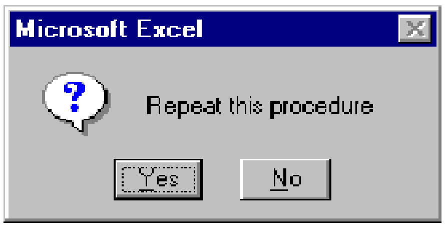
Constant	Value	Default
vbDefaultButton1	0	First button is default.
vbDefaultButton2	256	Second button is default.
vbDefaultButton3	512	Third button is default.

The value returned by the function depends upon which button was 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. The constants are specified by Visual Basic:

Button Selected	Constant	Value
OK	vbOK	1
Cancel	vbCancel	2
Abort	vbAbort	3
Retry	vbRetry	4

Ignore	vbIgnore	5
Yes	vbYes	6
No	vbNo	7

The following example performs some statements and then displays the following message box prompting whether the commands should be repeated:



Repeat this procedure

If the Yes button is pressed, the macro is repeated, otherwise the macro stops.

```
Sub Enter_Sales()
    Do Until Repeat = vbNo
            *statements*
            Repeat = MsgBox(prompt:="Repeat this
procedure", _
                    buttons:=vbYesNo + vbQuestion)
    Loop
End Sub
```

Exercise 4

1. Write a procedure to increase the size of the current selection by one row and one column.

2. Write a procedure to activate the last cell in the current selection.

3. Write a procedure to select the current region and then resize the selection so as not to include the first row and first column.

4. Write a procedure to select the current column of the current region.

5. Write a procedure to sum a column of numbers and extend the selection to include the total at the bottom.

Answers to Exercise 4

Question 1

```
Sub PlusRowPlusColumn()
    Selection.Resize(Selection.Rows.Count + 1, _
            Selection.Columns.Count + 1).Select
End Sub
```

Question 2

```
Sub LastCell()
    Selection.Offset(Selection.Rows.Count - 1, _
            Selection.Columns.Count - 1).Activate
End Sub
```

Question 3

```
Sub RegionLessLabels()
    Selection.CurrentRegion.Select
    Selection.Offset(1, 1).Resize(Selection.Rows.Count - 1, _
            Selection.Columns.Count - 1).Select
End Sub
```

Question 4

```
Sub CurrentColumn()
```

```
    Selection.Offset(ActiveCell.CurrentRegion.Row -
ActiveCell.Row). _

    Resize(ActiveCell.CurrentRegion.Rows.Count).Select
End Sub
```

Question 5

```
Sub SumCol()
    ActiveCell.Offset(Selection.Rows.Count).Value = _
    "=sum(r[-" & Selection.Rows.Count & "]c:r[-1]c)"
    Selection.Resize(Selection.Rows.Count + 1).Select
End Sub
```

Module 5 - Variables

A variable is a piece of information that is stored while a procedure is running. Once the information has been stored the variable can be referred to many times within the procedure. Each variable in a procedure is referred to by a unique name. If you don't declare your variables, they will all be of Varient Data Types, which is inefficient in terms of storage and speed, as you see later on.

Assigning Data to Variables

You can assign values to VBA variables. Think of a variable as a name that you can use to store a particular value.

To assign a value in cell A1 to a variable called Bonus, you could use the following statement.

```
Bonus = Worksheets(“Sheet1”).Range(“A1”).Value
```

It is also sensible to use comments to describe the purpose of variables so that you and other people can understand what could be otherwise cryptic names.

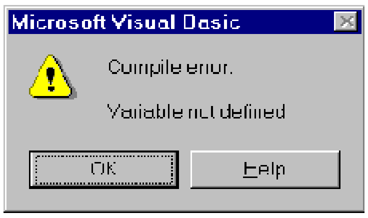
Declaring Variables

Variables that are to be used in a procedure are usually declared at the start of that procedure in order to identify them and the type of data that they will hold.

In Visual Basic it is not necessary to declare variables, but by doing so it is possible to speed up the procedure, conserve memory and prevent errors from occurring.

Because variables do not have to be declared Visual Basic assumes any variable that has not yet been used to be a new variable. This means that a misspelt variable name will not be recognised as such by Visual Basic.

This problem can be avoided by choosing to explicitly declare every variable. This tells Visual Basic that every variable will be declared in advance and any others used are therefore wrong. When Visual Basic encounters an undeclared variable the following message is displayed:


Microsoft Visual Basic

Compile error:

Variable not defined

To do this the following statement must be placed at the top of the Visual Basic module:

Local, Module-Level and Public Variables

A **local variable** is declared within a procedure and is only available within that procedure.

A **module-level variable** is available to all procedures in the module in which it is declared, but not to any other modules. Module-level variables are created by placing their **Dim** statements at the top of the module before the first procedure.

A **public variable** is available to every procedure in every module. Public variables are created by using the **Public** statement instead of the **Dim** statement and placing the declarations at the top of a module before the first procedure.

To conserve memory, declare variables at the lowest level possible, e.g. do not declare a variable as public or module-level if local is sufficient.

Naming Variables

See Appendix A for a recommended variable naming convention.

Data Types

The data type of a variable determines what type of data the variable can hold. For example a variable could hold an integer, a string or a date.

The data type is specified in the Dim statement:

Dim *variable* **As** *type*

If no type is specified the variable will be given the Variant data type.

Variable Scope

A Variant variable can hold numeric data, character text or a date. It can hold different types of data at different points in the same procedure. The current data type of the variable depends on the last value assigned to the variable.

```
Dim Anydata         ‘              Assumes Variant variable

Anydata = "30"      ‘              Anydata contains the string
"30"

Anydata = Anydata - 12 ‘           Anydata becomes numeric
and contains 18

Anydata = Anydata & "XYZ"          ‘Anydata becomes a string
containing "18XYZ"
```

There is no need to convert the data type before performing a calculation on the variable, however it would not be possible to perform an arithmetic operation on the variable if it does not contain data that is easily interpreted as a number, for example “18XYZ”

Specific Data Types

If a specific data type is assigned to a variable, the variable will only be able to hold that type of data. This can make a procedure easier to write and decode as it will always be known what the restrictions are for that variable.

Numeric Data

There are several different data types that can be specified to hold numeric data: Integer, Long, Single, Double and Currency.

Integer and **Long** variables hold whole numbers.

Integer variables can hold values between -32,768 and 32,767.

Long variables can hold values between 2,147,483,648 and 2,147,483,647.

Integers store less memory than variants and operations can be performed faster.

Single, **Double** and **Currency** variables can hold whole or part numbers.

Single and **Double** variables are held as floating point and are used for very large or very small numbers. They have much larger ranges than currency but may incur small rounding errors.

Single variables can hold values between -3.402823E38 and -1.401298E-45 for negative values and between 1.401298E-45 and 3.402823E38 for positive values.

Double variables can hold values between -1.79769313486232E308 and -4.94065645841247E-324 for negative values and between 4.94065645841247E-324 and 1.79769313486232E308 for positive values.

Decimal variables with 28 places to the right of the decimal; smallest non-zero number is +/-0.0000000000000000000000000001

Object variables are stored as 32-bit (4-byte) addresses that refer to objects. Using the **Set** statement, a variable declared as an **Object** can have any object reference assigned to it.

Currency variables hold the data as fixed point. This provides complete accuracy and is particularly useful for additions and subtractions but may not be so efficient for multiplication and division involving very large or very small numbers.

Currency variables can have up to 15 figures to the left of the decimal point, and up to 4 figures to the right.

Currency variables can hold values between -922,337,203,685,477.5808 and 922,337,203,685,477.5808.

