MICROSOFT EXCEL
VISUAL BASIC FOR APPLICATIONS
INTERMEDIATE

IMPORTANT NOTE

Unless otherwise stated, screenshots in this lesson were taken using Excel 2007 running on Window XP Professional. There may be, therefore, minor differences in the appearance and layout of dialog boxes and screens if you are using other Excel versions, or if you are running on Windows 2000 or Windows Vista.

Concepts, discussions, procedures and functionality, however, remain unchanged.

¶

Contents

¶

REVIEW EXERCISE

AUTOMATING A WORKSHEET USING VBA BASICS

1. Open the file, **Daily Profit and Loss**.
2. Launch the VBA Editor and open the module sheet in the **Daily Profit and Loss** project containing the **CreateTable** procedure.
3. Add code to the module that:
    a. adds formulas in cells B4 and C4
    b. formats cells B1 to C2 with a bold font
    c. formats cells A2 to A4 with an italic font
    d. formats cells A4 to C4 with a grey fill (***Tip***: ColorIndex = 15)
4. Create a button on **Sheet1** of the workbook and assign the **CreateTable** macro to it.
5. Use this button to run the macro and check that it runs correctly. It should result in the following.

	A	B	C
1		**USA**	**Europe**
2	*Sales*		
3	*Costs*		
4	*Profit*	0	0

6. Correct any errors.
7. Enter the following data into the newly created table.

	USA	**Europe**
Sales	35,000	42,000
Costs	25,000	25,000

8. In the same module sheet there is already a procedure named **TestProfit** that tests cell B4 and makes the font bold if its value is 15000 or over, and formats it red if below 15000.

    Edit the procedure with a loop so that after it tests cell B4, it also tests cell C4.
9. Assign a keystroke of your choice to this macro and run it to check that it works correctly. The font in cell B4 should be made red and the font in cell C4 bold.
10. Put right any errors and then save and close the file.

LESSON 1 - EXPLORING THE RANGE OBJECT

In this lesson, you will learn how to:

Refer to ranges

Use collections

Manipulate ranges with the Offset and Resize functions

REFERRING TO A RANGE

Discussion

Procedures will almost certainly need to work with ranges; these ranges will usually differ in size and position in a workbook and hence, there needs to be flexible ways of identifying and selecting them.

The Range Property

As a result, there are many ways of referring to a range. The first and most commonly seen after recording a macro is using the Range property:-

Range(*reference*)

eg. **Range("A1")**

Range("B2 : D10")

An alternative way of selecting a range of cells that can provide more flexibility, is to separate the first cell from the last one with a comma, eg.

Range("A1", "A10")

...refers to the range A1 : A10 and gives the same result as using Range("A1:A10").

Several (non contiguous) ranges can be referred to by typing them into the Range argument with commas separating them and quote marks (") at the beginning and end, eg.

Range("A1:A10, C1:C10, E1:E10")

... refers to three ranges.

Another way of using referring to a range is from a currently selected cell to another fixed point in a worksheet, eg.

Range(ActiveCell, ActiveCell.Offset(10, 0))

This provides rather more flexibility because it will always refer to the range from the active cell to the cell 10 rows below it. So, for example, if the active cell is B4, the range B4 : B14 will be selected.

All the above methods refer to and identify a range object that can then have an appropriate method or property applied to it, eg.

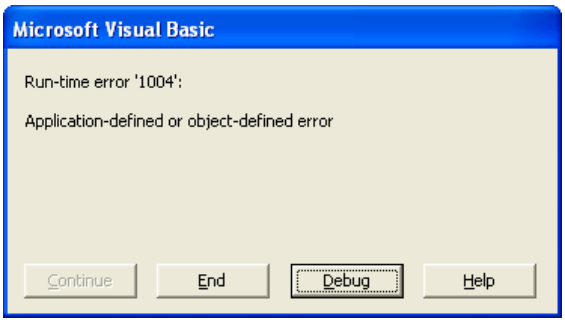
Range("A1", "A10").Interior.ColorIndex = 15

(add a light grey fill colour to the range A1 : A10)

The Cells Property

The most significant difference between using the **Cells** property to the **Range** property, is that the Cells property identifies cells numerically (using what is called an *Index*) AND it can only identify single cells, not ranges.

For example, you cannot use **Cells("A1")** to refer to cell A1 as this would return an error.



A run-time error message

Cells("A1:A10") would return the same error.

The correct way of using the Cells property is to use an *Index* number in the brackets. For example, cell A1 can be referred to in two ways:

Cells(1) or **Cells(1, 1)**

Cells(1) identifies it as cell number 1 of the worksheet, ie. A1. The Index counts along the row before returning to the beginning of the next row and resuming from where it left off. **B1**, therefore, would be **Cells(2)**, **C1** would be **Cells(3)**... etc.

In **Excel 2000 – 03**, the last cell in row 1 (IV1) would be referred to with an index of 256 - Cells(256). A2 would then have the index 257 - Cells(257).

In **Excel 2007**, however, there are 16,384 columns (unless you are running it in "*compatibility mode*"). The last cell in row 1 (XFD1), therefore, would be Cells(16384) and A2 would be Cells(16385).

It makes things easier, therefore, the use the cells property with TWO index numbers, the first being the row number and the second the column number. For example, A2 can be referred to as:

Cells(1, 1) - the cell that is in row 1 and column 1 of the worksheet (A1).

Cells(2, 1) - the cell that is in row 2 and column 1 of the worksheet (A2)

Cells can also be used on its own as a collection (see page 10). It would then refer to ALL the cells on a worksheet or a selection, eg.

ActiveSheet.Cells.Clear or just **Cells.Clear**

...removes formatting and contents from every cell on the current (active) sheet

Sheets(Sheet2").Cells.Font.Name = "Calibri"

... formats all cells on Sheet2 to the Calibri font.

numCels = Selection.Cells.Count

... returns to the variable numCels the number of cells in the selected area

The Cells property is generally more powerful and flexible than the Range property, although Range is still best used when referring to specific (absolute) cells or ranges.

The Cells property and the Range property can be used together as follows:

Range(Cells(1, 1),Cells(3, 3))

This refers to the range A1 : C3, Cells(1, 1) being A1 and Cells(3, 3) being C3.

Procedures

1. Launch or switch to the **VB Editor**.
2. Identify in the **Project Explorer** pane, the workbook (VBA project) that you want to add code to.

3. Open the module sheet containing the code or, insert a new module.
4. Position the cursor in the procedure where you want to write the code.
5. Type the object that you want to refer to.
6. Type a full stop.
7. Type the method or property that you want to apply to the object.
8. Press Enter.

COLLECTIONS

Discussion

It was explained in the previous topic that **Cells** may be used on its own to represent a "collection." Relevant methods and properties can then be applied to a whole collection (or "type") of object. For example:

Cells.Select

...select ALL the cells on the active worksheet

Other collections include:

Object	**Refers to**	**Example**
WorkBooks	All open Excel files	**WorkBooks.Close** *(closes all open Excel files)* **WorkBooks.Save** *(saves all open Excel files)* **WorkBooks.Add** *(creates a new, blank Excel file)*
Sheets or WorkSheets	All the sheets of an active workbook	**Sheets.PrintOut** *(prints all the sheets in the current workbook)* **Sheets.Select** *(groups all the sheets in the current workbook)* **Sheets.Count** *(returns the number of sheets in the active workbook)*

Columns	All the columns of the current worksheet	**Columns.Count** **Columns.ColumnWidth = 20**
Rows	All the rows of the current worksheet	**Rows.Count** **Rows.ColumnWidth = 20**
ChartObjects	All the charts on the current sheet	**ChartObjects.Count** **ChartObjects.Delete** **ChartObjects.Copy**

To identify and refer to a single item in a collection, you normally have to refer to it by its *index* number, or by its name (as a string). For example:

- **WorkBooks(1)** is the first workbook opened (assuming that several are open).
- **WorkBooks("Sales Data.xls")** is specifically the open file named Sales Data.
- **Sheets(1)** is the first sheet from the left of the workbook.
- **Sheets("January")** is specifically the sheet named January.
- **Rows(1)** refers to row **1** of the active worksheet.
- **Columns(1)** is the same as **Columns("A")** and refers to **column A** of the active worksheet.

To select a range of columns, for example from B through G, would require the code Columns("B : G").Select. This is the only way to specify a range of columns within a collection, numbers (eg. Columns(2 : 7) are not allowed.

Procedures

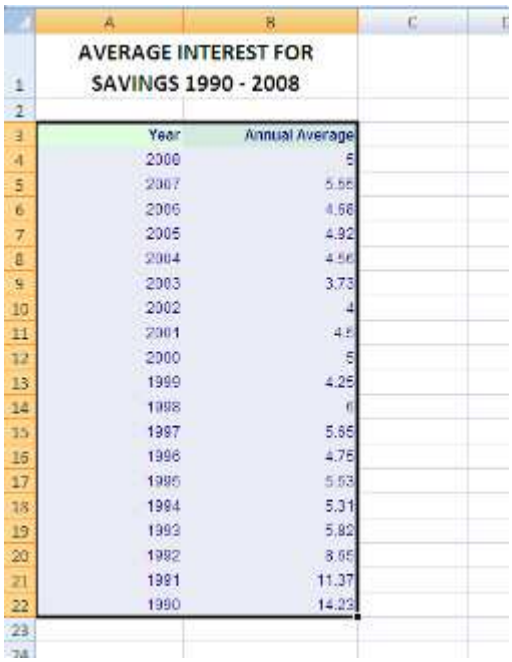
1. Launch or switch to the **VB Editor**.
2. Identify in the **Project Explorer** pane, the workbook (VBA project) that you want to add code to.
3. Open the module sheet containing the code or, insert a new module.
4. Position the cursor in the procedure where you want to write the code.
5. Type the object that you want to refer to.
6. Type a full stop.
7. Type the method or property that you want to apply to the object.
8. Press Enter.

THE CURRENTREGION PROPERTY

Discussion

Referring to a range using CurrentRegion can be very useful when you do not know what the range references are going to be, or how many rows and/ or columns it will have.

The CurrentRegion property identifies a contiguous range on a worksheet (a range that is bounded by a combination of a blank column and a blank row).



	A	B
1	AVERAGE INTEREST FOR SAVINGS 1990 - 2008	
2		
3	Year	Annual Average
4	2008	5
5	2007	5.55
6	2006	4.56
7	2005	4.92
8	2004	4.56
9	2003	3.73
10	2002	4
11	2001	4.5
12	2000	5
13	1999	4.25
14	1998	6
15	1997	5.85
16	1996	4.75
17	1995	5.53
18	1994	5.31
19	1993	5.82
20	1992	8.65
21	1991	11.37
22	1990	14.23
23		

A contiguous range (A3 : B22)

To achieve this, CurrentRegion needs a "starting point" (an expression that returns a range object). The starting point can be any cell on the range. The syntax is:

<*expression*>.CurrentRegion.<*method or property*>

In the picture above, the code for selecting the contiguous range A3 to B22 might be:

Range("A3").CurrentRegion.Select

If a cell is already active on the range, the code could be written as follows:

ActiveCell.CurrentRegion.Select

For example:

Range("A1 : G200").Select

Selection.Copy Destination:= Sheets(1).Range("A2")

... will always select the cells from A1 to G200, copy them and paste them into cell A2 on the second sheet of the workbook.

Nevertheless, if you want to run the same procedure on a different table that does not span from cells A1 to G200, then the procedure will not work correctly. Using the CurrentRegion property, however, it would be successful, eg.

ActiveCell.CurrentRegion.Select

Selection.Copy Destination:= Sheets(1).Range("A2")

The example above would require a cell to be selected on the table first, but would succeed a table containing any number of rows and columns anywhere on the worksheet.

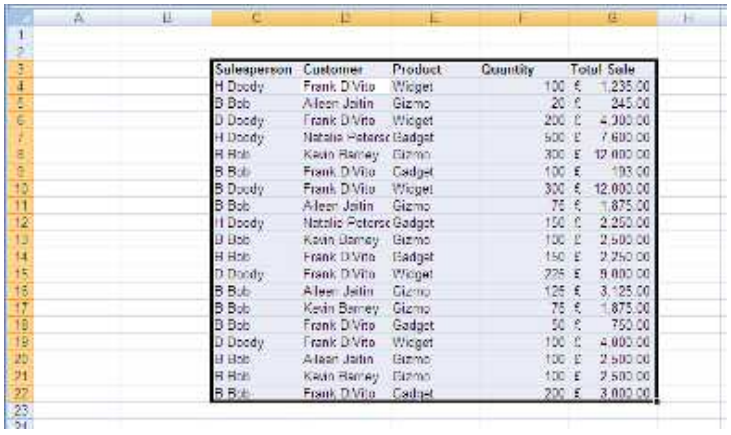
Another useful way of selecting cells CurrentRegion would be to use arguments for it. The following example would select the cell 2 Rows down and 2 Columns to the right, from the first cell in the current region.

ActiveCell.CurrentRegion(2, 2).Select

An alternative method would be to use the activate method. This would be useful if the current region was selected first and needed to be kept selected. Using activate will make a cell active without removing any cell selection.

ActiveCell.CurrentRegion.Select

ActiveCell.CurrentRegion(3, 6).Activate



Salesperson	Customer	Product	Quantity	Total Sale
H Doody	Frank D'Vito	Widget	100	£ 1,235.00
B Bob	Aileen Jatin	Gizmo	20	£ 245.00
D Doody	Frank D'Vito	Widget	200	£ 4,300.00
H Doody	Natalie Peterse	Gadget	500	£ 7,600.00
H Bob	Kevin Barney	Gizmo	300	£ 12,000.00
B Bob	Frank D'Vito	Gadget	100	£ 193.00
B Doody	Frank D'Vito	Widget	300	£ 12,000.00
B Bob	Aileen Jatin	Gizmo	75	£ 1,875.00
H Doody	Natalie Peterse	Gadget	150	£ 2,250.00
B Bob	Kevin Barney	Gizmo	100	£ 2,500.00
H Bob	Frank D'Vito	Gadget	150	£ 2,250.00
D Doody	Frank D'Vito	Widget	225	£ 9,000.00
B Bob	Aileen Jatin	Gizmo	125	£ 3,125.00
B Bob	Kevin Barney	Gizmo	75	£ 1,875.00
B Bob	Frank D'Vito	Gadget	50	£ 750.00
D Doody	Frank D'Vito	Widget	100	£ 4,000.00
B Bob	Aileen Jatin	Gizmo	100	£ 2,500.00
H Bob	Kevin Barney	Gizmo	100	£ 2,500.00
B Bob	Frank D'Vito	Gadget	200	£ 3,000.00

CurrentRegion(2, 2).Activate

Procedures

1. Launch or switch to the **VB Editor**.
2. Identify in the **Project Explorer** pane, the workbook (VBA project) that you want to add code to.
3. Open the module sheet containing the code or, insert a new module.
4. Position the cursor in the procedure where you want to write the code.
5. Type the object that you want to refer to.
6. Type a full stop.
7. Type the method or property that you want to apply to the object.
8. Press Enter.

THE OFFSET & RESIZE PROPERTIES

Discussion

Offset

The Offset property is discussed in the Excel VBA Introduction booklet with respect to it identifying a single cell that is "away" from another one (see Excel VBA Introduction (Student Edition), page 63) . For example:

ActiveCell.Offset(1,0).Select

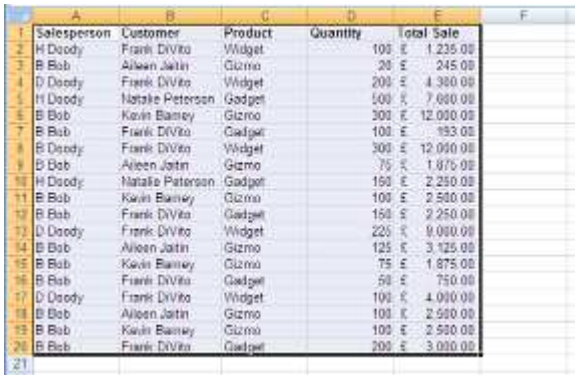
... would select the cell that is 1 row and 0 columns away from the active cell. If the active cell was B10, therefore, Offset(1,0) would refer to B11 (one row down, same column).

The Offset property can also be used to offset an entire range of cells.