Character Data

A variable that is to hold character information is declared as String. String variables can hold up to 65,535 characters.

By default a string variable is of variable length and the length of the variable depends on the length of the string of data that is stored in it.

Strings can be declared to have a fixed length:

Dim *variable* **As String *** *size*

To declare a string variable called name that is always of length 50:

```
Dim name As String * 50
```

If the data string that is assigned to the name variable is longer than 50 characters, the data is truncated. If the data string that is assigned to the name variable is shorter than 50 characters, the rest of the variable is filled with blanks.

Date and Time

Dates and times are held in variables that are of **Date** type. They are held as numbers. The number to the left of the decimal point represents the date and the number to the right of the decimal point represents the time. Midnight is 0, and midday is 0.5.

Any recognisable literal date or time can be assigned to a date variable, enclosed in # symbols.

For example:

```
Dim AnyDate As Date

AnyDate = #25/6/94 13:20#
AnyDate = #January 12, 1994 3:15am#
AnyDate = #15 March 1994#
AnyDate = #16:15#
```

Date variables can hold dates between January 1 0100 and December 31 9999, and times between 0:00:00 and 23:59:59.

Boolean

Boolean variables hold either True or False. This is useful when something has only two possible values. Instead of testing for the specific values, the Boolean variable can be set to true or false and it’s value tested at any necessary point in the procedure.

True and False values are often implied:

```
If Male Then                ‘Tests if Male is equal to True
If Not Male Then '      Tests if Male is equal to False
```

Arrays

An array is a special type of variable that can contain many elements of information simultaneously.

If an array is declared with a specific data type then every element in the array must be of that type. However it is possible to declare an array of variant variables.

An array can have fixed dimensions, in which case the number of elements that the array can contain is included in the declaration or the array can be dynamic and the size reset when required in the procedure

To declare an array:

Dim *variable* **(***dimension***) As** *Type*

The following creates an array called AnArray that has 15 variant variables contained within it:

```
Dim AnArray (14)
```

The specified dimension is the upper bound of the array. By default the lower bound of an array is 0 and thus the array has 15 elements. This is called a zero-based array.

It is possible to change the default lower bound by using the **Option Base** command before any declaration in a module.

For example the following statement would make the lower bound equal to 1:

Option Base 1

Alternatively, the lower bound can be provided when declaring the array using the **To** keyword:

```
Dim AnArray (1 To 15)
```

To declare a dynamic array, the dimension list should be left empty:

```
Dim AnArray ()
```

The array size can then be set at an appropriate point in the procedure using the **ReDim** statement, perhaps using the value of another variable.

The following sets the size of the array to be the value of the variable NumberFound:

```
ReDim AnArray (1 To NumberFound)
```

Using the optional keyword **Preserve** after ReDim preserves the data in an existing array when it is redimensioned.

Data is assigned to or returned from an array element by referring to the element’s position within the array.

The following sets the second element of AnArray to hold the number 56:

```
AnArray (2) = 56
```

An array can have up to 60 dimensions.

The following declares a two dimensional 5 by 3 array:

```
Dim AnArray (4, 2)
```

To assign data to the third element of the second dimension:

```
AnArray (2, 1) = 56
```

Objects

An Object variable can have objects assigned to it using the **Set** statement. This can make a procedure easier to read and quicker to write.

The following assigns a worksheet object to a variable:

```
Dim Sales As Object
Set Sales = Worksheets("Sales")
```

Constants

A constant is a named item that retains a constant value throughout the execution of a program, as opposed to a variable, whose value can change during execution. Constants are defined using the **Const** statement. Constants can be used anywhere in procedures in place of actual values. A constant may be a string or numeric literal, another constant, or any combination that includes arithmetic or logical operators except Is. For example:

```
Sub BackgroundColour()
      Const Red = 3
      Selection.Interior.ColorIndex = Red
End Sub
```

Using Option Explicit

Option Explicit

Variables are then declared using the Dim statement:

Dim *variablename*

The variable exists until the end of the procedure is met.

To save time the Option Explicit setting can be automatically set in all modules.

- To insert Option Explicit automatically in all new modules:

1. In the Visual Basic Editor, choose Tools ⇨ Options.
2. Select the Editor tab.
3. Select Require Variable Declaration.
4. Choose the OK button.

This command does not add “Option Explicit” to existing modules, only to new ones.

It is possible to declare variables without using Option Explicit. However, Visual Basic does not then notify any undeclared variables.

Exercise 5

1. Start a new workbook.

2. Enter the following data on a worksheet:

	A	B	C	D	E
1		Week1	Week2	Week3	Week4
2	Sales				
3	Costs				
4	Profit				

3. Enter a formula at B3 to calculate Costs at 20% of Sales in B2. Fill the formula across to E3.

4. Enter a formula at B4 to calculate Profit as Sales (B2) less Costs (B3). Fill the formual across to E4.

5. Rename the worksheet “Sales”.

6. Write a procedure named InputSales to prompt a user to enter the sales figures for weeks 1 to 4 of a given month:

a) Declare a string variable MonthOfSales to hold the month of the sales.

b) Declare a variant array SalesData to hold the four sales figures.

c) Use the InputBox function to prompt the user for the month of sales and store the answer in the variable MonthOfSales.

d) Use the InputBox function to prompt the user for the four weeks sales figures. Store each of the answers in the elements of the SalesData array.

e) Enter the contents of the MonthOfSales variable in the range A1 in upper case (use the Visual Basic function UCase()).

f) Enter the sales figures from the SalesData array in the range B2:E2.

g) Auto-fit the width of columns A to E. This command can be recorded in to the existing procedure.

h) Rename the module sheet “M_Sales”.

7. Assign the InputSales macro to a macro button on the Sales worksheet.

8. Save the workbook as SALES.XLS.

Answers to Exercise 5

```
Option Explicit

Sub InputSales()
    Dim MonthOfSales As String
    Dim SalesData(1 to 4)
    MonthOfSales = InputBox(prompt:= "Enter month of
sales", _
            title:= "Month")
    SalesData(1) =InputBox(prompt:= "Enter sales for week
1")
    SalesData(2) =InputBox(prompt:= "Enter sales for week
2")
    SalesData(3) =InputBox(prompt:= "Enter sales for week
3")
    SalesData(4) =InputBox(prompt:= "Enter sales for week
4")
    Range("A1").Value = UCase(MonthOfSales)
    Range("B2:E2").Cells(1).Value = SalesData(1)
    Range("B2:E2").Cells(2).Value = SalesData(2)
    Range("B2:E2").Cells(3).Value = SalesData(3)
    Range("B2:E2").Cells(4).Value = SalesData(4)
    Range("B2:E2").NumberFormat = "£0.00"
    Columns("A:E").EntireColumn.AutoFit
End Sub
```

Module 6 - Control Structures and Loops

A recorded procedure (macro) is executed statement by statement from the beginning to the end. It is possible however to use certain statements to change the flow of execution. For example, different sections of code can be executed depending on a condition or a section of code can be repeated a specified number of times or until a condition is met. These statements are called control structures as they control the flow of the procedure.

IF, Then, Select Case, Do, For and For Each, GoTo

Conditional control structures perform a block of statements depending upon the result of a condition.

If...Then...Else

The syntax of the **If...Then...Else** control structure is:

```
If condition Then
      statements to perform if condition is true
Else
      statements to perform if condition is false
End If
```

When Visual Basic meets the **If** statement it evaluates the condition as true or false.

If the condition is evaluated as true, execution of the procedure continues with the statements following the **If** until the **Else** statement is met. At this point execution continues from the command after the **End If** statement.