The following example moves a selected range 1 cell down and 1 column to the right:

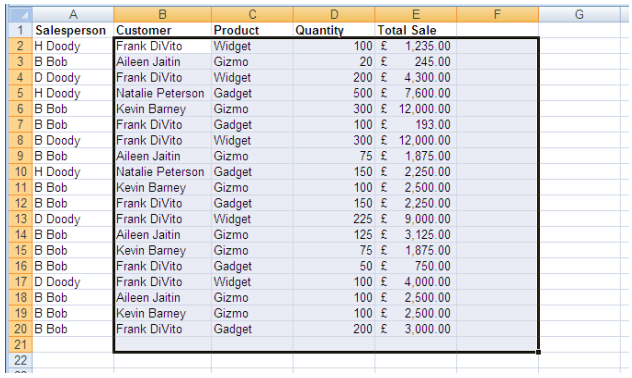
Range("A1").CurrentRegion.Select

Selection.Offset(1,1).Select



	A	B	C	D	E	F
1	Salesperson	Customer	Product	Quantity	Total Sale	
2	H Doody	Frank DiVito	Widget	100	£ 1,235.00	
3	B Bob	Aileen Jaitin	Gizmo	20	£ 245.00	
4	D Doody	Frank DiVito	Widget	200	£ 4,300.00	
5	H Doody	Natalie Peterson	Gadget	500	£ 7,600.00	
6	B Bob	Kevin Barney	Gizmo	300	£ 12,000.00	
7	B Bob	Frank DiVito	Gadget	100	£ 193.00	
8	B Doody	Frank DiVito	Widget	300	£ 12,000.00	
9	B Bob	Aileen Jaitin	Gizmo	75	£ 1,875.00	
10	H Doody	Natalie Peterson	Gadget	150	£ 2,250.00	
11	B Bob	Kevin Barney	Gizmo	100	£ 2,500.00	
12	B Bob	Frank DiVito	Gadget	150	£ 2,250.00	
13	D Doody	Frank DiVito	Widget	225	£ 9,000.00	
14	B Bob	Aileen Jaitin	Gizmo	125	£ 3,125.00	
15	B Bob	Kevin Barney	Gizmo	75	£ 1,875.00	
16	B Bob	Frank DiVito	Gadget	50	£ 750.00	
17	D Doody	Frank DiVito	Widget	100	£ 4,000.00	
18	B Bob	Aileen Jaitin	Gizmo	100	£ 2,500.00	
19	B Bob	Kevin Barney	Gizmo	100	£ 2,500.00	
20	B Bob	Frank DiVito	Gadget	200	£ 3,000.00	
21						

Range("A1").CurrentRegion.Select



	A	B	C	D	E	F	G
1	Salesperson	Customer	Product	Quantity	Total Sale		
2	H Doody	Frank DiVito	Widget	100	£ 1,235.00		
3	B Bob	Aileen Jaitin	Gizmo	20	£ 245.00		
4	D Doody	Frank DiVito	Widget	200	£ 4,300.00		
5	H Doody	Natalie Peterson	Gadget	500	£ 7,600.00		
6	B Bob	Kevin Barney	Gizmo	300	£ 12,000.00		
7	B Bob	Frank DiVito	Gadget	100	£ 193.00		
8	B Doody	Frank DiVito	Widget	300	£ 12,000.00		
9	B Bob	Aileen Jaitin	Gizmo	75	£ 1,875.00		
10	H Doody	Natalie Peterson	Gadget	150	£ 2,250.00		
11	B Bob	Kevin Barney	Gizmo	100	£ 2,500.00		
12	B Bob	Frank DiVito	Gadget	150	£ 2,250.00		
13	D Doody	Frank DiVito	Widget	225	£ 9,000.00		
14	B Bob	Aileen Jaitin	Gizmo	125	£ 3,125.00		
15	B Bob	Kevin Barney	Gizmo	75	£ 1,875.00		
16	B Bob	Frank DiVito	Gadget	50	£ 750.00		
17	D Doody	Frank DiVito	Widget	100	£ 4,000.00		
18	B Bob	Aileen Jaitin	Gizmo	100	£ 2,500.00		
19	B Bob	Kevin Barney	Gizmo	100	£ 2,500.00		
20	B Bob	Frank DiVito	Gadget	200	£ 3,000.00		
21							
22							

Selection.Offset(1,1).Select

This proves useful where a table has to be selected excluding the top row and left column.

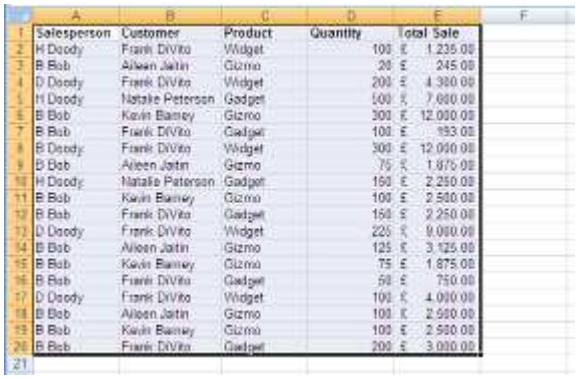
Resize

Using the Resize function enables a selected range of cells to be made bigger or smaller by increasing or decreasing the number of rows and columns that it contains.

The new size of the range is "measured" from its top left corner. In the example below, a current region is selected and then resized to 15 rows and 4 columns.

ActiveCell.CurrentRegion.Select

Selection.Resize(15, 4).Select



	A	B	C	D	E	F
1	Salesperson	Customer	Product	Quantity	Total Sale	
2	H Doody	Frank DiVito	Widget	100	£ 1,235.00	
3	B Bob	Aileen Jaitin	Gizmo	20	£ 245.00	
4	D Doody	Frank DiVito	Widget	200	£ 4,300.00	
5	H Doody	Natalie Peterson	Gadget	500	£ 7,600.00	
6	B Bob	Kevin Barney	Gizmo	300	£ 12,000.00	
7	B Bob	Frank DiVito	Gadget	100	£ 193.00	
8	B Doody	Frank DiVito	Widget	300	£ 12,000.00	
9	B Bob	Aileen Jaitin	Gizmo	75	£ 1,875.00	
10	H Doody	Natalie Peterson	Gadget	150	£ 2,250.00	
11	B Bob	Kevin Barney	Gizmo	100	£ 2,500.00	
12	B Bob	Frank DiVito	Gadget	150	£ 2,250.00	
13	D Doody	Frank DiVito	Widget	225	£ 9,000.00	
14	B Bob	Aileen Jaitin	Gizmo	125	£ 3,125.00	
15	B Bob	Kevin Barney	Gizmo	75	£ 1,875.00	
16	B Bob	Frank DiVito	Gadget	50	£ 750.00	
17	D Doody	Frank DiVito	Widget	100	£ 4,000.00	
18	B Bob	Aileen Jaitin	Gizmo	100	£ 2,500.00	
19	B Bob	Kevin Barney	Gizmo	100	£ 2,500.00	
20	B Bob	Frank DiVito	Gadget	200	£ 3,000.00	
21						

ActiveCell.CurrentRegion.Select

	A	B	C	D	E	F
1	Salesperson	Customer	Product	Quantity	Total Sale	
2	H Doody	Frank DiVito	Widget	100	£ 1,235.00	
3	B Bob	Aileen Jaitin	Gizmo	20	£ 245.00	
4	D Doody	Frank DiVito	Widget	200	£ 4,300.00	
5	H Doody	Natalie Peterson	Gadget	500	£ 7,600.00	
6	B Bob	Kevin Barney	Gizmo	300	£ 12,000.00	
7	B Bob	Frank DiVito	Gadget	100	£ 193.00	
8	B Doody	Frank DiVito	Widget	300	£ 12,000.00	
9	B Bob	Aileen Jaitin	Gizmo	75	£ 1,875.00	
10	H Doody	Natalie Peterson	Gadget	150	£ 2,250.00	
11	B Bob	Kevin Barney	Gizmo	100	£ 2,500.00	
12	B Bob	Frank DiVito	Gadget	150	£ 2,250.00	
13	D Doody	Frank DiVito	Widget	225	£ 9,000.00	
14	B Bob	Aileen Jaitin	Gizmo	125	£ 3,125.00	
15	B Bob	Kevin Barney	Gizmo	75	£ 1,875.00	
16	B Bob	Frank DiVito	Gadget	50	£ 750.00	
17	D Doody	Frank DiVito	Widget	100	£ 4,000.00	
18	B Bob	Aileen Jaitin	Gizmo	100	£ 2,500.00	
19	B Bob	Kevin Barney	Gizmo	100	£ 2,500.00	
20	B Bob	Frank DiVito	Gadget	200	£ 3,000.00	
21						

Selection.Resize(15, 4).Select

The Resize property proves useful where a table has to be selected excluding the bottom row and right column. To achieve this, VBA has to do a bit of work for you! It has to count the number of rows and columns in the table so that a calculation can be performed to determine how many rows and columns to resize the table to.

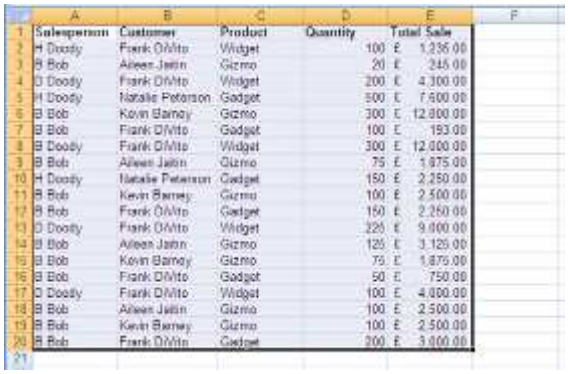
The example below uses variables to count the number of rows and columns in a table (the selection), and then adds 1 to determine the resize parameters:

ActiveCell.CurrentRegion.Select

numRows = Selection.Rows.Count + 1

numCols = Selection.Columns.Count + 1

Selection.Resize(numRows, numCols).Select



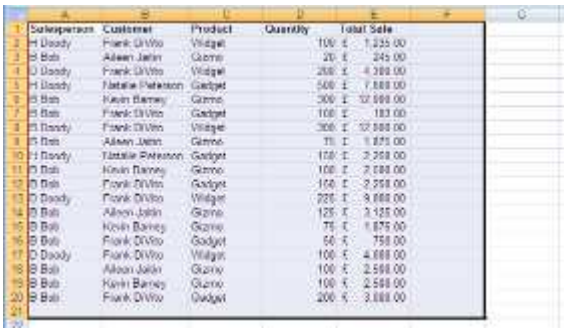
ActiveCell.CurrentRegion.Select

numRows = Selection.Rows.Count + 1

= 20 + 1 = 21

numCols = Selection.Columns.Count + 1

= 5 + 1 = 6



Selection.Resize(numRows,numCols).Select

The example above could also be written without the use of variables, although this may make the code more complicated to write and understand:

ActiveCell.CurrentRegion.Select

Selection.Resize(Selection.Rows.Count + 1, Selection.Columns.Count + 1).Select

The Offset and Resize properties work well together as in the following example. A range is offset by 1 row and 1 column and then resized to 1 row and 1 column less, so that it excludes the blank row and blank column at the bottom and right.

```
ActiveCell.CurrentRegion.Select


numRows = Selection.Rows.Count -1
numCols = Selection.Columns.Count -1


Selection.Offset(1, 1).Resize(numRows, numCols).Select
```

Procedures

1. Launch or switch to the **VB Editor**.
2. Identify in the **Project Explorer** pane, the workbook (VBA project) that you want to add code to.
3. Open the module sheet containing the code or, insert a new module.
4. Position the cursor in the procedure where you want to write the code.
5. Type the object that you want to refer to.
6. Type a full stop.
7. Type the method or property that you want to apply to the object.
8. Press Enter.

EXERCISE

WORKING WITH THE RANGE OBJECT

Task - To practice range object methods and functions.

1. Insert a new module in your *Personal Macro Workbook* and write a sub procedure named, **LayoutTable** that:
   a. selects a whole table of data;
   b. adjusts the width of all the columns to 12pts;
   c. formats the font colour of the first column to blue;
   d. aligns the text centrally in the first row.
2. Assign this macro to a custom button on your **My Macros** toolbar.
3. Open the file, **Ranges**.
4. Run and test **LayoutTable** on the table in **Sheet1** of this workbook.
5. Write another sub procedure in your *Personal Macro Workbook* named, **FormatNumbers** that selects a whole table excluding the top row and the left-most column and formats it to a currency style.
6. Assign this macro to a custom button on your **My Macros** toolbar.
7. Run and test **FormatNumbers** on the table in Sheet 1 of this workbook.
8. Create a final sub procedure in your *Personal Macro Workbook* named, **LastCell** that selects the last cell of a table (the one in the bottom right-hand corner) and:
   a. makes the font size 14pts;
   b. adds a yellow fill colour; and
   c. autofit the column.
9. Assign this macro to a custom button on your **My Macros** toolbar.
10. Run and test **LastCell** on the table in Sheet 1 of this workbook.
11. Create a new sub procedure in your *Personal Macro Workbook* named, **RunAllMacros**.
12. Call into this sub procedure the three macros created above.
13. Assign this macro to a custom button on your **My Macros** toolbar.
14. Run and test **RunAllMacros** on the table in **Sheet2** of this workbook.
15. Save and close the file.
16. Exit **Excel** to ensure that the *Personal Macro Workbook* is saved.
17. Re-launch Excel.

LESSON 2 - VARIABLES, CONSTANTS AND ARRAYS

In this lesson, you will learn how to:

Declare Variables

Understand Scope and Visibility

Define Constants

Declare Arrays

DECLARING VARIABLES

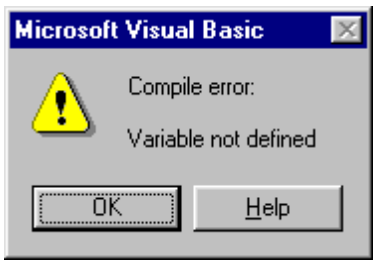
Discussion

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 VBA, 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 a variable spelled incorrectly during code writing would not be recognised by Visual Basic.

This problem can be avoided by choosing to declare explicitly every variable. This tells Visual Basic that every variable will be declared in advance and any others used, misspelt or not, will be incorrect. When Visual Basic encounters an undeclared variable, the following message is displayed:



Error Message – Variable not defined

To explicitly declare variables, the following statement is required at the top of the Visual Basic module:

Option Explicit

Variables are then declared by using the Dim Statement.

Dim variable name

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

The **Option Explicit Statement** can be set automatically to appear in all modules.

Procedures

1. In the Visual Basic Editor, Select **Tools... Options**.
2. Select the **Editor** Tab.
3. Select **Require Variable Declaration**.
4. Click on the **OK** button.

The Option Explicit Statement is added to new modules NOT existing modules.

SCOPE AND VISIBILITY

Discussion

It is sometimes a necessity to "scope" a variable correctly, when calling procedures from within a module or indeed across several modules. The result of scoping these variables correctly would signify whether or not the variable has maintained its data.

If the data has been lost then, this is known as lost visibility.

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 module in the workbook. 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 at module-level if local is sufficient.

Procedures

1. Launch or switch to the **VB Editor**.
2. Identify the workbook (VBA project) to which you want to add code in the **Project Explorer** pane.
3. Open the module sheet containing the code or, insert a new module.
4. To declare a variable at "***module level,***" position the cursor at the very top of the module sheet and type **Dim**.

   To declare a "***public***," variable, position the cursor at the very top of the module sheet and type **Public**.
5. Type a **space**.
6. Type the **name** for the variable.
7. Type a **space**.
8. Type **As**.
9. Type a **space**.
10. Type or select from the list and appropriate **data type** for the variable.
11. Press **Enter**.

CONSTANTS

Discussion

A constant is a named item that retains a constant value throughout the execution of a program, as opposed to a variable, whose value may change during execution.

By storing a value as a constant, it indicates to anyone reading or having to edit the procedure, that wherever you see the constant being used, its value will always be as assigned in the **Const** statement. Like many things in programming, it is an example of good practice and intended to keep the code neat, concise and easily understood.

It is usual to type your constants all UPPER CASE to distinguish them from VBA words (mixture of upper and lowercase) and variables (all lower case).

Constants are defined using the **Const** statement. Constants can be used anywhere in a 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 Font_Colour()
Const RED = 3
Const BLUE = 5
Const YELLOW = 6
```

Can then be used thus.....

```
    Selection.Font.ColorIndex = RED
    Selection.Interior.ColorIndex = YELLOW
    Selection.BorderAround ColorIndex:= BLUE

End Sub
```

Like variables, constants should be declared in order to inform, prevent errors and manage computer memory. Constants are not, however, "dimmed" in the same way as variables. They are declared as type immediately after the name of the constant. In the example below, a constant has been created to store the value of pi (π).