If the condition is evaluated as false Visual Basic looks for the **Else** statement and continues execution from the command after the **Else** statement until it meets the **End If** statement. At this point execution continues from the command after the **End If** statement.

Only one set of statements is performed.

The **Else** part of the structure is optional and may not always be necessary in which case the structure would be:

If *condition* **Then**
    Statements to perform if condition is true
End If

In this instance the statements will only be performed if the condition is true otherwise execution continues with statements after the **End If** statement.

If only one statement needs to be executed when the condition is true, the control structure can be written as:

If *condition* **Then** *single statement to perform if condition is true*

If several conditions are to be tested a series of **ElseIf** statements can be included:

If *condition1* **Then**
    Statements to be performed if condition1 is true
ElseIf *condition2* **Then**
    Statements to be performed if condition2 is true
ElseIf...
...
Else
    Statements to be performed if none of the conditions are true
End If

Visual Basic evaluates condition1. If condition1 is true Visual Basic executes the statements following the **If** statement until it meets an **ElseIf**, **Else** or **End If** statement at which point control is passed to the statements below the **End If**.

If condition1 is not true Visual Basic evaluates condition2, repeating the above procedure, and so on until a condition is found to be true.

If no conditions are true the statements after the **Else** statement are executed, if it exists.

Select Case

Select Case can be used as an alternative to **If...Then...ElseIf** where several conditions are to be tested. For **Select Case** the *same expression* has to be evaluated in every instance whereas for I**f...Then...ElseIf** each condition can involve different variables.

The syntax of the **Select Case** control structure is:

Select Case *expression*
Case *value1*
    statements
Case *value2*
    statements
...
Case Else
    statements
End Select

A single expression is evaluated at the top of the Case structure. Each case is then checked for this value, and the appropriate statements executed. Control then passes to the statement below the **End Select** statement. If no Case statement holds the evaluated value then the statements below the **Case Else** statements are executed. **Case Else** is optional.

Looping Control Structures

Do…Loop

The **Do...Loop** structure has two different syntax's:

1. Do While...Loop

Do While *condition*
    statements
Loop

When Visual Basic meets the **Do While** statement it evaluates the condition.

If the condition is false execution continues with the statements after the **Loop** statement.

If the condition is true Visual Basic performs the statements after the **Do While** statement. On meeting the **Loop** statement Visual Basic re-evaluates the condition and repeats the above process.

The statements inside the loop are executed until the condition is no longer true. If the condition is not true initially the statements will never be executed.

The statements inside a **Do While...Loop** structure must contain a way in which the condition can become false otherwise the loop will continue indefinitely!

2. Do Until...Loop

Do Until *condition*
      statements
Loop

When Visual Basic meets the **Do Until** statement for the FIRST time it continues with the statements directly below until it meets the **Loop** statement. On meeting the Loop statement Visual Basic evaluates the condition

If the condition is false execution continues with the statements after the **Loop** statement.

If the condition is true Visual Basic performs the statements after the **Do Until** statement until it meets the **Loop** statement. On meeting the Loop statement Visual Basic re-evaluates the condition and repeats the same process.

The statements inside the loop are executed until the condition is no longer true. The statements will always be executed at least once.

The statements inside a **Do Until..Loop** structure must contain a way in which the condition can become true otherwise the loop will continue indefinitely!

Do Until *condition***...Loop** is exactly the same as **Do While Not** *condition***...Loop**

Exit Do

The **Exit Do** statement can be used any number of times within a **Do...Loop** structure allowing a way out of the loop. On meeting the **Exit Do** command execution is continued with the statements after the **Loop** statement.

For...Next

The **For...Next** control structure allows a series of statements to be repeated a given number of times.

The syntax of the **For...Next** structure is:

For *counter* **=** *start* **To** *end* [**Step** *step*]
      statements

Next

The structure repeats the statements for a number of times depending on the values of *start, end* and *step*.

Counter is a variable containing a number. The initial value of *counter* is set equal to *start*. Each time the block of statements is executed *counter* is incremented by the value of *step*. *Step* can be positive or negative.

If *step* is positive the statements will be executed provided the value of *counter* is less than or equal to the value of *end*, otherwise execution continues with the statements after the **Next** statement.

If *step* is negative the statements will be executed provided the value of *counter* is greater than or equal to the value of *end*, otherwise execution continues with the statements after the **Next** statement.

Step has a default value of 1.

For Each...Next

The syntax of the **For Each...Next** structure is:

For Each *element* **In** *group*
    statements
Next

The Structure repeats the statements for each element in an array or collection

The following example converts every cell in the selection to upper case by using the Excel function UCase:

```
Sub UpperCase()
    For Each Item In Selection
            Item.Value = UCase(Item.Formula)
    Next
End Sub
```

“item” is an object variable.

The following example names all the worksheets in the active workbook Week 1, Week 2 etc.:

```
Sub NameSheets()
    Counter = 1
    For Each Sheet in Worksheets
```

```
        Sheet.Name = "Week " & Counter
        Counter = Counter + 1
    Next
End Sub
```

The **For Each...Next** structure is particularly useful for referring to collections of controls in customised dialog boxes.

Exit For

The **Exit For** statement can be used any number of times within a **For...Next** structure or a **For Each...Next** structure allowing another way out of the loop. On meeting the **Exit For** command, execution is continued with the statements after the **Next** statement.

Non-Structured Statements

GoTo

The **GoTo** statement causes the programme flow to branch unconditionally to a specified line within the same procedure. Lines are indicated by means of a label; a piece of text followed by a [:] (colon).

```
Sub TestBranch()
    ...
    If Number = 1 Then GoTo Finish
    ...
Finish:
    MsgBox "Macro ended"
End Sub
```

GoTo statements are difficult to read and debug. Use structured controls whenever possible.

Sub Procedures

It is possible to call one procedure from within another procedure by entering the procedure name as a statement.

This is particularly useful for a series of statements that are frequently required. The statements can be written as a separate procedure and the procedure called whenever the statements are required.

It also makes a procedure easier to read and decode if it performs just one or two tasks, calling other procedures as required.

If one procedure calls a second procedure, Visual Basic first looks for the second procedure in the module containing the first one. If the second procedure can not be found in the same module then Visual Basic looks in the other modules in the same workbook and then in other workbooks to which reference has been established (see below).

Calling Procedures in Another Workbook

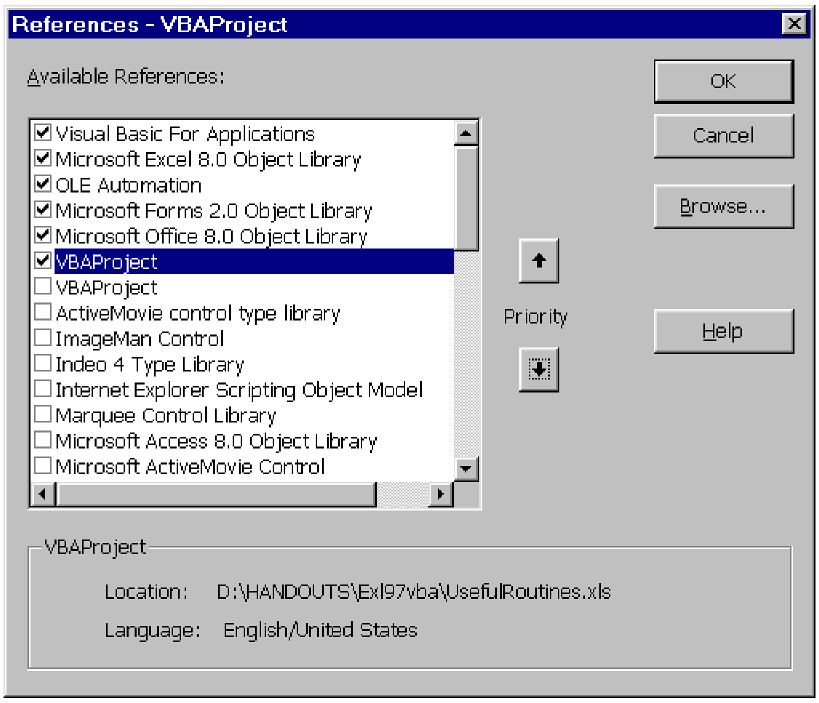
Procedures can only be called from another workbook if a reference from Visual Basic to that workbook is established.

- To establish a reference to another workbook:

1. In the Visual Basic Editor, open the workbook to which the reference is to be established and the workbook from which the reference is to be established.

2. In the Project Explorer window, select the project from which the reference is to be established.

3. Choose **Tools** ⇨ **References**.


References - VBAProject

Available References:

- [✓] Visual Basic For Applications
- [✓] Microsoft Excel 8.0 Object Library
- [✓] OLE Automation
- [✓] Microsoft Forms 2.0 Object Library
- [✓] Microsoft Office 8.0 Object Library
- [✓] VBAProject
- [ ] VBAProject
- [ ] ActiveMovie control type library
- [ ] ImageMan Control
- [ ] Indeo 4 Type Library
- [ ] Internet Explorer Scripting Object Model
- [ ] Marquee Control Library
- [ ] Microsoft Access 8.0 Object Library
- [ ] Microsoft ActiveMovie Control

VBAProject

Location: D:\HANDOUTS\Exl97vba\UsefulRoutines.xls

Language: English/United States

4. Check the box next to the VBAProject for the reference.

5. Choose the **OK** button.

Passing Arguments to Subprocedures

Subprocedures are a convenient way or “reusing” the same portion of procedure code more than once. However, the subprocedure may act differently depending upon where it is called from. This is achieved by “passing” the subprocedure additional information when it is called.

In the following example text is passed to a subprocedure “Alert” which is then used as an argument by the MsgBox statement:

```
Sub InputNumber()
    Number = InputBox("Enter a number between 100 and
200")
    Select Case Number
    Case Is < 100
            Alert Text:="too small"
    Case Is > 200
            Alert Text:="too big"
    Case Else
            Alert Text:="OK"
    End Select
End Sub



Sub Alert(Text)
    Beep
    MsgBox "Number is " & Text
End Sub
```

Exercise 6

1. In the workbook SALES.XLS on the module M_Sales:

a) Copy and paste the procedure InputSales, renaming the copy InputSales2.

b) In the procedure InputSales2, replace the four InputBox statements prompting for the sales figures with a single statement inside a FOR..NEXT loop. Include the week number in the prompt.

c) In the procedure InputSales2, replace the four statements entering the sales figures with a single statement inside the existing FOR..NEXT loop.

2. Choosing the Cancel button in an InputBox dialog box does not cancel the procedure, it only dismisses dialog box. Insert two lines in the InputSales2 procedure so that if a user chooses Cancel in an InputBox dialog box the procedure is exited.

3. Start a new workbook and insert a new module:

a) Write a procedure named LowerCase to change the case of each cell in a selection to lower case.

b) Write a procedure named ProperCase to change the case of each cell in a selection to proper (title) case.

c) Write a procedure named DelRows to give the user the opportunity to delete the active row in response to a MsgBox function prompt. Build a DO UNTIL...LOOP into the procedure so that it continues until the user chooses Cancel.

d) Save the workbook as CONTROLS.XLS (this workbook will be needed again later in the course).

4. Insert a new module in the CONTROLS.XLS

a) Record a procedure named Colours to change the interior colour and font colour of a selection of cells.

b) Write a procedure named ChangeColours that calls the procedure Colours passing it different arguments for the interior and font colours depending upon whether the selected cell contains a number above or below 10. Modify the Colours procedure accordingly.

c) Modify the ChangeColours and Colours procedures with a For Each...Next loop so that they will work on a selection of cells.

Answers to Exercise 6

Question 1

```
Sub InputSales2()
      Dim MonthOfSales As String
      Dim SalesData(1 to 4)
      Dim WeekNumber As Integer
      MonthOfSales = InputBox(prompt:= "Enter month of
sales", _
                title:= "Month")
      For WeekNumber = 1 To 4
                SalesData(WeekNumber) =InputBox( prompt:= _
                          "Enter sales for week " _ & WeekNumber)
                Range("B2:E2").Cells(WeekNumber).Value  = _
                          SalesData(WeekNumber) *
      Next
      Range("A1").Value = UCase(MonthOfSales)
      Columns("A:E").EntireColumn.AutoFit
End Sub
```

Question 2

```
Sub InputSales2()
      Dim MonthOfSales As String
      Dim SalesData(1 to 4)
      Dim WeekNumber As Integer
      MonthOfSales = InputBox(prompt:= "Enter month of
sales", _
                title:= "Month")
      If MonthOfSales = "" Then Exit Sub
      For WeekNumber = 1 To 4
                SalesData(WeekNumber) =InputBox( prompt:= _
                          "Enter sales for week " _ & WeekNumber)
                If SalesData(WeekNumber) = "" Then Exit Sub
                Range("B2:E2").Cells(WeekNumber).Value  = _
                          SalesData(WeekNumber) *
      Next
      Range("A1").Value = UCase(MonthOfSales)
      Columns("A:E").EntireColumn.AutoFit
End Sub
```

* The following statement can also be used outside of the For loop:

```
Range("B2:E2").Value = SalesData
```

Question 3

```
Sub LowerCase()
    For Each Cell In Selection
            Cell.Value = LCase(Cell.Formula)
    Next
End Sub
```

```
Sub ProperCase()
    For Each Cell In Selection
            Cell.Value = Application.Proper(Cell.Formula)
    Next
End Sub
```

```
Sub DelRows()
    Do Until Response = vbCancel
            Response = MsgBox("Delete row " _
                    & ActiveCell.Row & "?", 3)
            If Response = vbYes Then
                    ActiveCell.EntireRow.Delete
            Else
                    ActiveCell.Offset(1, 0).Select
            EndIf
    Loop
End Sub
```

Question 4

```
Sub Colours()
    With Selection.Interior
            .ColorIndex = 5
            .Pattern = xlSolid
    End With
    Selection.Font.ColorIndex = 3
End Sub
```

```
Sub ChangeColours()
    Const Red = 3, Blue = 5
    If ActiveCell.Value > 10 Then
            Colours inside:=Blue, text:=Red
    Else
            Colours inside:=Red, text:=Blue
    End If
End Sub



Sub Colours(inside, text)
    With Selection.Interior
            .ColorIndex = inside
            .Pattern = xlSolid
    End With
    Selection.Font.ColorIndex = text
End Sub
```

```
Dim Cell as Object          'declare as module level variable
Sub ChangeColours()
    Const Red = 3, Blue = 5
    For Each Cell In Selection
        If IsNumeric(Cell.Value) And Not
IsEmpty(Cell.Value) Then
            If Cell.Value > 10 Then
                Colours inside:=Blue, text:=Red
            Else
                Colours inside:=Red, text:=Blue
            End If
        End If
    Next Cell
End Sub



Sub Colours(inside, text)
    With Cell.Interior
        .ColorIndex = inside
        .Pattern = xlSolid
    End With
    Cell.Font.ColorIndex = text
End Sub
```

Module 7 - Objects, Properties, Methods, Events and Error Handling

Errors

It is only possible to verify that code is performing correctly if it is tested with every possible combination of input and if all output is checked. Visual Basic provides various debugging tools to help locate the source of any errors that are detected. These tools help to identify exactly where something has gone wrong. For example, **Step Into, Step Over** and **Step Out** allow the flow of execution to be followed statement by statement, **Break Mode** allows execution to be halted at a given point and the value of an expression displayed.