```
Const PI as Single = 3.142
```

In the example further up the page, the constants would be declared as follows:

```
Const RED as Byte = 3
Const BLUE as Byte = 5
Const YELLOW as Byte = 6
```

Or, in the single line form:

```
Const RED As Byte = 3, YELLOW As Byte = 6, BLUE As Byte = 5
```

Procedures

1. Launch or switch to the **VB Editor**.
2. Identify the workbook (VBA project) to which you want to add code in the **Project Explorer** pane.
3. Open the module sheet containing the code or, insert a new module.
4. Identify the procedure that you wish to declare the constant in or, start typing a new procedure.
5. Position the cursor immediately below the **Sub** statement at the top of the procedure.
6. Type **Const**.
7. Type a **space**.
8. Type a **name** for the constant. *Tip*: constant names are normally written all UPPER CASE.
9. Type a **space**.
10. Type **As**.
11. Type or select from the list a **data type** for the variable.
12. Type a **space**.
13. Type **=.**
14. Type a **value** for the constant.
15. Press **Enter**.

ARRAYS

Discussion

An array is a special type of variable that can contain many "elements" of information simultaneously - in essence a large variable. Think of it as a box containing many separate sections, rather like pigeon holes in a post room.

An array stores large numbers of values more efficiently that using an equivalent numbers of individual variables. Not only does an array make better use of computer memory, but it can be "filled" extremely quickly and flexibly with a minimum amount of code. Storing a column of 100 numbers in individual variables would take 100 lines of code. With an array, it can be done with just three!

DECLARING AN ARRAY

Discussion

An array must be declared before it can be used. Like a variable, it is declared with a **Dim** statement, followed by the name of the array. What makes an array different is that after the name, there are brackets containing a number (or numbers). This number (or numbers) denote how many elements the array contains. This is referred to as the *dimension* of the array. For example, the following example declares an array containing 5 elements:

Dim arrData (4)

You may now be asking the question: "Why 4 when the array must contain 5 elements?"

The number 4 is used because it is the *upper bound* of the array that is used in the brackets, NOT the number of elements required. Because the lower bound is 0, an upper bound of 4 does, therefore, indicate that the array contains 5 elements.

arrData(0)	arrData(1)	arrData(2)	arrData(3)	arrData(4)

Also like variables, it is good practice to declare an array *as type*. If an array is declared with a specific data type (**As Single**, for example), then every element in the array must be of that type. It is possible, however, to declare an array as a variant, in which case the elements could contain different data types.

Dim variable (*dimensions*) as *Type*

Assuming the above array (arrData) will be storing large numbers involving decimals, the array would be declared as follows:

Dim arrData(4) As Single

Earlier, it was explained that all elements in an array must be of the same data type. You can get around this restriction by declaring your array as a variant. A variant array, however, will consume more memory than other types.

The same data types as for variables can be used. These are given in the table below:

Data type	Memory size	Storage capability
Byte	1 byte	0 to 255
Boolean	2 bytes	True or False
Integer	2 bytes	-32,768 to 32,767
Long	4 bytes	-2,147,483,648 to 2,147,483,647
Single	4 bytes	-3.402823E38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values
Double	8 bytes	-1.79769313486231E308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values
Currency	8 bytes	-922,337,203,685,477.5808 to 922,337,203,685,477.5807
Decimal	14 bytes	+/ -79,228,162,514,264,337,593,543,950,335 with no decimal point; +/ -7.9228162514264337593543950335 with 28 places to the right of the decimal; smallest non-zero number is +/ -0.0000000000000000000000000001
Date	8 bytes	January 1, 100 to December 31, 9999
String (variable-length)	10 bytes + string length	0 to approximately 2 billion

Declaring an array actually reserves memory for the entire array. VBA does not care whether you fill it up or not, so be conservative when defining array elements.

Procedures

1. Launch or switch to the **VB Editor**.
2. Identify the workbook (VBA project) to which you want to add code in the **Project Explorer** pane.
3. Open the module sheet containing the code or, insert a new module.
4. Identify the procedure that you wish to declare the array in or, start typing a new procedure.
5. Position the cursor immediately below the **Sub** statement at the top of the procedure.
6. Type **Dim**.
7. Type **space**.
8. Type a name for the array.
9. Type a space.
10. Type an opening bracket (.
11. Type the ***upper bound*** for the array, (this will be a single number).
12. Type a closing bracket).
13. Type a **space**.
14. Type **As**.
15. Type or select from the list an appropriate **data type** for the array.
16. Press **Enter**.

ARRAY TYPES

Discussion

Changing the lower bound

The array described above is referred to as a *zero-based array* because the lower bound is 0.

An array, however, can have ANY number as its lower bound, most usually a 1. This makes it easier to use the array because it is more natural to the human mind to start counting from 1 rather than 0!

To create an array with a lower bound that is NOT 0 you have to declare it as follows:

Dim arrData (1 To 5) As *Type*

The array above would contain 5 elements numbered from 1 to 5.

arrData(1)	arraData(2)	arrData(3)	arrData(4)	arrData(5)

If you wish ALL arrays on a module sheet to use 1 as the lower index, you can type the words:

Option Base 1

...at the top of the module sheet (a module-level declaration). This makes is more convenient because when it comes to declaring arrays in the module's procedures, you can omit the **1 To** part of the declaration. Hence, an array containing 5 elements would be declared as:

Dim arrData (5)

The **Option Base 1** statement at the top of the module sheet indicates that the lower bound is 1. The following extract from the VB Editor clarified the code.

```
Option Base 1

Sub TransferData()

Dim myArray(5) As String ' declares a five element array for storing text

' fills the  elements 1 to 5

myArray(1) = "Monday"
myArray(2) = "Tuesday"
myArray(3) = "Wednesday"
myArray(4) = "Thursday"
myArray(5) = "Friday"

'Returns the array to range A1 : E1 on Sheet1

Sheets("Sheet1").Range("A1:E1") = myArray


End Sub
```

Procedures

1. Launch or switch to the **VB Editor**.
2. Identify the workbook (VBA project) to which you want to add code in the **Project Explorer** pane.
3. Open the module sheet containing the code or, insert a new module.
4. Identify the procedure that you wish to declare the array in or, start typing a new procedure.
5. Position the cursor immediately below the **Sub** statement at the top of the procedure.
6. Type **Dim**.
7. Type **space**.
8. Type a name for the array.
9. Type a space.
10. Type an opening bracket (.
11. Type the *lower bound* (this will be a single number).
12. Type a space.
13. Type To.
14. Type the *upper bound* (this will be a single number)
15. Type a closing bracket).
16. Type a **space**.
17. Type **As**.
18. Type or select from the list an appropriate **data type** for the array.
19. Press **Enter**.

Multi-dimensional arrays

The arrays described in the previous topics are *one-dimensional* arrays. A useful analogy is to imagine the elements in one single row.

Arrays can have up to 60 dimensions, although 1, 2 or 3 are normally used. A 2-dimensional array, for example, is useful for storing values from a worksheet that are in columns and rows.

A 2-dimensional array is created by declaring it as follows:

Dim arrData (3 , 2) As *Type*

Or,

Dim arrData (1 To 4 , 1 To 3) As *Type*

Or, if **Option Base 1** is used

Dim arrData (4 , 3) As *Type*

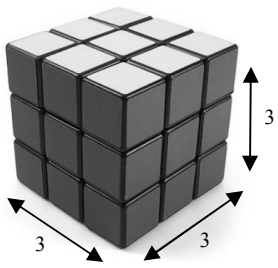
Any of the above would create an array that is 4 elements tall and 3 elements wide. The example below fills the array with values from the range A1 to C4.

```
Dim arrData(1 To 4, 1 To 3) As Single

arrData(1, 1) = Range("A1").Value
arrData(1, 2) = Range("B1").Value
arrData(1, 3) = Range("C1").Value
arrData(2, 1) = Range("A2").Value
arrData(2, 2) = Range("B2").Value
arrData(2, 3) = Range("C2").Value
arrData(3, 1) = Range("A3").Value
arrData(3, 2) = Range("B3").Value
arrData(3, 3) = Range("C3").Value
arrData(4, 1) = Range("A4").Value
arrData(4, 2) = Range("B4").Value
arrData(4, 3) = Range("C4").Value
```

A 3-Dimensional array could be declared as follows:

Dim arrData (1 To 3 , 1 To 2 , 1 To 4) As *Type*

Think of this array as a cube that has depth as well as height and width.



To use this array to store values, you would need to refer to all three dimensions, for example:

ArrData(1 , 1 , 1) = ...

ArrData(1 , 1 , 2) = ...

ArrData(1 , 1 , 3) = ...

ArrData(1 , 1 , 4) = ...

ArrData(1 , 2 , 1) = ...

ArrData(1 , 2 , 2) = ...

ArrData(1 , 2 , 3) = ...

ArrData(1 , 2 , 4) = ...

ArrData(2, 1 , 1) = ...

ArrData(2, 1 , 2) = ...

.....etc.

Procedures

1. Launch or switch to the **VB Editor**.
2. Identify the workbook (VBA project) to which you want to add code in the **Project Explorer** pane.
3. Open the module sheet containing the code or, insert a new module.
4. Identify the procedure that you wish to declare the multidimensional array in or, start typing a new procedure.
5. Position the cursor immediately below the **Sub** statement at the top of the procedure.
6. Type **Dim**.
7. Type a **space**.
8. Type a **name** for the array.
9. Type a **space**.
10. Type an opening bracket (.
11. Type the *upper bound* of the first dimension of the array (this will be a single number).
12. Type a **comma** .
13. Type the *upper bound* of the second dimension of the array (this will be a single number)

14. Continue typing commas and the upper bound of any additional dimensions that you want your array to have.
15. Type a closing bracket).
16. Press **Enter**.
17. Type a **space**.
18. Type **As**.
19. Type a **space**.
20. Type and appropriate **data type** for the array.
21. Press **Enter**.

Keep in mind that an array should simplify a process, not complicate it. Arrays can become very complex, so if the array seems to take on a life of its own, you might want to rethink your strategy before things get too complicated!

ASSIGNING VALUES TO AN ARRAY

Discussion

As described above, an array stores its values in elements (separate sections of the box). Each element has a numeric "address" (**Index**) to identify where in the array it is stored . A simple analogy would be to imagine a tall building (the array) where each floor is identified by a number (the element *index*).

Note that the lowest element is not referred to as floor (1) but floor (0). By default, arrays number their elements starting with 0 NOT 1. The lowest index of an array (usually number 0) is known as the *lower bound*. The highest index of the array is known as the *upper bound*.

As with a variable, creating an array starts with giving it a name. The naming rules for variables apply equally to arrays and are given below:

- Single characters (eg. a, b, c, x, y, z) should be avoided unless the array if being used for simple counting purposes. Using this type of name can cause ambiguity and it is better to use descriptive, but concise, names for your array, eg: **newnames**, **salesfigs**, **startdates**, **numcells**.

- An array name must start with a letter and not a number. Numbers can be included within the name, but not as the first character, eg. **salesfigs1** but not **1salesfigs**.
- The first character of an array name should be left lowercase. Because VBA words always start with an uppercase character (eg. ActiveCell, WorkBook, Cells, etc), keeping the first character lowercase helps make your arrays (like variables) stand out more clearly in the sub procedure.
- Avoid using names that conflict with VBA words such as: "activecell", "sheets", "font", "cells", etc.
- Spaces cannot be used in array names. You can separate words by either capitalisation, eg. **newNames**, or by using the underscore character, eg. **new_names**.
- Most punctuation marks (eg. , . : ; ? -) and special characters (eg. $, %, ^, &, #, }) cannot be used.
- An array name can be no longer than 250 characters.

It is common practice to commence a variable name with the prefix – **arr**.

The example below shows how to create an array named **arrData** that stores the values in the range B11:B16 in its elements numbered from 0 to 5.

```
arrData(0) = Range("B11").Value
arrData(1) = Range("B12").Value
arrData(2) = Range("B13").Value
arrData(3) = Range("B14").Value
arrData(4) = Range("B15").Value
arrData(5) = Range("B16").Value
```

arrData(0) through to **arrData(5)** can be used like normal variables. The following example returns the contents of the above array into the range(D10:D15) of Sheet2.

```
Sheets("Sheet2").Range("D10").Value = arrData(0)
Sheets("Sheet2").Range("D11").Value = arrData(1)
Sheets("Sheet2").Range("D12").Value = arrData(2)
Sheets("Sheet2").Range("D13").Value = arrData(3)
Sheets("Sheet2").Range("D14").Value = arrData(4)
Sheets("Sheet2").Range("D15").Value = arrData(5)
```

Procedures

1. Launch or switch to the **VB Editor**.
2. Identify the workbook (VBA project) to which you want to add code in the **Project Explorer** pane.
3. Open the module sheet containing the code or, insert a new module.
4. Identify the procedure that you wish to declare the constant in or, start typing a new procedure.
5. Position the cursor in the procedure where you want to assign values to the array.
6. Type the **name** of the array.
7. Type an opening bracket **(**.
8. Type the **index number** of the array element where you want to store the value (this will be a single number).
9. Type a closing bracket **)**.
10. Type a **space**.
11. Type **=.**
12. Type a **space**.
13. Type the **value** that you want to store.
14. Press **Enter**.

FILLING ARRAYS USING LOOPS

Discussion

In the examples above, the elements in the array were filled with values individually (eg. arrData(1) = Range("A1").Value). This would be very time consuming where many values are concerned, and lacking in flexibility.

A common method, therefore, of filling an array is by using a **For... Next** loop. The example below fills an array with values from the range A1 to A10. By referring to the range with the **Cells** property, and making use of the **For** loop variable, the array can be loaded in a fraction of a second.

```
Dim arrData(1 To 10) As Integer
Dim i as Byte
For i = 1 To 10
  arrData(i) = Cells(i, 1).Value
  Next i
```

The value in each element of the array can then be manipulated as necessary.

In the example below, the array, **arrData**, is filled with values from range A1 to A10 of Sheet1. The values are then returned to range A1 to J1 of Sheet2 with an additional 15% added.

```
Dim arrData(1 To 10) As Integer
Dim i As Byte
For i = 1 To 10
   arrData(i) = Sheets("Sheet1").Cells(i, 1).Value
Next i
For i = 1 To 10
   Sheets("Sheet2").Cells(1, i).Value = arrData(i) * 1.15
Next i
```

The following example uses a nested For... Next loop to fill a 2-dimensional array with values from the range A1 to C3.

```
Dim arrData(1 To 3, 1 To 4) As Integer
Dim i As Byte
Dim j As Byte

For i = 1 To 3
   For j = 1 To 4
         arrData(i, j) = Cells(i, j).Value
   Next j
Next i
```

Procedures

1. Launch or switch to the **VB Editor**.
2. Identify the workbook (VBA project) to which you want to add code in the **Project Explorer** pane.
3. Open the module sheet containing the code or, insert a new module.
4. Identify the procedure that you wish to declare the constant in or, start typing a new procedure.
5. Position the cursor in the procedure where you want to assign values to the array.

6. Type **For**.
7. Type a **space**.
8. Type a **variable name** for the For loop (eg. i).
9. Type **=**.
10. Type the lower bound of the array that you want the loop to fill.
11. Type a space.
12. Type To.
13. Type a space.
14. Type the upper bound of the array that you want the loop to fill.
15. Press Enter.
16. Type the **name** of the array.
17. Type an opening bracket (.
18. Type the **variable name** that you have used for the loop.
19. Type a closing bracket).
20. Type a **space**.
21. Type **=.**
22. Type a **space**.
23. Type the **value** that you want to store.
24. Press **Enter**.
25. Type **Next**.
26. Type a **space**.
27. Type the **variable name** for the loop.
28. Press **Enter**.

DYNAMIC ARRAYS

Discussion

In earlier examples, we have known what size to make the array in advance, and hence, been able to create the appropriate number of elements and/ or dimensions. This is known as a *fixed-size* array.

Sometimes, you may not know how many elements you need in an array until it has been calculated out, usually in the form on a variable. In these cases, you declare a dynamic array by creating a name for the array, adding brackets but leaving them empty, eg.

Dim arrData() As *Type*

Once the number of elements and dimensions for the array have been determined, it can be declared again but using the **Redim** statement. In the following example, a dynamic array is declared at the beginning of the sub procedure. The number of elements required is then calculated out and the array is redimensioned to the correct size.

Dim arrData()

(declares the dynamic array)

numCels = Selection.Cells.Count

(counts the number of cells in a selected area of the worksheet)

ReDim arrData (1 To numCels) As Integer

(creates an array with the same number of elements as there are cells in the selected area)

A dynamic array can be redimensioned as many times as necessary, but you cannot change its data type from how you originally declared it. For example, the code below would give an error message:

Dim arrData() As String

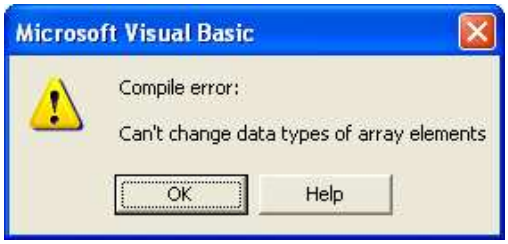
(declare dynamic array)

var1 = Selection.Cells.Count

(calculate size of array)

ReDim arrData(1 To var1) as Integer

(redim array as different data type)



Error Message

If all the elements in a dynamic array have been filled with values, and you need to make it bigger, you can redim it again. Beware, however, that you may lose all the original values in it. To avoid this happening you must use the **Preserve** keyword after **ReDim**.

In the example below, the dynamic, 1-dimensional array **arrData(1 To 4)** has been already been filled with values. It is then redimmed to make it bigger by two extra elements (1 To 5). The keyword **Preserve** is used in the **ReDim** statement so that the values in (1 To 3) are not lost.