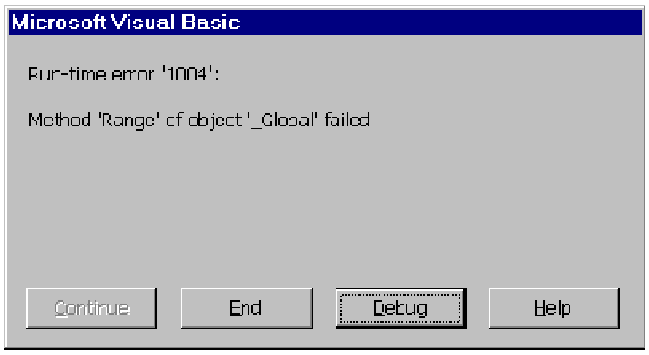
Types of Errors

There are various types of errors that can occur when writing and running a procedure. These can be broadly separated into three areas: language errors, run-time errors and logic errors.

Language errors occur when the code is written incorrectly. Visual Basic detects these errors when the insertion point is moved away from the line while writing the procedure or just before the procedure is run. One way to help minimise language errors is by using the **Option Explicit** statement. With this statement included Visual Basic will only accept variable names that have been declared and will detect any errors in the spelling of variable names.

Run-time errors occur and are detected by Visual Basic when a statement tries to perform an operation that cannot be carried out. When this happens Visual Basic halts execution of the macro with a series of available options.

For example, the following error may occur when a statement refers to an object that does not exist:



Run-time error '1004':

Method 'Range' of object '_Global' failed

The type of error that has occurred is described using an error number and a description. Various options are available:

Choose the **End** button to end the macro at that point.

Choose the **Debug** button to highlight the offending code in the Visual Basic Editor.

Choose the **Help** button for help about the type of error.

Logic errors are errors that occur when the code appears to perform correctly but doesn't produce the correct result. Logic errors can also produce run-time errors. For example a particular case may be missing from a Select Case statement, or a variable may contain a different value to the one expected causing another variable to be set incorrectly at a later point in the procedure. The larger a procedure is the more difficult it is to detect logic errors and it is for the detection of logic errors that the debugging tools are particularly useful.

Capturing Errors

If an error occurs whilst a macro is running, a run-time error, it will either cause the macro to fail or make it act unpredictably, unless the error is trapped in some way.

The simplest way to deal with errors is to use the On Error statement which can take three forms. The error "trap" is "turned on" prior to the likely occurrence of an error and "sits in wait" until an error occurs or until the trap is "turned off".

Notice that an error-handling routine is not a Sub or Function procedure. It is a block of code marked by a line label or line number. To prevent the block of code from being executed, when no error has occurred, it must be bypassed, or the procedure or function ended, before it is reached (see below).

Manipulating Errors

On Error Goto Label

If this statement is placed before the instruction that causes the error, control of the macro branches to the line with the label.

The number of any error generated is returned by the Err function.

Resume and Resume Next

The Resume statement returns control to the line at which the error occurred and attempts to run the statement again.

The Resume Next statement passes control in the macro to the line following the line that caused the error.

Using a Resume statement anywhere except in an error-handling routine causes an error.

On Error Goto 0

This statement disables a previous On Error Resume Next or On Error Goto Label statement. When the next error occurs an error message will be generated and the macro will fail.

Err Function

When On Error Resume Goto label is used to trap errors the number of any error is returned by the Err function. Err acts like a public variable. The value of Err can be tested and acted upon.

Error() Function

The Error function returns message text corresponding to an error number.

Error and Err Statements

The Error statement simulates an error, sets the value of Err and interrupts the macro. The Err statement sets the value of Err but does not simulate an error or interrupt the macro.

Error Handling Example

In the following example an error trap is set to prevent the macro failing if the active workbook is saved as DATA.XLS to drive A. If there is no disk in drive A then an error number 1004, with associated message “Cannot access ‘A:’ ”, occurs. Control is passed to the line ErrHandle. The error handling code tests the Error( ) function to see if no disk was present (N.B. cannot test the value of Err as this returns 1004 for other reasons). A message box is displayed informing the user of the original error message and asking them to insert a disk. When the **OK** button is chosen the procedure resumes at the line that caused the error, i.e. ActiveWorkbook.SaveAs.

If a disk is present in A but the file DATA.XLS already exists on that disk Excel displays a message asking if the file is to be overwritten. If the user chooses the **No** or **Cancel** buttons error 1004 occurs but the value of Error is different. In this case, the error handling code displays a message to inform the user that the file has not been saved and the procedure resumes at the line after the one that caused the error.

When the file is saved successfully and no error occurs, then the error handling code is not executed, or executed again, because the macro is exited before it is reached.

```
Sub ErrorTest()
    On Error GoTo ErrHandle
    ActiveWorkbook.SaveAs Filename: = "A:\DATA.XLS"
    On Error GoTo 0
    Exit Sub
ErrHandle:
    If Error() = "Cannot access ‘A:’." Then
        MsgBox Error() + "Insert floppy disk in drive A."
        Resume
    Else
        MsgBox "File not saved."
        Resume Next
    End If
End Sub
```

Module 8 – Debugging

Break Points

Execution of a procedure can be halted at a specific point by placing a **breakpoint** into the code. When a breakpoint is met Visual Basic enters break mode. The values of variables can then be viewed and execution can be continued in **step** mode. Alternatively, execution can be normally resumed and the procedure will continue running until the next breakpoint is met, or execution ends normally.

Setting Breakpoints

- To set or remove a breakpoint:

1. Activate the module containing the procedure where the breakpoint is to be placed/removed.

2. Position the insertion point anywhere within the statement where the breakpoint is to be placed/removed.

3. Choose **Debug ⇨ Toggle Breakpoint**, choose the Toggle Breakpoint button from the Edit toolbar or press [F9].

Quick Watch

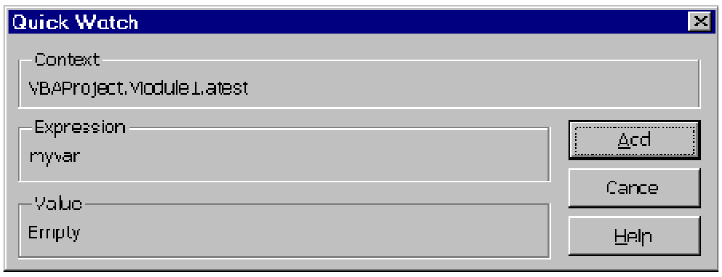
Instant watch can be used to display the value of any expression while in break mode.

- To display the value of an expression with Instant Watch:

1. When in break mode, select the expression to be evaluated.

2. Choose the Quick Watch button from the Debug toolbar.

The Quick Watch dialog box is displayed:


Quick Watch

Context: VBAProject.Module1.atest

Expression: myvar

Value: Empty

Choose the **Add** button to add the expression to the Watch pane of the debug window. The expression can then be watched for the rest of the procedure.

Stepping

Stepping can be used to step through each statement of a procedure while in break mode. On selecting one of the step options the current statement is executed and then the procedure is halted again. This enables the values of any expressions to be displayed at any time, the flow of execution to be observed and the exact point at which an error occurs to be spotted.

There are three ways to step through a procedure, **Step Into, Step Over** and **Step Out**. These are selected by choosing the appropriate button on the Debug toolbar.

Step Into will step through all statements including any that are in a called procedure. When the end of the called procedure is reached Step Into returns to the procedure that contained the call.

Step Over is identical to Step Into except that it treats any called procedure as one statement to be executed. It executes every statement in the called procedure before halting at the statement following the call.

Step Back executes the remaining lines of a procedure in which the current execution point lies.

Module 9 - Forms (dialog boxes)

Dialog boxes and forms allow applications to interact with their users. A *built-in* Excel dialog box can be used in a procedure. A *predefined dialog box* can be used, allowing a quick and easy way to request information from or display information to the user. *Custom forms* can also be defined incorporating various *controls* including selection lists and check boxes allowing more extensive interaction.

Controls can also be added to worksheets and chart sheets to create custom forms.

Custom Forms

Custom forms can be created on worksheets, chart sheets or in the Visual Basic Editor.

Controls

Custom forms are created by adding controls to a worksheet, chart sheet or user defined form. Buttons, text boxes, list boxes and check boxes are all examples of controls. It is even possible to use controls from outside of Excel. These type of controls can be especially written and are known as *ActiveX* controls.

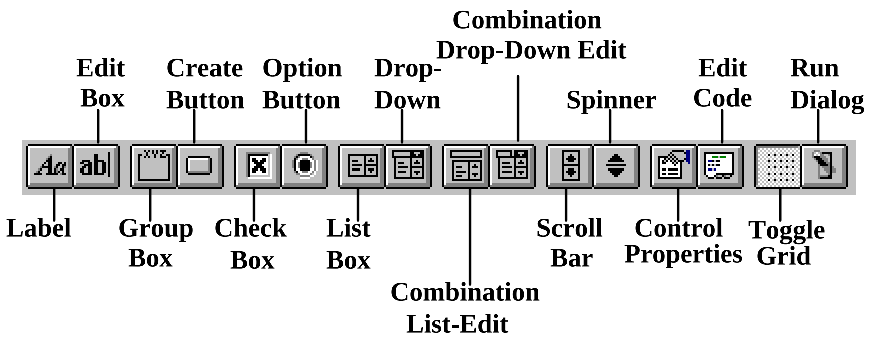
The simplest way to create a form is to place controls directly onto a worksheet so that they are near to relevant cells and those cells recalculate automatically. This is particularly useful if only one control is required.

Forms designed in the Visual Basic Editor can be much more sophisticated.

Worksheet Forms

Toolbar Controls

Controls are placed on a worksheet using the Forms toolbar:



- To place a control onto a worksheet:

1. Click the button for the required control on the Forms toolbar.

2. On the worksheet, drag until the control is the required size and shape and release the mouse button.

The following controls are available from the Forms toolbar:

Control	Description
Label	Text that is displayed to the user, including names, instructions etc.
Edit Box	A box in which the user can enter text, numbers or cell references.
Group Box	A border containing a group of buttons or other controls.
Button	A command button such as OK or Cancel.
Check Box	A box indicating whether an option is set.
Option Button	A button for selecting one of a group of mutually exclusive options. A group of option buttons should be placed in a group box.
List Box	A list of options, one or more of which can be selected.
Drop-Down	A single uneditable text box with an arrow, paired with a drop-down list that appears when the user selects the arrow.
Combination List-Edit	A single editable text box combined with a list box.
Combination Drop-Down Edit	An empty edit box with an arrow, paired with a drop-down list that appears when the user selects the arrow.
Scroll Bar	A horizontal or vertical scroll bar for changing numeric values.

Spinner	A pair of buttons for incrementing or decrementing a displayed value.

The Edit Box and Combination list boxes can not be inserted on a worksheet.

Setting the Properties of a Control

Certain default properties are assigned to each control. Some of these properties can be changed using the Control Properties button on the Forms toolbar, or the **Format** menu.

- To change a property:

1. Select the control.

2. Choose **Format** ⇨ **Control**, or choose the Control Properties button on the Forms toolbar.

3 Change the required property.

4. Choose the **OK** button.

Cell Links

Check box, option button, list box, drop-down, scroll bar and spinner controls have a Cell Link property. A value is returned to the Cell Link corresponding to the item or option selected or set by the user in the control.

The value in the cell link may be directly of use in the worksheet or it may need to be evaluated in a formula containing functions such as IF(), INDEX() or lookup.

The cell link need not be on the same worksheet as the control.

Input Ranges

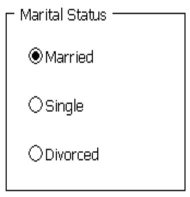
List box and drop-down controls have an Input Range property. This property is used to set the list of values to appear in the list box.

Group Boxes and Option Buttons

Option buttons are normally arranged in groups. A user can choose one option from a group; the options are mutually exclusive.

To create a group of option buttons, first add a group box of sufficient size to hold all of the option buttons it is to contain, then add the required option buttons into the group. Make sure the options do not stretch outside of the group box.

When a user select an option button in the group a value is returned to the cell link corresponding to order of the option button in the group. For example, if a user chose the Single option from the following group, the value 2 would be returned to the cell link:



The order of the options buttons is the order in which they were added to the group.

Lists and Drop-Down Lists

To create a list or drop-down list on a worksheet, first type the list of values to appear in the list on some other part of the worksheet or in another worksheet or workbook.

Add a list or drop-down control and set the input range property to the cell range containing the typed list. The value returned to the cell link corresponds to the value's position in the list. For example, the second value in the list would return the value 2 to the cell link.

User Defined Forms

Use the following procedure to create a user defined form or custom dialog box:

1. Create a UserForm

   In the Visual Basic Editor, choose **Insert** ⇨ **UserForm**.

2. Add controls to the UserForm

Find the control to add in the Toolbox and drag the control onto the form.

3. Set control properties

   Right-click a control in design mode and click Properties to display the Properties window.

4. Initialise the controls

   Initialise controls in a procedure before the form is shown, or add code to the Initialize event of the form.

5. Write event procedures

   All controls have a predefined set of events. For example, a command button has a Click event that occurs when the user clicks the command button. Write event procedures that run when the events occur.

6. Show the dialog box

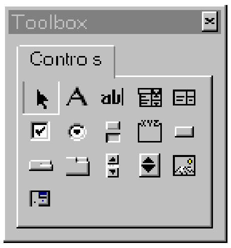
   Use the Show method to display a UserForm.

7. Use control values while code is running.

   Some properties can be set at run time. Changes made to the dialog box by the user are lost when the dialog box is closed.

Adding Form Controls

To display the Toolbox, choose **View ⇨ Toolbox** or click the Toolbox button on the Standard toolbar.



The Toolbox contains a similar list of controls to the Excel Forms toolbar. Additional controls include: TabStrip, MultiPage, RefEdit and Image.

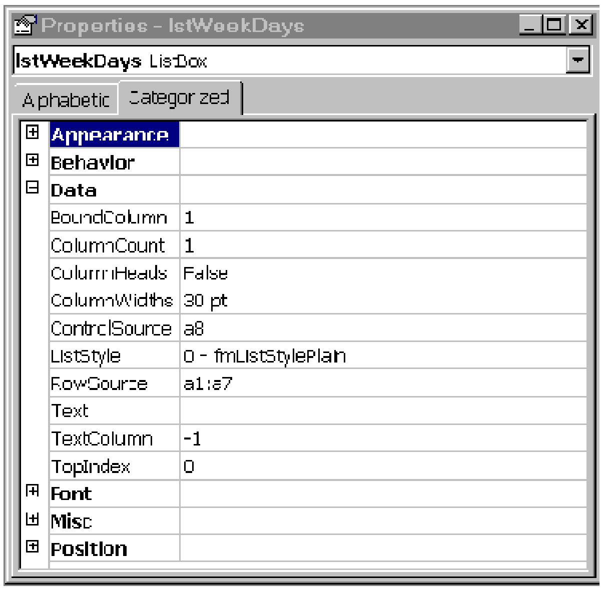
Use the **Format** commands to align and arrange the controls on the form.