```
Dim arrData() as String
(declare dynamic array)

ReDim arrData (1 To 3)
(redimension array to contain 3 elements)
arrData(1) = "Tom"
arrData(2) = "Harry"
arrData(3) = "Joe"
(fill array)

Redim Preserve arrData (1 To 5)
(redeclare array to make it bigger but retain its existing values)
arrData(4) = "Sally"
arrData(5) = "Jane"
(fill additional elements with values)
```

If **Preserve** had not been used, **arrData(4)** and **arrData(5)** would have successfully stored Sally and Jane, but Tom, Harry and Joe would have been lost from **arrData(1)**, **(2)** and **(3)**.

Once you have finished using a dynamic array, you can use a ReDim statement to reclaim the memory your array consumed, eg **ReDim arrData(0 , 0)** or **Erase arrData**.

Procedures

1. Launch or switch to the **VB Editor**.

2. Identify the workbook (VBA project) to which you want to add code in the **Project Explorer** pane.
3. Open the module sheet containing the code or, insert a new module.
4. Identify the procedure that you wish to declare the constant in or, start typing a new procedure.
5. Position the cursor immediately below the **Sub** statement at the top of the procedure.
6. Type **Dim**.
7. Type a **space**.
8. Type a **name** for the array.
9. Type a **space**.
10. Type an opening bracket **(**.
11. Type a closing bracket **)**.
12. Press **Enter**.
13. Position the cursor further down the code where you want to dimension the array.
14. Type **ReDim**.
15. Type an opening bracket **(**.
16. Type the dimension(s) for the array
17. Type a closing bracket **)**.
18. Press **Enter**.

THE ARRAY FUNCTION

Discussion

The Array function is an easy way to create and fill an array with values. Type all the values that you want to store into the function argument separated by commas and it will return a simple, 1-dimensional array that you can use in the same order as the values were typed in.

- The first element of the Array function always has an index of 0, irrespective of any Option Base statement.
- The Array function always returns an array of data type Variant.

Example 1 – Create array from a list text strings

```
Dim arrWkDays () as Variant
arrWkDays = Array("Mon","Tue","Wed","Thu",Fri")
```

Example 2 – using numbers as arguments

```
Dim arrTens as Variant
arrTens = Array(10, 20, 30, 40, 50, 60, 70, 80, 90)
```

Returning values from the Array function

As for arrays in general, the name of the array followed by the required element index number in brackets, eg.

In example 1, **Range("A1").Value = arrWkDays(0)** would return **Mon**.

In example 2, **Range("A1").Value = arrTens(4)** would return **50**.

Procedures

1. Launch or switch to the **VB Editor**.
2. Identify in the **Project Explorer** pane, the workbook (VBA project) that you want to add code to.
3. Open the module sheet containing the code or, insert a new module.
4. Position the cursor immediately below the **Sub** statement at the top of the procedure.
5. Type **Dim**.
6. Type a **space**.
7. Type a **name** for the array.
8. Type a **space**.
9. Type **As**.
10. Type a **space**.
11. Type **Variant**.
12. Press **Enter**.

13. Position the cursor where you want to assign values to the array function.
14. Type the **name** of the array as previously declared.
15. Type **=**.
16. Type **Array**.
17. Type an opening bracket **(**.
18. Type the **value(s)** that you want the array to store, separating each one with a **comma**.
19. Type a closing bracket **)**.
20. Press **Enter**.

EXERCISE

VARIABLES AND ARRAYS

Task 1: Write a procedure that prompts the user to enter data, using variables and arrays.

1. Open the file **Forecast**. This workbook already contains a macro named **CreateNewForecast** that makes a copy of the sheet - **Template** - at the end of the workbook.
2. Add to this sub procedure, VBA code that:
    a. prompts the user by means of an **Input Box**, for a month name to give to the sheet. This value must be stored in a locally declared variable, and then used to rename the sheet.
    b. prompts the user by means of four **Input Boxes**, for some values to enter into cells B2, C2, D2 and E2. These values should be stored in a locally declared, 1-dimensional array and then entered into the appropriate cells.

    (Tip: Try using **FOR NEXT** loops for this purpose)
3. Assign the sub procedure to a worksheet button on the Template sheet and then run and test it using some made-up data of your own. Correct code if necessary.
4. Save and close the file.

Task 2: Use constants for storing data.

1. Insert a new module sheet into your **Personal Macro Workbook**.
2. Declare at module level, a constant named, **PURPLE** and assign to it the value, 29.
3. Declare at module level, a constant named, **ORANGE** and assign to it the value, 45.
4. Declare at module level, a constant named, **ROSE** and assign to it the value, 38.
5. Declare at module level, a constant named, **BLUE** and assign to it the value, 5.
6. Using the above constants, write a procedure named, **ApplyColours** that formats:
    a. the first row of a table with a purple background and a rose font;
    b. the first column of a table with an orange background and blue font.

7. Open the file, **Sales Analysis**.
8. Assign the macro created above to a custom button on your **My Macros** Toolbar.
9. Run and test the sub procedure on the first two sheets of this workbook. Correct code if necessary.
10. Save and close the file.

Task 3: Use a 2-dimensional array for storing data.

1.	Open the file, **Sales Analysis**.
2.	Switch to the VB Editor and insert a module sheet into this workbook.
3.	Write a sub procedure named, **TransferData** that uses a 2-dimensional array to store the data in range **B2** to **E7** of the sheet named **Gross**.
4.	Add to the procedure additional code that returns the data discounted by 20% to the same range on the sheet named **Net**.
5.	Assign this macro to a worksheet button on the **Gross** sheet.
6.	Run and test the macro again. Correct any errors.
7.	Save and close the file.

LESSON 3 - USER DEFINED FUNCTIONS

In this lesson, you will learn how to:

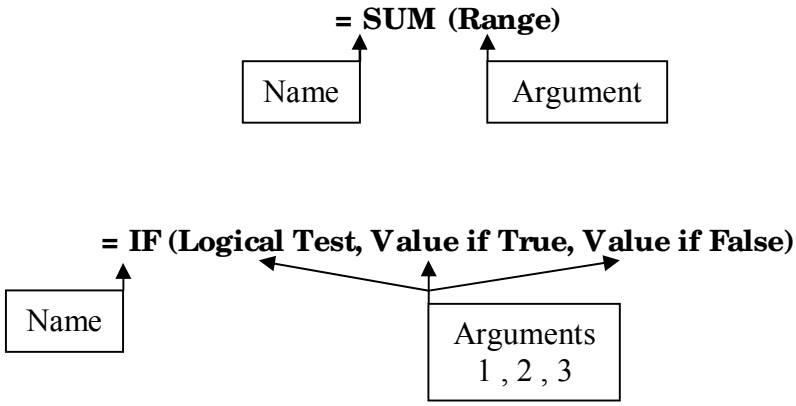
- Use built-in Excel functions in a sub procedure
- Use VBA functions in a procedure
- Create and use custom function procedures

USING EXCEL WORKSHEET FUNCTIONS IN VBA

Discussion

Excel comes with many built-in functions as standard. In Excel 2007, the number is approximately 350, although in previous versions it was significantly less. Each function is designed to carry out a calculation by being given just a few pieces of information. In some cases, the calculation performed is difficult to do by other methods, but in many cases, the calculation performs a very simple or menial task, but with greater ease and economy of time for the user.

Probably the best know Excel function is the "SUM" function. The sum function is designed to add up numerical values in cells. All functions come in two parts, a name and argument(s), for example:



You can use many built-in Excel functions in sub procedures. In Excel 2007, the number is approximately 280, but earlier versions had significantly less. The syntax is as follows:

Application.WorkSheetFunction.<*function name*> (< *argument(s)*>)

To help you indentify available functions, the *Auto Members List* displays a list of them after typing the dot (.) following **WorksheetFunction**.

Members list of Excel worksheet functions

The following statement returns into cell G10, the sum of the values in cells G2 to G9.

Range("G10").Value = Application.WorkSheetFunction.Sum(Range("G2:G9"))

Note that the argument for the Sum function is not expressed in the usual Excel way, (eg. G2:G9), but as a valid VBA reference to G2:G9, ie. **Range("G2:G9")**.

Alternatively, **Selection** can be used as the argument. The following example selects the range G2:G9 and returns the SUM result of the selection into cell G10.

```
Range("G2:G9").Select
Range("G10").Value = Application.WorkSheetFunction.Sum(Selection)
```

The statement can be shortened by omitting **WorksheetFunction**. This is fine if you know the name of the function that you wish to use, because it will not produce the relevant *Auto Members List* to help you.

```
Range("G10").Formula = Application.Sum(Range("G2:G9"))
```

Note that both examples above do NOT write the actual formula into the cell. The calculation is performed in the sub procedure, and the result only is returned to the cell.

To show the SUM function in the cell, the following code has to be used:

```
Range("G10").Formula = "=Sum(G2:G9)"
```

This literally writes the formula into the cell as a string. Excel then takes over and carries out the calculation.

For a full list of VBA functions (Excel 2003 but valid for other versions too) can be found at the following Web address:

http:/ / msdn.microsoft.com/ en-us/ library/ aa221655(office.11).aspx

Procedures

1. Launch or switch to the **VB Editor**.
2. Identify in the **Project Explorer** pane, the workbook (VBA project) that you want to add code to.
3. Open the module sheet containing the code or, insert a new module.
4. Position the cursor in the procedure where you want to use the workbook function.
5. Type the object to which you want the worksheet function to refer to, eg. **ActiveCell**
6. Type a **full stop**.
7. Type **Value**.

8. Type **=**.
9. Type **Application**
10. Type a **full stop**.
11. Type **WorkSheetFunction**.
12. Type a **full stop**.
13. Type the **name of the function** that you want to use or, select it from the list.
14. Type an opening bracket **(**.
15. Type the **argument(s)** for the function (this has to be in a VBA form, eg. **Range("A1:A5")** NOT **A1:A5**).
16. Type a closing bracket **)**.
17. Press **Enter**.

VBA FUNCTIONS

Discussion

In addition to the Excel worksheet functions, there are approximately 92 VBA functions (in Excel 2007) that can be used in sub procedures. In many cases, they perform equivalent calculations to the Excel worksheet functions, but in some cases augment them.

In some cases, where there is a VBA function to carry out a calculation, the equivalent Excel worksheet function cannot be used, and will not appear in the *Auto Members List* following **WorksheetFunction.**.

Examples of VBA functions that have equivalent Excel worksheet functions, but which **are not** written the same are:

VBA Function	**Excel Worksheet Equivalent**	**Description**
Date	=TODAY()	Returns the current system date.
Rnd	=RAND()	Returns a random number greater than or equal to 0 and less than 1.
UCase(*string*)	=UPPER(*text*)	Converts text to upper case.
LCase(*string*)	=LOWER(*text*)	Converts text to lower case.
IsNumeric(*expression*)	=ISNUMBER(*value*)	Checks whether a value is a number and return TRUE or FALSE

Examples of VBA functions that have equivalent Excel worksheet functions, and which **are** written the same are:

VBA Function	Excel Worksheet Equivalent	Description
Round(*expression,digits*)	=ROUND(*number,digits*)	Rounds a number to the closest number of digits, eg. =ROUND(34.567,1) displays 34.6. =ROUND(34567,-3) displays 35000.
Int(*expression*)	=INT(*number*)	Returns the whole part of a number only, eg. =INT(34.567) displays 34 and discards the decimals.
Abs(*number*)	=ABS(*number*)	Returns a number without its sign (ie. a negative becomes a positive; a positive remains a positive).
Day(*date*)	=DAY(*serial number*)	Returns the day that a date falls on.
Month(*date*)	=MONTH(*serial number*)	Returns the month that a date falls on.
Year (*date*)	=YEAR(*serial number*)	Returns the year that a date falls on.
Now	=NOW()	Return the current system date AND time.
Pmt(*rate,periods,pv*)	=PMT(*rate,nper,pv*)	Calculates the repayments for a loan.
Fv(*rate,periods,payment*)	=FV(*rate,nper,pmt*)	Calculates the future value of an investment.

There follow some examples of how some of these VBA functions might be used:

Range("A1").Value = Date

(*Displays the current date in cell A1. The date is linked to the system clock so will be updated automatically.*)

ActiveCell.Value = Int(Rnd * 50)

(Displays in the currently selected cell a random whole number between 0 and 49 (inclusive))

Cells(2 , 2).Value = UCase("vba is great fun")

(*Displays VBA IS GREAT FUN in cell B2*)

ActiveCell.Offset(0 , 1).Value = Year(ActiveCell.Value)

(*Assuming the active cell contains the date 15/05/2009, the cell immediately to its right will display 2009*)

If IsNumeric(ActiveCell.Value) Then ActiveCell.Clear

(*Deletes the content of an active cell if it is a number*)

Note that functions existing in VBA only, can only be used to return a value in the sub procedure that that they are used in. For example, if you type directly into the cell of a worksheet:

Range("E2").Value = "=UCASE(D2)" (where cell D2 contains the text, *hello world*)

..... you would get the #NAME? error. This is because although UCase works in a VBA sub procedure, it is not recognised by Excel as a valid worksheet function. The correct worksheet function to use is: **= UPPER(*text*)**.

Procedures

1. Launch or switch to the **VB Editor**.
2. Identify in the **Project Explorer** pane, the workbook (VBA project) that you want to add code to.
3. Open the module sheet containing the code or, insert a new module.
4. Position the cursor in the procedure where you want to use the VBA function.
5. Type the object to which you want the function to refer, eg. **ActiveCell**
6. Type a **full stop**.
7. Type **Value**.
8. Type **=**.
9. Type the **VBA function name**.
10. Type an opening bracket **(**.
11. Type the **argument(s)** for the function.
12. Type a closing bracket **)**.
13. Press **Enter**.

USER-DEFINED FUNCTIONS

Discussion

It was described above how built-in worksheet or VBA functions can be used in sub procedures. There may be times, however, when none these functions meet your needs or meet then just partly. This is when a user-defined function procedure may have to be created.

Unlike a sub procedure, a function procedure ***only returns a value***. It does NOT move around a workbook selecting worksheets and cells, changing their properties and applying methods to objects. Like the dozens of built-in Excel and VBA functions (eg. SUM, COUNT, LOOKUP, IF, PMT, DATE, etc...), they merely carry out a calculation and return the result, either into the cell that they have been typed into or to the sub procedure that they are being used in.

A function procedure is enclosed by, **Function** and **End Function** statements.

The function procedure must have a name that is (usually) followed by arguments inside brackets. The arguments are the variables (pieces of information) that the function needs in order to calculate out and return a value.

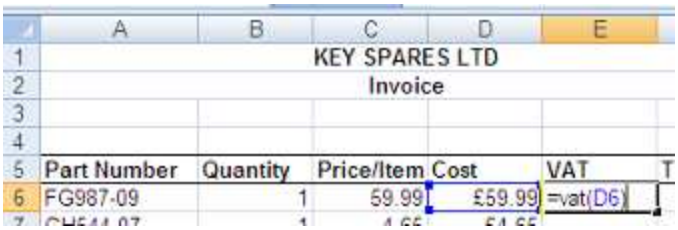
The following example shows a function procedure that is designed to calculate out Value Added Tax (at the time of writing this was 15%).

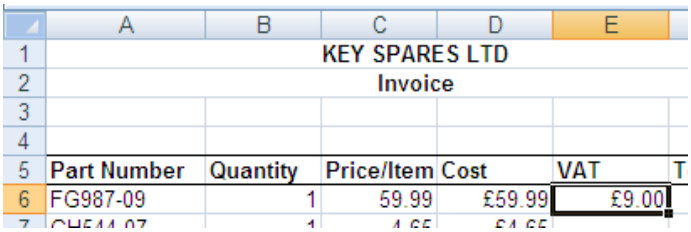
```
Function VAT(amount)
   VAT = amount * 0.15
End Function
```

The function procedure name is **VAT** and it requires a single piece of information (variable) to calculate out a value: **amount**.

The calculation that the function needs to perform to return a value for VAT is expressed in the statement: **VAT = amount * 0.15**.

This function could be used directly in a cell on a worksheet or it could be used in a sub procedure. The example below shows it being used on a worksheet. By typing in **=VAT** followed by an **opening bracket**, the "**amount**" that you want to carry out the VAT calculation on (in this case, cell D6) and a closing bracket, the result will be returned to the cell that the function is being typed into upon pressing **Enter**.





	A	B	C	D	E
1	KEY SPARES LTD				
2	Invoice				
3					
4					
5	Part Number	Quantity	Price/Item	Cost	VAT
6	FG987-09	1	59.99	£59.99	£9.00

The examples below shows it being used in the same way, but in a sub procedure:

Range("E6").Value = VAT(Range("D6"))

(This will return the result only of the function into cell E6, not the formula itself)

Range("E6").Formula = "=VAT(Range("D6"))

Or,

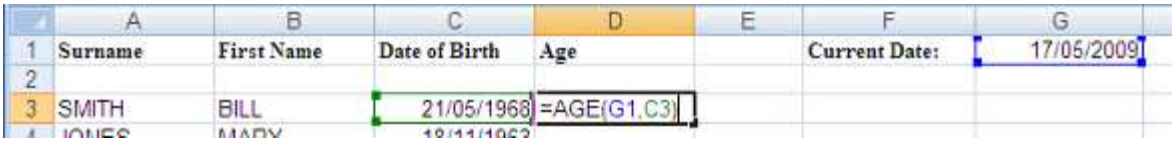
ActiveCell.Formula = "=VAT(ActiveCell.Offset(0 , -1).Value)

(These will write the actual formula into cell E6)

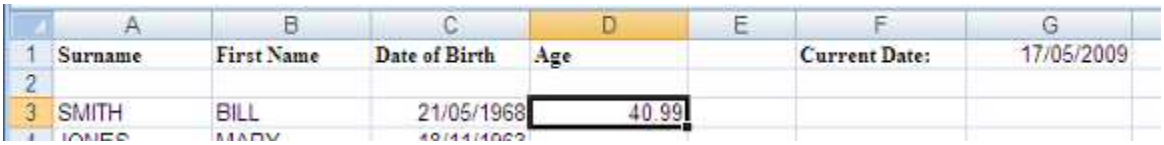
More examples are given below using multiple arguments and more code that is complex.

The following uses two arguments to calculate the difference in years and decimal fractions of a year, between two dates. The two arguments are separated by a comma:

```
Function AGE(current_date, date_of_birth)
        days = current_date  -  date_of_birth
        AGE = days / 365.25
End Function
```



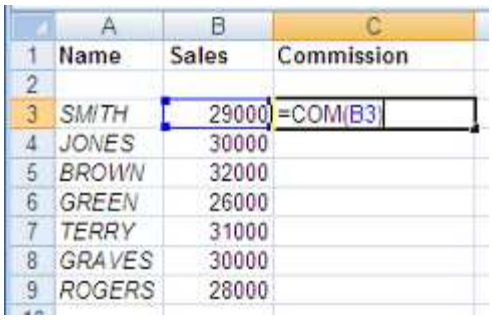
	A	B	C	D	E	F	G
1	Surname	First Name	Date of Birth	Age		Current Date:	17/05/2009
2							
3	SMITH	BILL	21/05/1968	=AGE(G1,C3)			



	A	B	C	D	E	F	G
1	Surname	First Name	Date of Birth	Age		Current Date:	17/05/2009
2							
3	SMITH	BILL	21/05/1968	40.99			

The following function return 10% of sales but only where the sales value is greater than or equal to 30000.

```
Function COM(sales_value)
  If sales_value >= 30000 Then
                COM = sales_value * 0.1
  Else
        COM = 0
  End If
End Function
```



	A	B	C
1	Name	Sales	Commission
2			
3	SMITH	29000	=COM(B3)
4	JONES	30000	
5	BROWN	32000	
6	GREEN	26000	
7	TERRY	31000	
8	GRAVES	30000	
9	ROGERS	28000	

	A	B	C
1	Name	Sales	Commission
2			
3	SMITH	29000	0
4	JONES	30000	3000
5	BROWN	32000	3200
6	GREEN	26000	0
7	TERRY	31000	3100
8	GRAVES	30000	3000
9	ROGERS	28000	0

Procedures

1. Launch or switch to the **VB Editor**.
2. Identify in the **Project Explorer** pane, the workbook (VBA project) that you want to add code to.
3. Open the module sheet or, insert a new module.
4. Position the cursor in the module where you want to create the function procedure.
5. Type **Function**.
6. Type a **space**.
7. Type a **name** for the function.
8. Type an opening bracket (.
9. Type the **variable name(s)** for the function argument(s).
10. Type a closing bracket).
11. Press **Enter**.
12. Type the necessary calculation(s) to return a value for the function.
13. Type **End Function**
14. Press **Enter**.

GETTING HELP WITH USER DEFINED FUNCTIONS

Discussion

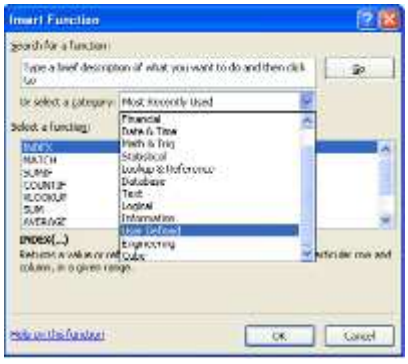
Excel offers help in two ways when entering your user-defined function into a worksheet.

The first is a Help Tip that appears as you type the function name in a cell. It appears as a list showing functions that match your spelling. This is only available in Excel 2007. There are NO Help Tips for user-defined functions in previous versions of Excel.

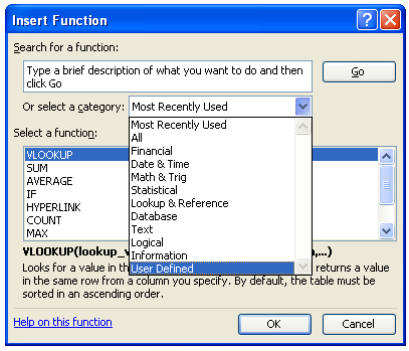
Help List in Excel 2007

The second way that you can get help with a user-defined function is by opening the **Insert Function** window. This is available in all versions of Excel and offers help on all Excel worksheet functions as well as any user defined ones available.

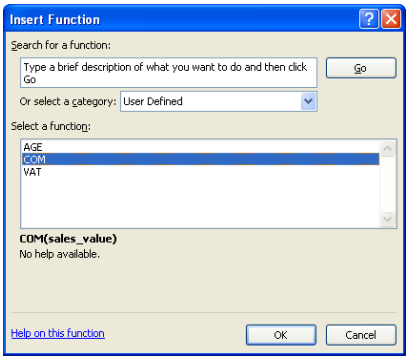
To find the user defined functions, you have to filter the **Select a function:** list by selecting **User Defined** from the **Or select a category:** dropdown list.

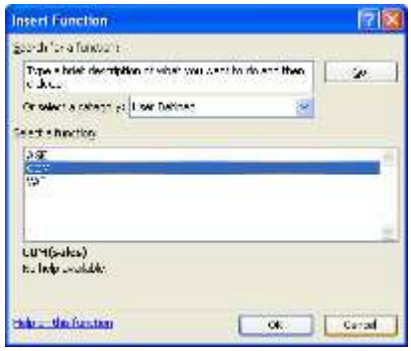


The Insert Function window (Excel 2007)



The Insert Function window (Excel 2000 -03)





Help pages are not available for user-defined functions. These have to be created using Microsoft tools that are not part of VBA or Office.

Procedures

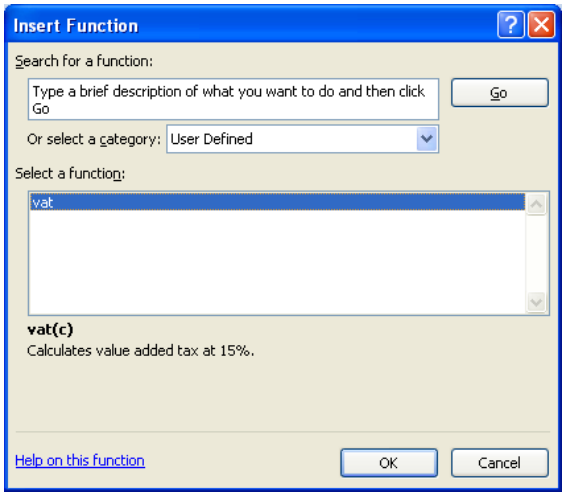
1. Select the cell where you want to use the user-defined function.
2. Press the  button.
3. Click the **Or select a category:** list.
4. Select **User Defined**.
5. Select the required function.
6. Click **OK**.
7. Enter the required arguments.
8. Click **OK**.

It is possible, however, to add a description for a user defined function as follows.

Procedures

	Excel 2000 - 2003	Excel 2007
1.	Click the **Tools** menu.	Click the **View** tab on the **Ribbon.**
2.	Select **Macro ▶ Macros...**	Click the **Macros** button.
3.		Select **View Macros...**.
4.	Type the name of the user-defined function in the Macro name: text box.	Type the name of the user-defined function in the Macro name: text box.
5.	Click the **Options** button.	Click the **Options** button
6.	Type a description for the macro.	Type a description for the macro.
7.	Click **OK**.	Click **OK**.
8.	Click **Cancel**.	Click **Cancel**.

The description for the macro will appear when the macro is selected in the **Insert Function** window.



Insert Function window showing description for a user-defined function

DECLARING VARIABLES IN USER DEFINED FUNCTIONS

Discussion

As in a sub procedure, it is good practice to declare any variables used in a function procedure. This includes:

- the function name
- the arguments, and
- any additional variables created in the calculation.

The function name variable is declared at the end of the Function statement immediately after the closing brackets of the arguments. The argument variables are declared within the brackets of the argument following each variable. Any additional variable created are declared at the beginning of the function procedure immediately below the Function statement. The following example declares the variables as used in the AGE function above. Underlining is used for clarity only:

```
Function AGE(current_date As Date , date_of_birth As Date) As Single
Dim days As Single
        days = current_date - date_of_birth
        AGE = days / 365.25
End Function
```

USING RANGES IN USER DEFINED FUNCTIONS

Discussion

There are many Excel worksheet functions that take a range (or ranges) as arguments, eg. SUM, AVERAGE, COUNT, MAX, MIN, to name but a few.

Ranges can also be used as variables in user defined procedures and are normally handled in the function procedure with a For... Each... Next loop (or loops).

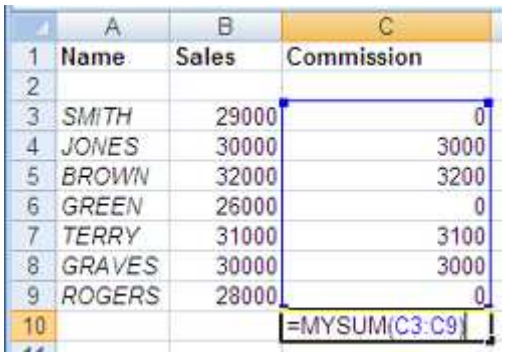
The following example does the same as a simple SUM function:

```
Function MYSUM(cells_to_sum as Range) As Single
  For Each indCel in cells_to_sum
        MYSUM = MYSUM + indCel.Value
  Next indCel
End Function
```

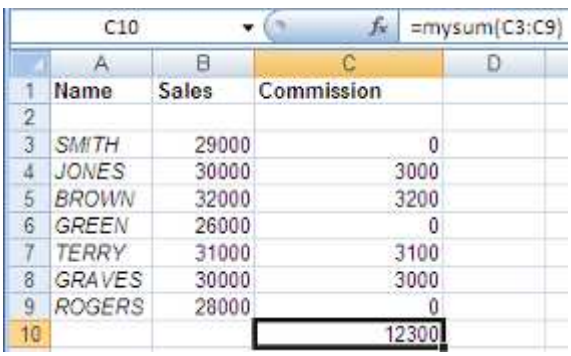
The loop repeats for as many cells as there are in the given argument (**cells_to_sum**). At each iteration of the loop, the value of **MYSUM** becomes its previous value plus the value of the next cell in cells_to_sum.

In the example shown in the pictures below, the loop iterates through the range C3 to C9.

The value of MYSUM first time round the loop will be 0 (the value in the first cell). The second time round the loop the value of MYSUM will be 3000 (its old value – 0, plus 3000 – the value in the second cell of cells_to_sum); the third time round the loop the value of MYSUM becomes 6200 (its old value – 3000, plus 3200 – the value in the third cell of cells_to_sum). Hence, by the time it has completed the loop, the value of MYSUM has risen to 12300. In essence, the loop carries out a cumulative addition of cells_to_sum.



	A	B	C
1	Name	Sales	Commission
2			
3	SMITH	29000	0
4	JONES	30000	3000
5	BROWN	32000	3200
6	GREEN	26000	0
7	TERRY	31000	3100
8	GRAVES	30000	3000
9	ROGERS	28000	0
10			=MYSUM(C3:C9)



C10 =mysum(C3:C9)

	A	B	C	D
1	Name	Sales	Commission	
2				
3	SMITH	29000	0	
4	JONES	30000	3000	
5	BROWN	32000	3200	
6	GREEN	26000	0	
7	TERRY	31000	3100	
8	GRAVES	30000	3000	
9	ROGERS	28000	0	
10			12300	

Procedures

1. Launch or switch to the **VB Editor**.

2. Identify in the **Project Explorer** pane, the workbook (VBA project) that you want to add code to.
3. Open the module sheet or, insert a new module.
4. Position the cursor in the module where you want to create the function procedure.
5. Type **Function**.
6. Type a **space**.
7. Type a **name** for the function.
8. Type an opening bracket **(**.
9. Type the **variable name(s)** for the function argument(s).
10. Type a closing bracket **)**.
11. Press **Enter**.
12. Type the necessary calculation(s) to return a value for the function.
13. Type **End Function**
14. Press **Enter**.

EXERCISES

USER DEFINED FUNCTIONS

Task 1: Create User Defined Functions.

1. Open the file **Practice Functions**.
2. The Temperature sheet gives formulas for converting Fahrenheit into Centigrade and vice versa. Create two user-defined functions that will do the same.

   Suggest starting with the lines:

   Function DegF(TempC) to convert Fahrenheit to Centigrade and,

   Function DegC(TempF) to convert Centigrade to Fahrenheit .
3. Create a function for calculating the hypotenuse of a right-angle triangle. Use the existing formula in the Pythagoras sheet as a guide but beware that in Visual Basic, the function for square root is SQR not SQRT as in Excel.
4. Save and close the file.
5. The Gas Bill sheet gives a complex series of calculations to arrive at a total charge in column G. Create a user-defined function that will calculate the total for each quarter. Test it in column H. The results should match those in column G. You will probably need to use some extra variables in your code and a suggestion for the arguments is:

   Function GasBill (UnitsUsed, LowUnits, HighRate, LowRate)
6. Save and close the file.

Task 2: Create Complex User Defined Functions.

1. Open the file **Sales Tracking**.
2. Create in this workbook, a function procedure named **HighSales**, which will return the number of sales in a range that are over 3000 in value. The first line of the function procedure should be:

   HighSales(RangeToCount)
3. Use the function in cell **F22** of the spreadsheet, using the range **F2:F20** as the argument. The result should be 7.
4. Create another function in this workbook named **CountText**, which will return the number of cells in a range containing a text value. The first line of the function procedure should be:

   CountText(RangeToCount)

5. Use the function on cell **F23** of the spreadsheet, using the range **A2:E20** as the argument. The result should be 57.
6. Save and close the file.

LESSON 4 - ADD-IN APPLICATIONS

In this lesson, you will learn how to:

Create an Add-In

Viewing an Add-In's code

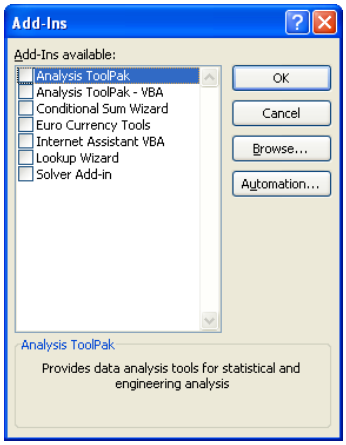
DEFINING AN ADD-IN

Discussion

An Excel Add-in is a compiled version of a workbook containing procedures, toolbars and menu configurations, user-defined functions and forms and many other VBA features. Add-ins usually provide additional functionality that makes the standard application more powerful. In its simplest form the Add-in may just contain user-defined functions.

An Add-in is saved like other files to a suitable location on your system (it has a **.xla** extension) and "added in" when/ if necessary to the Excel installation on your computer. Like the Personal Macro Workbook, the Add-in file is normally kept hidden in Excel, but can be viewed in the VB Editor.

Add-ins are a convenient way of distributing sub procedures, functions and other VBA functionality to groups of users who can load them into their own Excel application via a simple dialog box.



The Add-in dialog box (Excel all versions)

Excel comes with several built-in Add-ins; the number available varies with the version. These can be accessed and loaded by opening the Add-ins dialog box.

When a workbook is saved as an Add-in, Excel converts its code into a compressed form that cannot be read, altered or converted back into a workbook. This is to make the code in the Add-in work more quickly than in the original workbook.

The workbook's Title and Comments, set in File Properties, are used as the name and description to help identify Add-ins.

Because a compiled Add-in cannot be converted back to a workbook, it is essential that before saving as an Add-in, the workbook is saved separately as a workbook. If changes are needed, therefore, they can be written into the original workbook and then resaved as another (or the same) Add-in.

When a workbook is made into an add-in, worksheets in the workbook are hidden, and subroutines in the Add-in's project are hidden from the user (the routines do not appear in the macros dialog box).

If the Add-in is saved to the XLSTART folder, its functionality applies as if it was built into Excel.

CREATING AN ADD-IN FOR USER DEFINED FUNCTIONS

Discussion

Creating user-defined procedures as described in the previous lesson only makes them available in the workbook that contains their code. To make them available to all workbooks, the workbook containing the function procedures' code has to be saved as an Add-in, and then added in to the Excel installation on the user's computer.

To prevent the Add-in code being altered by users, it is advisable to password protect the workbook before saving it as an Add-in. It is also advisable to give the workbook file a title and description in order to make it more easily identified for use later.

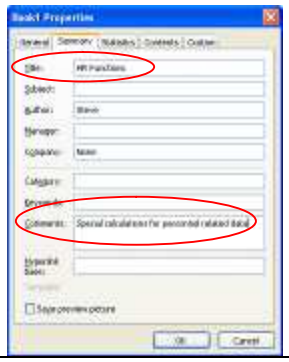
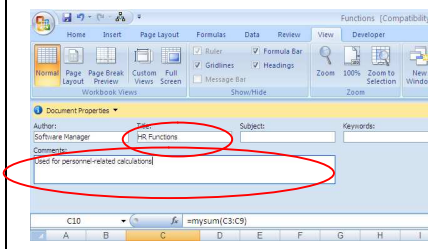
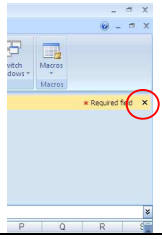
Procedures – Step 1: Protecting the Code

1. Launch or switch to the **VB Editor**.
2. Select the workbook (VBA project) that contains the function procedure code in the **Project Explorer** pane.
3. Click the **Tools** menu.
4. Select **VBA Project Properties...**.
5. Open the **Protection** tab.
6. Click the **Lock project for viewing** check box.
7. Type a password in the **Password** text box.
8. Repeat typing the password in the **Confirm password** text box.

9. Click **OK**.

Procedures – Step 2: Adding File Properties

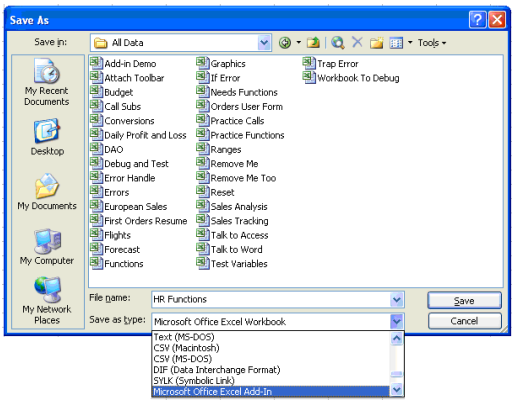
1. Switch to the **Excel** window.

2. Ensure you have active, the workbook containing the function procedure code.	
Excel 2000 - 03	**Excel 2007**
3. Click the **File** menu.	Click the **Microsoft Office Button**.
4. Select **Properties**.	Point to **Prepare** ▶ and select **Properties** in the side menu.
5. Type a descriptive name for the file in the **Title:** text box. 	Type a descriptive name for the file in the **Title:** text box. 
6. Type an explanation of or purpose for the file in the **Comments:** text box.	Type an explanation of or a purpose for the Add-in in the **Comments:** text box.
7. Click **OK**.	Click the **Close** button in the top right of the properties pane. 

Procedures – Step 3: Saving As an Add-in

Excel 2000 - 03	**Excel 2007**
1. Click the **File** menu.	Click the **Microsoft Office Button**.
2. Select **Save As**.	Point to **Save As** ▶ and select **Other Formats** in the side menu.
3. Type a name for the Add-in in the **Name:** text box.	

4. Select Microsoft Office Excel Add-In from the **Save as type:** dropdown list. (Wording may vary slightly between versions)



The Save As dialog box (Excel 2003 shown). Appearance may vary slightly between versions but basic functionality is the same.

5. If the Add-in is for your personal use only, leave the location as selected by Excel (the default *Add-Ins* folder is part of your personal profile).

   If the Add-in is to be shared with other users, select a suitable shared location in the **Save in:** dropdown list.

6. Click **Save**.

INSTALLING AN ADD-IN

Discussion

Once the Add-in has been created, prepared and saved as described in the topics above, you, and other users, will need to install it on your (their) computer to make the code available for use.

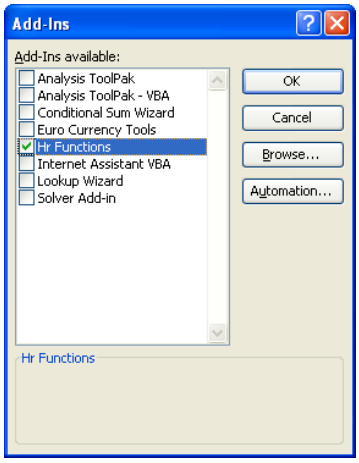
This is achieved by opening the Add-ins dialog box and depending on where it has been saved, either selecting it from the list (if it has been saved in the default location) or browsing to the location that it has been saved in (if you have selected your own location).

If the Add-in file has been saved in the default location, it will appear in the list with the name that you gave it as a **Title** in *File Properties*. If the Add-in file has been saved in a different location, it will appear in the list after your browse for it.

After the Add-in has been selected and the dialog box confirmed, the procedure(s) in it will be immediately available for use.

Procedures

Excel 2000 - 03	Excel 2007
1. Click the **Tools** menu.	Click the **Microsoft Office Button**.
2. Select **Add-ins**.	Click the **Excel Options** button.
3. Click the checkbox beside the Add-in that you want to install, or... ...click **Browse...** and open it from the location where it has been saved.	Select **Add-ins** in the pane at the left.
	At the bottom of the pane at the right, click the **Go...** button.
	Click the checkbox beside the Add-in that you want to install, or... ...click **Browse...** and open it from the location where it has been saved.
4. Click **OK**.	Click **OK**.



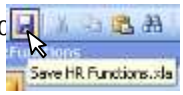
EDITING AN ADD-IN

Discussion

If you encounter errors and need to debug the code in your add-in, you can make the add-in visible and view the code provided you supply the proper project password.

Once the project is unprotected, you can view, debug the code and resave the Add-in.

Procedures

1. Launch or switch to the **VB Editor**.
2. In the **Project Explorer** pane, double click the Add-in (.xla file) in that you want to edit.
3. If password protected, you will be prompted for the password.
4. Open the module containing the function procedure(s) that you want to edit.
5. Make necessary changes to the code.
6. Click the **Save** button in the VB Editc 

REMOVING AN ADD-IN

Discussion

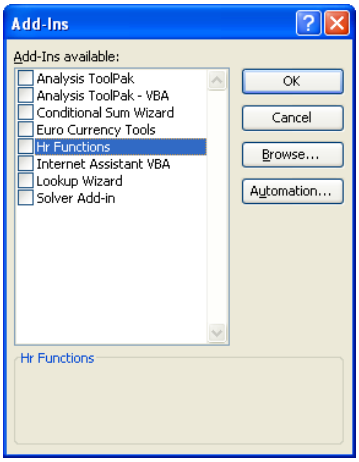
If you have many Add-in installed into Excel it can slow down its performance, particularly when starting up and it has to load them all.

It is a good idea, therefore, to uninstall any Add-ins that are no longer required for use. This does not delete the Add-in file (.xla); the file will still be available for re-installing another time.

Procedures

Excel 2000 - 03	Excel 2007
1. Click the **Tools** menu.	Click the **Microsoft Office Button**.
2. Select **Add-ins**.	Click the **Excel Options** button.
3. Click the checkbox beside the Add-in that you want to uninstall.	Select **Add-ins** in the pane at the left.
	At the bottom of the pane at the right, click the **Go...** button,
	Click the checkbox beside the Add-in that you want to uninstall.

4. Click **OK**.	Click **OK**.



EXERCISE

WORKING WITH AN ADD-IN

Task: To create and install to Excel, additional VBA functionality.

1. Open the file named, **Conversions**. This file contains several function procedures.
2. Carry out the necessary actions to save this file as an "add-in", named **Conversion Functions**.
3. Close the file.
4. Open the file named, **Needs Functions**.
5. Add into **Excel** the **Conversion Functions** saved above.
6. Test a couple of the functions on **Sheet1**.
7. Save and close **Needs Functions**.
8. Remove the **Conversion Functions** add-in from Excel.

LESSON 5 – TESTING AND DEBUGGING CODE

In this lesson, you will learn how to:

- Step through code to identify and/ or rectify potential problems
- Use the Locals and Watch Windows to observe variable values
- Set a Breakpoint to halt execution of a procedure
- Work with the Immediate Window to test code and variables

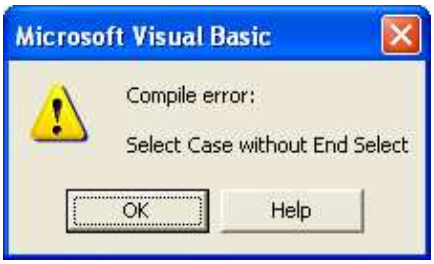
TYPES OF ERROR AND DEBUGGING

Discussion

There are various types of error that may occur whilst executing code (run-time), or during the writing of the code (compile-time).

Errors fall into four broad types:

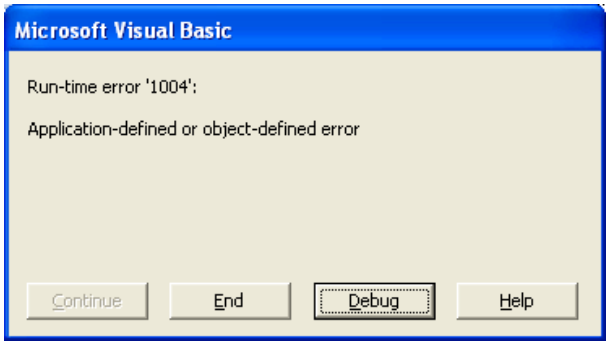
- **Language (syntax) errors** – occur as a result of misspells, leaving out essential punctuation, or omitting keywords at the end of a statement (eg. **If**... without **End If**).
- **Compile errors** – occur when the VB Editor cannot convert your statement into viable code such as when you try to use a method or property on an object that does not support it.
- **Run-time** errors – occur when the procedure is run and it encounters an unexpected problem that did not show up during compilation or that the developer did not anticipate.
- **Logical** errors – occur when the procedure produces unexpected results are can be due to many reasons including a miscalculation of variables, incorrect use of a method parameter or property value, or selecting an incorrect range, worksheet or workbook.



Compile Error Warning

The good news is that the VB Editor can identify many language and compile errors as soon as you press Enter at the end of a statement (or use the up/ down arrow, or click onto another statement). These errors, therefore, can normally be corrected immediately. Some language and compile errors, however, can only be identified during run-time.

In most cases, the cause of run-time errors can also be identified because the procedure comes to a halt with a Run-time error message. If the Debug button is clicked, it highlights in yellow the statement that failed (a condition known as break mode).



Run-time Error Warning

Logical errors can be the most difficult to pin down and discovering the cause usually involves going through the procedure statement by statement (a technique known as ***stepping***) until it becomes apparent where it goes wrong.

Stepping can also be used in conjunction with the **Watch** or the **Locals** window. These keep an eye on your variable as you step the code and can indicate where and when a variable may not be behaving as expected!

The VB Editor provides various tools to help locate the source and cause of any run-time errors detected. This lesson will examine four of them:

1. Stepping
2. Watching variables
3. Creating Breakpoints
4. Using the Immediate Window

STEPPING THROUGH A PROCEDURE

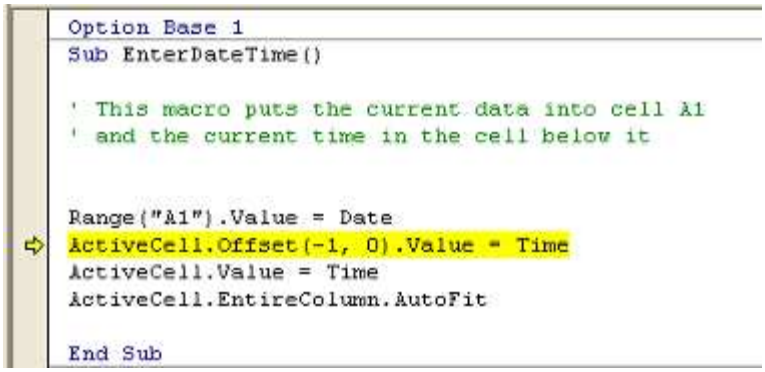
Discussion

Step Into

Normally, your code runs unattended; it executes until its logical end. When you are testing code, however, it is often useful to step through it line by line, watching each line of code take effect. This makes it easy to determine exactly which line is not producing the desired effect.

You can step through code line by line by selecting Step Into from the Debug menu of the VB Editor or, by pressing the F8 key to start the procedure in which the cursor is and then pressing F8 to "step" each statement.

Stepping causes VBA to execute each line one at a time, highlighting the next line of code in yellow. Note, the highlighted line is the line of code that will execute when you next press F8; it does not mean it has already been executed.



```
Option Base 1
Sub EnterDateTime()

' This macro puts the current data into cell A1
' and the current time in the cell below it


Range("A1").Value = Date
ActiveCell.Offset(-1, 0).Value = Time
ActiveCell.Value = Time
ActiveCell.EntireColumn.AutoFit