Form Control Properties

Setting Form Control Properties

Each form control has a list of properties that can be displayed in the Properties window. Different controls have different properties.

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.


Properties - lstWeekDays

lstWeekDays ListBox

Alphabetic | Categorized

⊞ Appearance	
⊞ Behavior	
⊟ Data	
BoundColumn	1
ColumnCount	1
ColumnHeads	False
ColumnWidths	30 pt
ControlSource	a8
ListStyle	0 - fmListStylePlain
RowSource	a1:a7
Text	
TextColumn	-1
TopIndex	0
⊞ Font	
⊞ Misc	
⊞ Position	

- Data Properties

Some of the most useful properties can be set in the Data category.

The **RowSource** property specifies the source, providing a list for a ComboBox or ListBox. The RowSource property accepts worksheet ranges from Excel.

The **ColumnSource** property identifies the data location used to set or store the Value property of a control. The ControlSource property identifies a cell; it does not contain the data stored in the cell. If the Value of the control is changed, the change is automatically reflected in the linked cell. Similarly, if the value of the linked cell is changed, the change is automatically reflected in the Value of the control.

The default value for ControlSource is an empty string. If

ControlSource contains a value other than an empty string, it identifies a linked cell. The contents of that cell is automatically copied to the Value property when the control is loaded.

Form Control Naming Convention

See Appendix A for a recommended naming convention for form controls.

Form Initialisation

Initialise controls at run time by using Visual Basic code in a macro. For example, fill a list box, set text values or set option buttons.

The following example uses the AddItem method to add data to a list box and then it sets the value of a text box and displays the form UserForm1:

```
Private Sub GetUserName()
    With UserForm1
            .lstWeekDays.AddItem "Monday"
            .lstWeekDays.AddItem "Tuesday"
            .lstWeekDays.AddItem "Wednesday"
            .lstWeekDays.AddItem "Thursday"
            .lstWeekDays.AddItem "Friday"
            .lstWeekDays.AddItem "Saturday"
            .lstWeekDays.AddItem "Sunday"
            .txtSalesPerson.Text = "Fred Bloogs"
            .Show
    End With
End Sub
```

Use code in the **Initialize event** of a form to set initial values for controls on the form. An advantage to setting initial control values in the Initialize event is that the initialisation code stays with the form.

Control and Dialog Box Events

UserForms and controls have a predefined set of events. For example, a command button has a Click event that occurs when the user clicks the command button and UserForms have an Initialize event that runs when the form is loaded.

To write a control or form event procedure, open a module by double-clicking the form or control and select the event from the Procedure drop-down list box.

Event procedures include the name of the control. For example, the name of the Click event procedure for a command button named cmdOK is cmdOK_Click.

If code is added to an event procedure and then the name of the control is changed, the code remains in procedures with the previous name.

To simplify development, it is a good practice to name controls correctly before writing event code.

Displaying and Dismissing a User Form

Show

To test a user form in the Visual Basic Editor, click Run Sub/UserForm button or press [F5].

To display a user form from Visual Basic, use the Show method. The following example displays the user form box named UserForm1:

```
Sub DisplayMyForm()
    UserForm1.Show
End Sub
```

Unload

Use the **Unload** method to remove a user form from memory. For example, the following code could be assigned to a cancel button on a form named UserForm1:

```
Private Sub cmdCancel_Click()
    Unload UserForm1
End Sub
```

The **Me** keyword behaves like an implicitly declared variable. For example, the following code could be assigned to a cancel button on any user form:

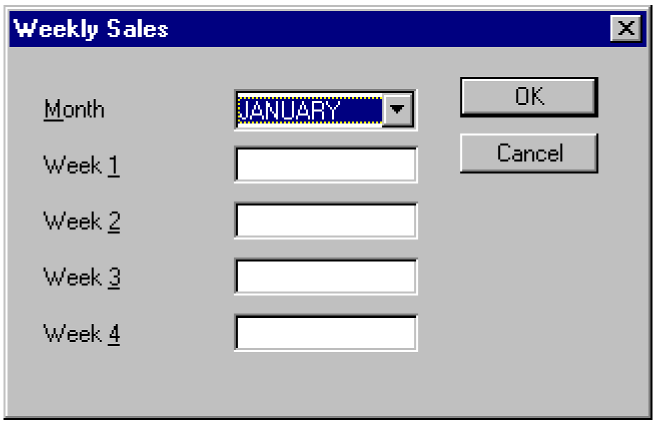
```
Private Sub cmdCancel_Click()
```

```
    Unload Me
End Sub
```

Exercise 7

1. Open the workbook SALES.XLS.

a) In the Visual Basic Editor insert a user form and create the dialog box below by adding label, edit box and drop-down controls:



b) Write a procedure to show the dialog box and insert the results onto the Sales worksheet but only if the user chooses the **OK** button.

c) Create an procedure to initialise the dialog box when it is shown.

2. Open the workbook WORKSHEET CONTROLS.XLS and investigate the worksheet controls and linked formulae.

3. Open the workbook LOAN FORM.XLS and investigate the custom dialog box and associated procedures.

Answers to Exercise 7

Question 1

```
Sub WeeklySales()
    Dim Count As Integer
    If DialogSheets("D_Sales").Show Then
            Worksheets("Sales").Unprotect
            Worksheets("Sales").Range("A1").Value = _
                    Worksheets("Links").Range("A14").Value
            For Count = 1 To 4

    Worksheets("Sales").Range("B2:E2").Cells(Count).Value
= _
                    DialogSheets("D_Sales").EditBoxes("Week"
& Count).Text
            Next

    Worksheets("Sales").Columns("A:E").EntireColumn.Auto
Fit
            Worksheets("Sales").Protect
    End If
End Sub



Sub Sales_Frame_Show()
    Dim box As Object
    DialogSheets("D_Sales").DropDowns("Month").Value =
Month(Now())
    For Each box In DialogSheets("D_Sales").EditBoxes
            box.Text = ""
    Next
End Sub
```

Module 10 – menus and toolbars

Building Custom Menus

In previous versions of Excel, toolbars contained only buttons. In Excel toolbars can contain buttons, menus or a combination of both.

The menu bar is a special toolbar at the top of the screen that contains menus such as **File**, **Edit** and **View**. The default menu bar contains menus and commands for working with worksheets. If working with a chart sheet or an embedded chart, the chart menu bar is displayed instead. Menu bars can be customised just like any built-in toolbar.

Some menu commands have images next to them so the user can quickly associate the command with the corresponding toolbar button.

When Excel is quit, changes made to the menu bar and built-in toolbars, any custom toolbars created and the toolbars currently displayed are saved in a toolbars settings file in the Windows folder. This settings file is saved as **username8.xlb**, where **username** is the user's Windows or network log-in name. If the computer is not connected to a network or not set up with a log-in prompt, the settings file is saved as **excel8.xlb**. The toolbar configuration saved in this file is used by default each time Excel is started.

Toolbars created or customised are available to all workbooks on the computer. To ensure that a custom toolbar is always available with a specific workbook, attach the toolbar to the workbook.

Customising a Toolbar

Change, or customise, a toolbar so it contains buttons or menu items for the most often used commands or macros. A menu bar, such as the Worksheet or Chart menu bar, are just special toolbars

- To add a custom button or menu item to a toolbar:

1. Choose the Commands tab of **Tools** ⇨ **Customize**.

2. From the Categories list box, select Macros.

3. Drag the Custom Menu Item or Custom Button topic from the Commands list box over the toolbar and then release the mouse. On a menu bar, drag the horizontal line to position on the menu where the item or button is to be inserted.

4. Right-click the added custom button or menu item and choose **Assign Macro**.

5. Select the macro to assign to the button or menu item.

6. Right-click the added custom button or menu item and edit the Name of the button or item. Include an "&" for an underlined character.

7. Choose the **Close** button.

Attaching Toolbars to Workbooks

To make sure that a custom toolbar is always available with a specific workbook, attach the toolbar to the workbook. Be sure to save the workbook after attaching a toolbar.

- To attach a custom toolbar:

1. Create the custom toolbar to attach to the workbook.
2. Open the workbook to which the toolbar is to be attached.
3. Choose the Toolbars tab of **Tools ⇨ Customize.**
4. Click the **Attach** button.

Click the custom toolbar to attach, and then click the **Copy** button.

Save More Than One Toolbar Configuration

If a group of customised toolbars are frequently used, sized and arranged on the screen in a particular way, save the configuration so that it does not have to be redisplayed each time.

- To save a toolbar configuration:

1. Make any changes to the built-in menu bar and toolbars and create any custom toolbars to save in the configuration. Then display the toolbars the way they are to appear.

2. Quit Excel.

3. In the Windows folder, locate the file named Username8.xlb or Excel8.xlb.

4. Rename the file, retaining the .xlb extension.

5. To use a saved configuration again, use **File** ⇨ **Open** to open the renamed toolbars settings file. Excel creates a new default toolbars settings file Username8.xlb when Excel is next quit.

Creating Custom Toolbars

If a macro is assigned to a toolbar, the macro is available at any time and for every sheet in the workbook, provided the toolbar is displayed. The workbook containing the macro is automatically opened, if it is not already open, when the button or menu item is clicked from another workbook.

A macro is usually assigned to a custom button or a custom menu item, but a macro can be assigned to a built-in button or item overriding the button's or item's normal function.

Display

ScreenUpdating

Turn screen updating off to speed up macro code. This will not only make procedures run faster but will also prevent distracting screen "flickering".

- To turn off screen updating:

```
Application.ScreenUpdating = False
```

Screen updating is automatically turned back on when the macro ends or it can be turned on during the course of a procedure by setting the property to True.

DisplayAlerts

Stop Excel’s prompts and alert messages while a macro is running. Excel chooses the default response.

- To stop the display of alerts:

```
Application.DisplayAlerts=False
```

Excel sets the DispalyAlerts property back to True when the macro stops running.

Exercise 8

1. Open the workbook CHANGE CASE.XLS.

2. Add the menu item Change Case to the Format menu of the Worksheet toolbar. Use “h” as the **accelerator key**.

3. Add the submenu items Lower, Proper and Upper to the Change Case menu item. Assign the macro LowerCase to the menu item Lower, the macro ProperCase to the menu item Proper and the macro UpperCase to the menu item Upper. Use “L”, “P” and “U” respectively as the accelerator keys.

4. Resave the workbook CHANGE CASE.XLS.

Module 11 – optional extras(if required)

Event Triggered Procedures

An event-handler procedure is a specially named procedure that is executed when a specific event occurs.

A example would be the CommandBttn1_Click procedure that is executed when a user clicks a command Button that is stored on a user form or a worksheet.

Below are examples of some of the events that Excel can recognise;

A workbook is opened or closed

A Cell is double-clicked

A workbook is saved

Data in a cell is entered or saved

The data in a chart is updated

A worksheet is activated or de-activated

To enable or disable events, execute the following VBA code;

Application.EnableEvents = False

A word of warning;

Disabling Events in Excel applies to all workbooks. E.G. If you disable events in your procedure and then open another workbook that has, say, a Workbook_Open Procedure, that procedure will not execute…

The Activate Event

The following procedure is executed whenever the workbook is activeted. This procedure simply maximises the active window.

```
Private Sub Workbook_Activate()

    ActiveWindow.WindowState = xlMaximized

End Sub
```

Examples of Other Workbook Events;

Activate	A Workbook is activated
BeforeClose	A Workbook is about to be closed
BeforeSave	A Workbook is about to be saved
Open	A Workbook is about to be opened
SheetCalculate	Any Worksheet is calculated (or Recalculated)
WindowResize	Any Workbook window is resized

Creating Add–In Applications

An add-in is added to a spreadsheet to give it added functionality.

One of the most popular is shipped with Excel, that's the Analysis ToolPak, which adds statistical and analysis capabilities which are not built into Excel.

Add-ins can be loaded and unloaded using the Tools – Add-Ins command.

Appendix A
Naming Conventions

Objects should be named without underscores, hyphens or other unusual characters such as **)(/?\[ ] {+=}**. A mixture of different letter case should be used to show word breaks such as JobNumber. Names that will conflict with Visual Basic properties or keywords such as “Name”, “Caption” and “Size” should be avoided.

User Form Controls

The following convention applies to unbound controls and to Visual Basic controls. Controls that are not to have any code attached to them will obviously not have to be renamed.

Control Type	**Prefix**
TextBox	txt
Label	lbl
ListBox	lst
ComboBox	cmb
CommandButton	cmd
Frame	grp
OptionButton	opt
ToggleButton	tog

Memory Variables

Memory variables should be prefixed with three lower case letters to indicate the data type of the variable:

Data Type	**Prefix**
String	str
Long	lng
Single	sng
Variant	var
Integer	int
Double	dbl

Index

P R E M I E R

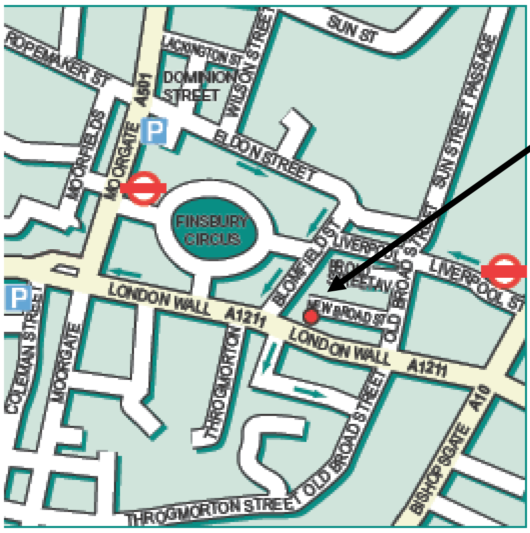
Consultancy and Training Services

Management ♦ Personal Development ♦ Software ♦ Technical ♦ e-Learning ♦ e-Business ♦ Business Integration ♦

Premier offers a comprehensive range of services including

Full training needs analysis	Consultancy
Tailored courses	Seminars
Scheduled course programme	Workshops
On-site training	‘One to one’ coaching
Training audits	e-Learning

How to find Premier’s City training centre



We are here:
Maven Centre
3rd Floor, 42 New Broad Street
London
EC2M 1SB

Bookings and enquiries:

Telephone +44 (0) 20 7729 1811
Fax +44 (0) 20 7729 9412
Email enquiries@premiercs.co.uk
Website www.premcs.com

Microsoft CERTIFIED Partner