End Sub
```

A Procedure being Stepped Into

By showing the **Debug** toolbar in the VB Editor (View, Toolbars), access to the various debugging and testing tools can be made quicker and more obvious.

The Debug Toolbar (all versions)

Button	Function
Toggle Breakpoint	Placing Breakpoints within your code will stop execution at that point.
Quick Watch	After selecting an expression, you would use the Quick Watch button and Add button. The watch window will show the information about the expression. (Use Step Into).
Step Into	Using the Step Into button allows you to execute one line of code at a time
Step Over	Step Over executes called sub Procedures without stepping through them.
Step Out	Step Out completes execution of the current procedure and pauses on the next line.

Step Over

If you are stepping a procedure that calls another procedure, the VB Editor will step into the called procedure and execute it line by line.

You can, therefore, click **Debug, Step Over** (or press **SHIFT F8**) to cause the entire called procedure to be executed immediately. This will make debugging simpler if you are confident that the problem does not lie within that called procedure.

Step Over takes you back to the statement following where the called procedure was called.

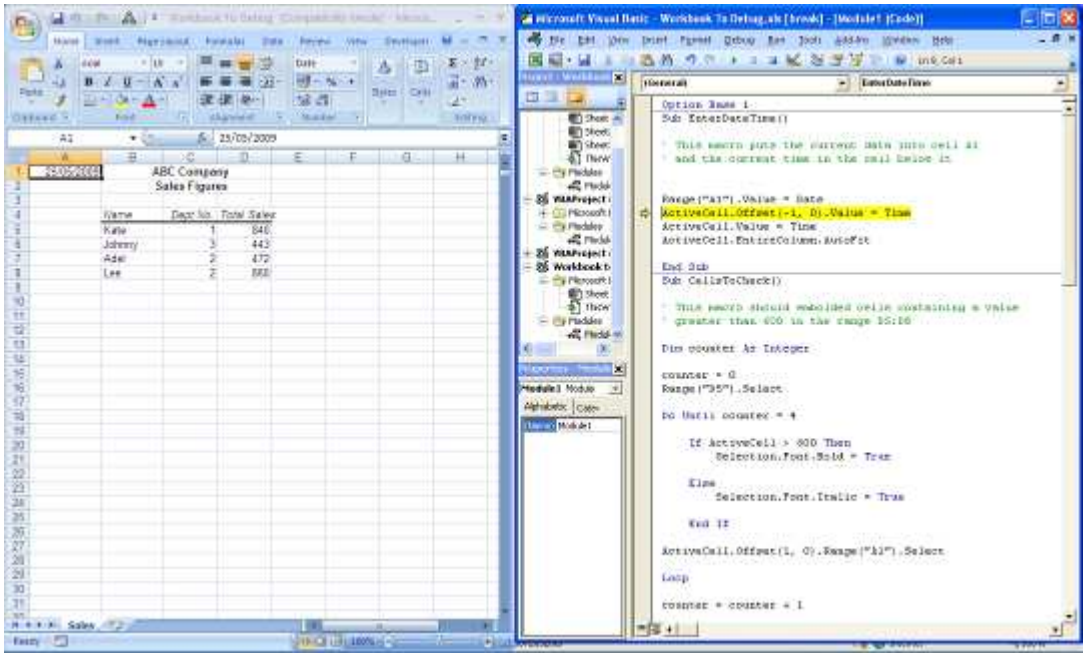
Step Out

In you are stepping through a called procedure statement-by-statement, you can click **Debug**, **Step Out** (or press **CTRL+SHIFT+F8**) to come out of the that procedure. This causes VBA to execute until the end of the procedure (an `End Sub` or `Exit Sub` statement) and then stop at the line of code immediately following the line that called the procedure.

A useful way of stepping is to tile the VB Editor window and the Excel window. As you step through the procedure, you will be able to see exactly what the effect of each statement is in Excel. This can help identify errors or potential problems more easily.

Procedures

1. Launch or switch to the **VB Editor**.
2. Identify in the **Project Explorer** pane, the workbook (VBA project) containing the procedure that you want to step through.
3. Open the module sheet containing the procedure.
4. Click the **Debug** menu.
5. Select **Step Into**.
6. Continue selecting **Step Into** (or press **F8**) until you get to the end of the procedure.



Stepping a Procedure - the VB Editor and Excel Tiled Side-by-Side

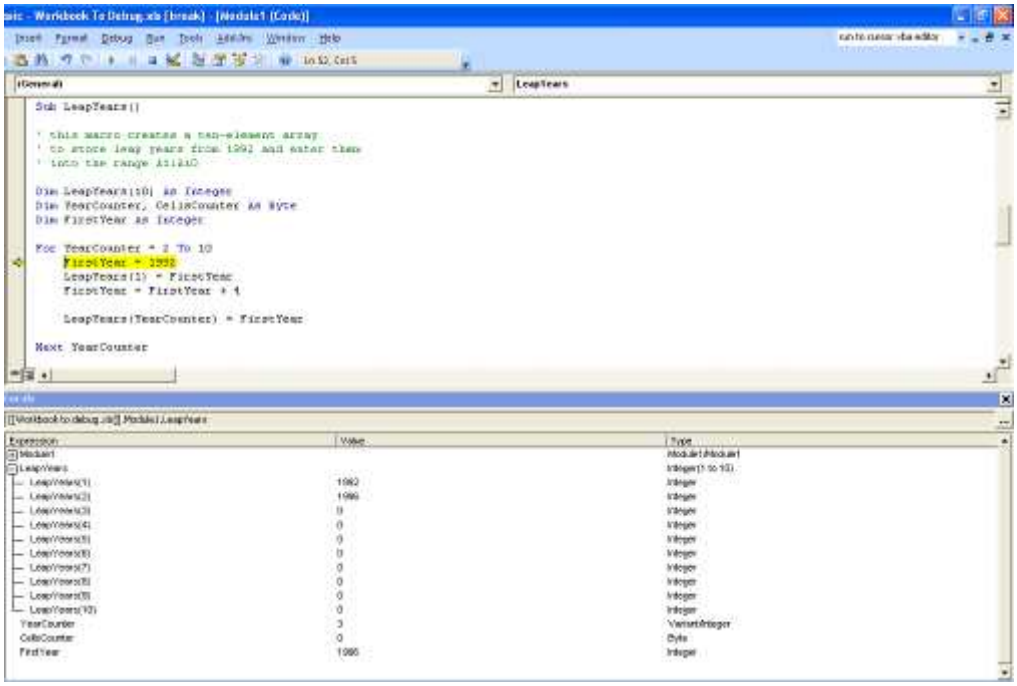
DISPLAYING VARIABLE VALUES

Discussion

Logical errors can, and often do, occur as a result of variables that are not calculating or picking up their values as expected. The **Locals** and **Watch** windows can be used while stepping to display the behaviour of variables and hence, assist in identifying errors and potential problems.

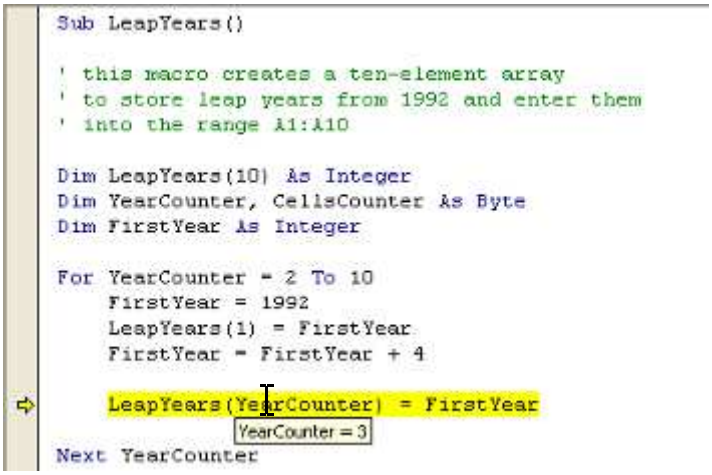
Locals Window

The Locals Window displays all the variables in a procedure (as well as global variables declared at the project or module level), and their values. This makes it easy to see exactly what the value of each variable is at a particular point in the procedure and how it changes as you step through the code. You can display the Locals Window by choosing it from the **View** menu. The Locals Window does not allow you to change the values of variables, it simply displays their names and values.



Procedure being Stepped with the Local Window Displaying all Variables in the Procedure and their Current Values

> The value of a variable at a particular point can also be seen as you step through a procedure by holding the mouse pointer over the variable name.



Displaying the Value of a Variable While Stepping

Procedure

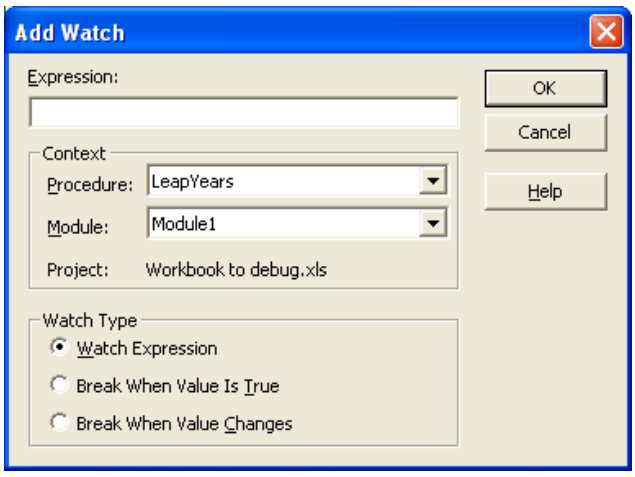
1. Launch or switch to the **VB Editor**.
2. Identify in the **Project Explorer** pane, the workbook (VBA project) containing the procedure that you want to add a watch to.

3. Open the module sheet containing the procedure.
4. Position the cursor on the procedure containing the variables that you want to watch.
5. Click the **View** menu.
6. Select **Locals Window**.
7. Click the **Debug** menu.
8. Select **Step Into**. Observe the variable values in the Locals window.
9. Continue selecting **Step Into** (or press **F8**) and observing the variable values until you get to the end of the procedure.

Watch Window

The Watch Window allows you to be more selective about which variable(s) you "watch." You can set a variable with three watch types:

Watch Type	Description
Watch Expression	Watches the variable statement-by-statement as you step through the procedure (same as Locals Window).
Break When Value Is True	Halts the procedure and enters break mode when the variable is given a value (value becomes TRUE).
Break When Value Changes	Halts the procedure and enters break mode each time that the value of the variable changes.



The Add Watch Window

Procedure

1. Launch or switch to the **VB Editor**.

2. Identify in the **Project Explorer** pane, the workbook (VBA project) containing the procedure that you want to add a watch to.
3. Open the module sheet containing the procedure.
4. Click the **View** menu.
5. Select **Watch Window**.
6. Right-click the variable name in the procedure that you want to watch.
7. Select **Add Watch...**.
8. Repeat 6 above for all other variables that you want to watch
9. Click **OK**.
10. Click the **Debug** menu.
11. Select **Step Into**. Observe the variable values in the Watch window.
12. Continue selecting **Step Into** (or press **F8**) and observing the variable values until you get to the end of the procedure.
13. Click the close button in the top right of the Watch window.

BREAK MODE

Discussion

A break point is a setting on a line of code that tells VBA to pause execution immediately before that line of code is executed and code execution is placed in what is called *break mode*. When the VB Editor is in break mode, you can edit the procedure as normal.

To put a break point on a line of code, place the cursor on that line select **Debug**, **Toggle Breakpoint,** press **F9**, or click in the grey bar at the left of the module next to the line where you want to add the break point.

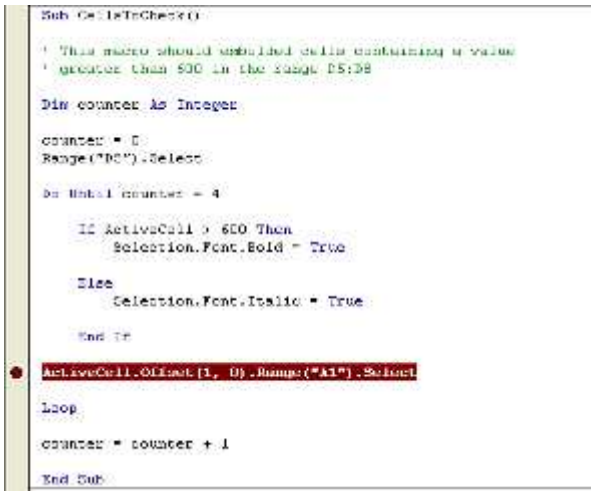
Removing a break point is the same as applying one (hence, "toggle").

When a line contains a break point, it is displayed with a brown coloured background and a large dot in the grey band at the left of the module sheet.

Immediately before this line of code is executed, it will appear with a yellow background. When a break point is encountered, code execution is paused but that line of code has not yet executed. You cannot place break points on blank lines, comment lines, or variable declaration lines (lines with `Dim` statements).

After a break point is encountered, you can resume normal code execution by pressing **F5**, selecting **Run**, **Continue**, clicking the **Continue** button, or stepping through the code line by line (**F8**).

> Break points are not saved in the workbook file. If you close the file, all break points are removed. Breakpoints are preserved only while the file is open.



Procedure with a Breakpoint Applied

Procedures

1. Launch or switch to the **VB Editor**.
2. Identify in the **Project Explorer** pane, the workbook (VBA project) that you want to add code to.
3. Open the module sheet containing the code that you want to add a breakpoint to.
4. Position the cursor in the procedure where you want to add the breakpoint.
5. Click the **Debug** menu.
6. Select **Toggle Breakpoint**.
7. Click the **Run** button  . The procedure will execute down to the breakpoint and then enter break mode.
8. Step through the rest of the procedure.
9. Position the cursor on the statement containing the breakpoint.
10. Click the **Debug** menu.

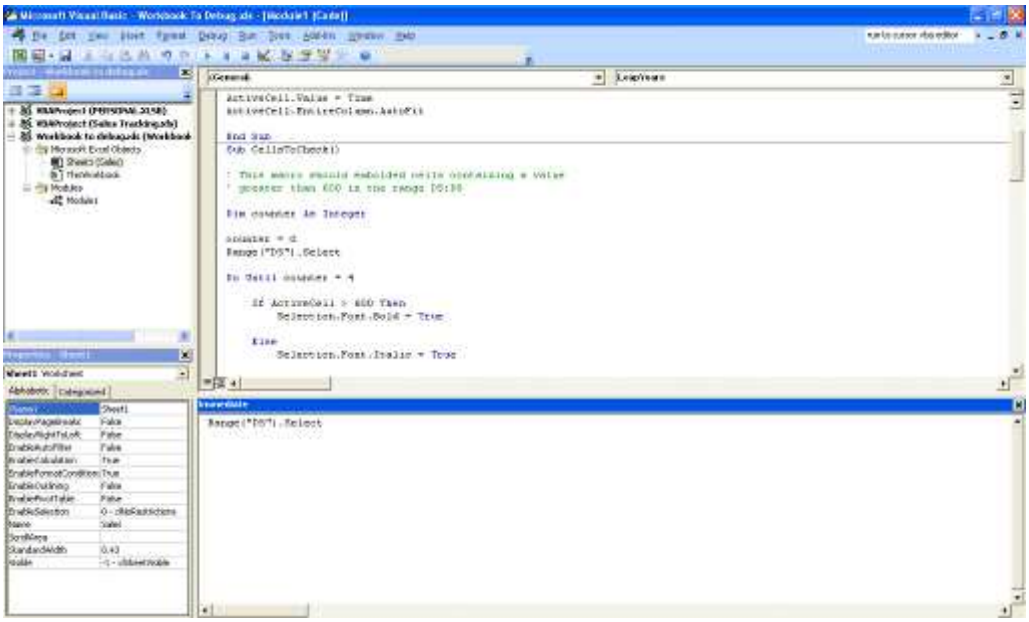
11. Select **Toggle Breakpoint**.

> Break points can also be toggled by clicking on the grey band immediately to the right of the module sheet.

Immediate Window

The Immediate Window is a window in the VB Editor where you can try out statements and/ or variables while your code is in Break mode or when no macro code is executing. It is most useful when testing new or problematic code.

To display the Immediate Window, press **CTRL+G** or select **View**, **Immediate Window**.



VB Editor with the Immediate Window Displayed

In the **Immediate** Window, you can test a statement by typing it in and pressing Enter. For example:

Range("A1").CurrentRegion.Select

or,

Selection.Interior.Color = vbMagenta

> A useful way of using the Immediate Window is to tile the VB Editor window and the Excel window. As you press Enter on each statement, you will be able to see exactly what the effect is in Excel.

To test a variable in the Immediate Window, create the variable name and give it a value. Press Enter. Type on the next line: ***?variablename*** and press **Enter**. The value of the variable will be given on the line below. For example:

numCols = Selection.Columns.Count

?numCols

12

Or,

?ActiveCell.Address
A10

Because of the way that the Immediate Window is designed to work, it will not allow you to test statement blocks (eg, If..., For... , Select Case... etc.). You can, however, combine several "logical" lines of code in to a single "physical" line of code using the colon (**:**) and execute this as an entire command. For example:

If ActiveCell.Value < 1000 Then: ActiveCell.Value = 0: Else: ActiveCell.Value = 1

End IF is **NOT** used when writing the **If** statement block on one single line.

You can extend a single statement across several lines in the Immediate Window by using the line-continuation character (a space followed by an underscore (_)).

The Immediate Window always behaves as if there is no **Option Explicit** statement in the active code module (see page 20), namely, you don't have to declare variables that you will be using in the Immediate Window commands. In fact, this is prohibited and you will receive an error message if you attempt to use **Dim** in the Immediate Window.

There is no programmable way of clearing the Immediate window. You have to select the text and press **Delete** on the keyboard.

Procedures

1. Launch or switch to the **VB Editor**.
2. Click the **View** menu,

3. Select **Immediate Window**.
4. Position the cursor in the immediate window.
5. Write and test code as necessary.

Debug.Print

You can use the **Debug.Print** statement anywhere in your code to display messages or variable values in the Immediate Window. These statements do not require any confirmation or acknowledgement so will not affect the execution of your code, so it is safe to leave in the code when it is distributed to your users.

They are used purely to explain, clarify, remind or note what is taking place at a particular point of the procedure.

They are much like comments except that they are written for you to see in the Immediate Window. If the Immediate Window is not displayed, **Debug.Print** has no effect.

For example, you can send a message to the Immediate Window when a particular section of code is executed.

Some code...

Debug.Print "Starting Code Section 1"

More code...

Debug.Print "Procedure is now calling Procedure2"

See also Appendix III, Debug.Assert on page 123.

EXERCISE

TESTING AND DEBUGGING CODE

Task: To test and identify problems and errors in a sub procedure

1. Open the file, **Workbook to Debug**.
2. Tile vertically the Excel and VB Editor windows.
3. Step the macro **EnterDateTime** and identify any problems.
4. Correct the error(s) and step the macro again to ensure it works correctly. It should enter the date into cell A1 and the time into the cell below it.
5. Run the sub procedure **CellsToCheck**.
6. Step the macro to identify the problem (you may also find it useful to "Watch" the counter variable as you step).
7. Run the **LeapYears** sub procedure and note the incorrect result.
8. Step the sub procedure together with the Locals window and identify the reason for the incorrect result. Correct the code so that it runs correctly.
9. Show the Debug toolbar.
10. Step through the procedure named, Main using the Step Into.
11. Repeat 10 above using the Step Over button. Note how it omits stepping the called procedures.
12. Save and close the file.

LESSON 6 – ERROR HANDLING & TRAPPING

In this lesson, you will learn how to:

Handle errors using an If statement

Trap errors using the On Error statement

Trap errors by means of their error codes

ERROR HANDLING USING IF

Discussion

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

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

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

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

Consider the examples below:

```
If Selection.Cells.Count >1 Then
        ...code
Else
        MsgBox "This macro requires more than one cell to be selected. Procedure will terminate"
        Exit Sub
End If
```

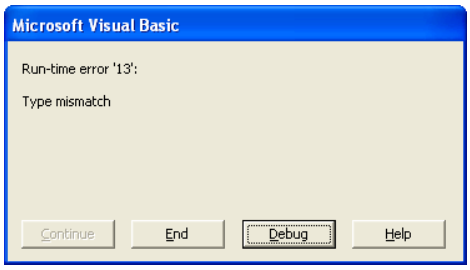
Or,

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

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

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

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



Type mismatch run-time error

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

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

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

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

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

Procedures

1. Launch or switch to the **VB Editor**.

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

ERROR TRAPPING

Discussion

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

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

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

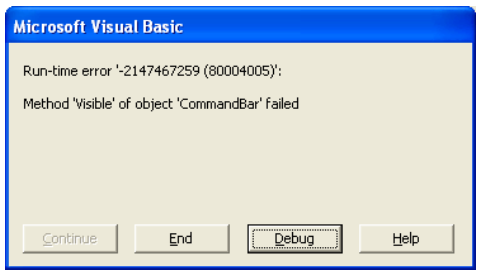
On Error Resume Next

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

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

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

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



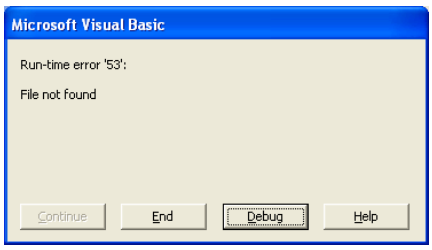
By writing the code as follows, the run-time error window will be ignored and the procedure will successfully continue and hide all the toolbars.

```
On Error Resume Next

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

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

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

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



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

```
On Error Resume Next

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

In the earlier example on page 84 where the user is prompted for a numeric input, On Error Resume Next would negate the need for the **If**... **Then**... **Else**.

```
On Error Resume Next

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

The result in this case, however, is hardly ideal because although the run-time error message does not appear, the procedure results in nothing happening! This is where a different On Error statement is needed.

On Error GoTo <*label*>

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

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

In the example on page 87, using On Error GoTo would be as follows:

```
On Error GoTo errhandle

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

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

End Sub
```

It is true that this code in its current form does not contribute any more than using the **If**...**Then**...**Else** explained on page 84. Its advantage, however, lies in its ability to return to where the error occurred and attempt the statement again. This involves the use of the word **Resume**.

Resume

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

```
On Error GoTo errhandle

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

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

End Sub
```

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

Creating an interactive message box is covered in the Excel Introduction to VBA Course. An extract of the course materials are given in Appendix IV on page 125. The example below assumes knowledge and experience of this topic.

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

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

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

```
Sub CalculateInterest

Dim interestRate as Single

On Error GoTo errhandle

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

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

End Sub
```

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

Procedures

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

TRAPPING ERRORS WITH ERR NUMBERS

Discussion

Most run-time errors generate error numbers (see Appendix II on page 119). When the On Error Goto *<Label>* is used to trap errors, the number of any error is returned by the **Err Function** that acts like a public variable. The value of **Err**, therefore, can be tested and acted upon using an **If**... **Then**... **Else** statement.

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

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

```
Sub CalculateInterest

Dim interestRate as Single
On Error GoTo errhandle

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

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

End Sub
```

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

```
errhandle:
If Err.Number = 13 Then
    response = MsgBox("Invalid Data Entered. Do you want to try again?", vbYesNo)
            If response = vbYes Then
                    Resume
            End If
ElseIf Err.Number = 9 Then
    MsgBox ("Sheet Interest Calc not found. Please check sheet names and re-run procedure")
Else
    MsgBox "Unexpected error. Procedure will terminate"
End If
```

Procedures

1. Launch or switch to the **VB Editor**.
2. Identify in the **Project Explorer** pane, the workbook (VBA project) that you want to add code to.
3. Open the module sheet containing the code or, insert a new module.
4. Position the cursor on the line below the **Sub** statement.
5. Type **On Error GoTo errhandle** (or other "label" to identify a point further down in the procedure where the procedure must divert to if a runtime error occurs.
6. A the bottom of the procedure and immediately above the **End Sub** statement, type errhandle (or other label used).
7. Type a **colon**.
8. Press **Enter**.
9. Type **If**.
10. Type a **space**.
11. Type **err.Number =** .
12. Type the anticipated error number.
13. Type a **space**.
14. Type **Then**.
15. Press **Enter**.
16. Type appropriate code to handle the error generated by the error number in the previous test.
17. Press **Enter**.
18. If it is anticipated that the procedure may generate additional error numbers, type **ElseIf**.
19. Type a **space**.
20. Type **err.Number =** .
21. Type the anticipated error number.
22. Type a **space**.
23. Type **Then**.
24. Press **Enter**.
25. Type appropriate code to handle the error generated by the error number in the previous test.

26. Press **Enter**.
27. Continue as described in 18 – 26 above for any further anticipated error codes.
28. Press **Enter**.
29. Type **Else:**.
30. Press **Enter**.
31. Type **MsgBox "An unexpected error has occurred"**. This is to cover any errors NOT anticipated by the If/ ElseIf(s) above.
32. Press **Enter**.
33. Type **End If**.
34. Position the cursor on the line immediately above the errhandle label.
35. Type **Exit Sub**.
36. Press **Enter**.

On Error GoTo 0

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

```
On Error Resume Next      ' trap errors from here onwards

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

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

Procedures

1. Launch or switch to the **VB Editor**.
2. Identify in the **Project Explorer** pane, the workbook (VBA project) that you want to add code to.
3. Open the module sheet containing the code or, insert a new module.

4. Position the cursor in the procedure where you want the error handling no longer to have an effect.
5. Type **On Error GoTo 0**.
6. Press **Enter**.

Err.Description

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

```
MsgBox "The error" & Err.Description & _
" has occurred. Please contact the Helpdesk and quote error number" _
        & Err.Number & ". Thank you."
```

It is true to say that this message box does not say much more than the normal, run-time error window, but it is rather less scary and upon clicking OK, does not potentially leave the procedure hanging in break mode.

EXERCISE

DEALING WITH POTENTIAL ERRORS IN PROCEDURES

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

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

Task 2: Trapping errors in a sub procedure.

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

c. return control to where the error occurred, if applicable.

8. Run and test the sub procedure again. Correct any errors.
9. Open the file **Remove Me Too**.
10. Leave this file open and switch back to the **Trap Error** workbook.
11. Run the **Delete Old File** macro and attempt to delete **Remove Me Too**. Is your previous code handling the error correctly?
12. De-activate the error handling code by commenting out the **On Error GoTo** statement at the top of your procedure.
13. Run the **Delete Old File** macro again.
14. Note the error message. How does it differ from the one displayed in 4 above? (*Tip*: note the error number)
15. Edit the **DeleteOldFile** sub procedure with some additional error-trapping code to deal with both potential errors in a more user-friendly way.
16. Re-activate the error handling code at the top of the procedure.
17. Test the macro by attempting again to delete **Remove Me Too**.
18. The file should be successfully deleted.
19. Save and close all open files.

BEWARE - files deleted by a macro cannot be recovered.

LESSON 7 - BUILT-IN DIALOG BOXES AND CUSTOM USERFORMS

In this lesson, you will learn how to:

Display built-in dialog boxes

Create and display user defined forms

Use controls on userforms

Add code to create event procedures for the useform

EXCEL DIALOG BOXES

Discussion

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

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

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

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

Procedures

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

USER-DEFINED FORMS

Discussion

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

Creating a functioning UserForm can be broken down into five steps:

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

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

INSERTING A USERFORM INTO A WORKBOOK

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

Procedures

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

ADDING CONTROLS TO A FORM

Controls are the objects that you can add to a user form so that the user can "talk" with it (hence, dialog box). These appear on the Toolbox toolbar when a UserForm is

inserted into a workbook in the VB Editor, and are quite literally drawn onto the form.

The UserForm Toolbox toolbar

These are as follows:

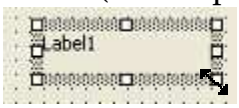
Name of Control	Button Image	Description
Select Objects		Selects objects on a form
Label		Static text that is displayed to inform the user
Text Box		Mostly used when a typed-in input is requested from the user
Combo Box		Displays a dropdown list of values to choose from
List Box		Displays a fixed list of values to choose from (possibly with scroll bar)
Check Box		A box allowing the user to set a yes/ no, true/ false value
Option Button		A button (often referred to as *radio button*) allowing the user to set a yes/ no, true/ false value. Usually used in *groups* (see below)
Toggle Button		A button allowing the user to set a yes/ no , true/ false value. The button appears pressed in when "on" and out when "off."
Frame		Used to create group of option buttons. This allows only one option button at a time to be active.
Command Button		A button for running a command. Most commonly used to **OK** or **Cancel** the UserForm.

Name of Control	Button Image	Description
Tab Strip		A TabStrip is used to view different sets of information for related controls
MultiPage		Used generally for large amounts of data that needs to be shown in separate tabs
Spin Button		Allows the user to select a numerical value by clicking up and down buttons. The Spin Button needs a text box to display (return) its values.
Scroll Bar		Similar to a Spin Button but in the form of a bar with a sliding "lever."
Image		Displays a picture. File formats vary with Excel versions but most common are: **.bmp**, **.jpg**, **.wmf**, **.gif**, **.ico**.
RefEdit		Similar to a text box control but contains a button at the right that collapses the form thus allowing easier selection of cells behind it.

It is advisable to have clear in your mind what controls the form should contain before you start. Design the form on paper before you start.

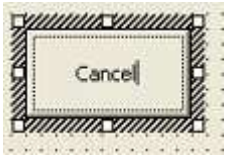
Procedures

1. Click in the Toolbox, the button for the control that you want to draw on the form.
2. Click the mouse pointer (black cross) on the form where you want the top left corner of the control to be placed.
3. Point to the grey border on the edge of the control (mouse pointer changes to a crosshair. 

4. Click and drag to move the control, if necessary.
5. Point to one of the white square on the corners or edges of the control (mouse pointer changes to a double headed arrow. 

6. Click and drag to resize the control, if necessary.

7. In the case of:
   - Labels
   - Command buttons
   - Check boxes
   - Options buttons

   Click on any visible text (Caption property) and replace it with a description of what the control stores or does, eg. a command button might have the text **OK** or **Cancel** on it. 

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

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

> Double-clicking a button on the Toolbox toolbar allows multiple controls of that type to be drawn on the form. Clicking the button again (or pressing ESC) cancels this operation.

FORM CONTROLS DESIGN TOOLS AND TECHNIQUES

Grouping

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

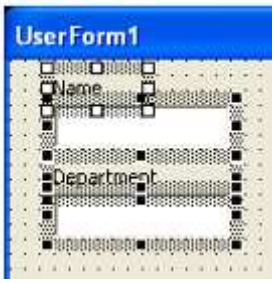
Controls can also be grouped together permanently.

Procedures

1. Click the first control that you want to include in the group.
2. Hold down the SHIFT key.
3. Click the next control that you want to include in the group.

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

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



Grouped Labels and Text Boxes on a UserForm

To permanently group controls together, select them as described above and then click the **Format** menu, **Group** command in the VB Editor.

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

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

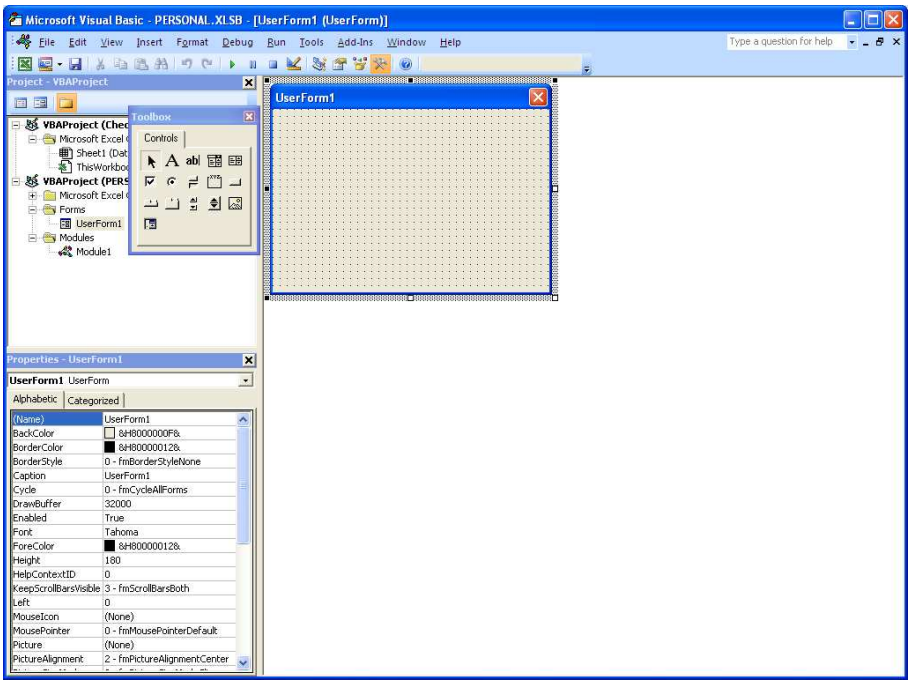
The UserForm Toolbar in the VB Editor

CONTROL PROPERTIES

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

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

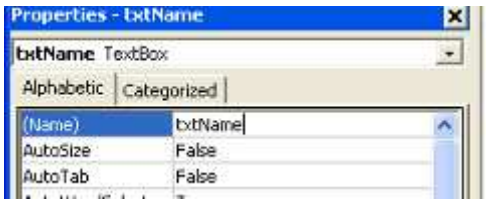
A comprehensive list of all properties for all form control would be too long and unnecessary for this booklet, but help can be sought by clicking onto a property in the properties pane and pressing **F1**.



VB Editor showing the Properties Pane in lower left corner

If the Properties pane is not visible, click **View** – **Properties**, press **F4** or click the **Properties** button  on the VB Editor toolbar.

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



Adding a Name Property to a Text Box Control

Control Naming Convention

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

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

Control Type	Prefix
Label	**lbl**
Text Box	**txt**
Combo Box	**cbo**
List Box	**lst**
Check Box	**chk**
Option Button	**opt**
Command Button	**cmd** or **btn**
Spin Button	**spn**
Scroll Bar	**scr**
Image	**img**

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

Procedures (naming a control)

1. Launch the VB Editor.

2. Identify the workbook containing the UserForm that you want to modify control properties for.
3. Display the form.
4. Select a control that you want to name.
5. Click the **Name** box in the **Properties** pane.
6. Type a name for the control, bearing in mind the naming convention described above.
7. Press **Enter**.

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

PROGRAMMING A USERFORM

Discussion

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

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

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

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

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

Beware of renaming a form or control after the event procedures have been written. The name of the procedure will not be automatically renamed and hence, the event procedure will not run as expected.

It is good practice to name controls as described in the previous topic before writing event code.

Procedures

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

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

http://msdn.microsoft.com/en-us/library/aa220733(office.11).aspx

The Cancel Command Button

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

```
Unload frmDataEntry
```

The OK Command Button

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

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

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

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

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

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

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

frmDataEntry.txtName.Value

The word Me can be used in place of the full form name. Me meaning "me, the form that I am on!" So instead of...

frmDataEntry.txtName.Value

you can use...

Me.frmDataEntry.txtName.Value

Text Boxes

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

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

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

Range("A2") = Me.txtName

Combo Boxes and List Boxes

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

Range("A3") = Me.cmbDepartment

or

Range("A4") = Me.lstLocation

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

Workbooks.Open Filename:= Me.lstFileNames

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

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

or

ChDir "s:\Excel\Stats\"

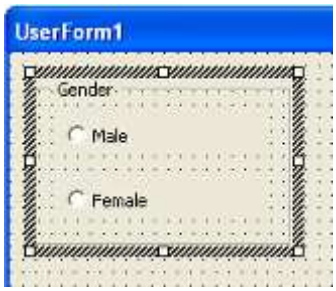
Workbooks.Open Filename:= Me.lstFileNames

In the following example, a combo box has been placed on the form to give the user a selection of fonts to use. Assuming that the range to format is A1 to E1, the code would be as follows:

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

Option Buttons

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



A Frame containing two Option Buttons

Invariably, a decision will need to be made which option button is selected and thus, an **If structure** used to carry out the necessary actions.

In the following example, a frame has been set up on a form with two option buttons, one for selecting male and one for selecting female. Whichever of the two is chosen must be placed into cell B4 as M or F.

```
If Me.optMale = True Then

    Range("B4").Value = "M"

Else

    Range("B4").Value = "F"

End If
```

Check Boxes

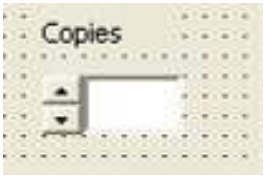
Because a check box has two conditions – true or false, an If statement has to be used to evaluate them. In the following example, a check box has been used for the user to state whether to print the workbook or not.

```
If Me.chkPrint.Value = True Then

    ActiveDocument.PrintOut

End If
```

Spin Buttons and Scroll Bars

A spin button or scroll bar control does not have a built in way of viewing its value. Under usual circumstances, a text box has be placed beside the control to show its current value.

The picture below, a spin button and a text box have been placed on a UserForm side by side.



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

Procedures

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

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

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

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

```
ActiveSheet.PrintOut Copies:=Me.spnCopies
```

FORM EVENTS

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

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

Example A

```
Private Sub UserForm_Initialize()

    With Me.cmbLocations

        .AddItem "Belfast"
        .AddItem "Cardiff"
        .AddItem "Edinburgh"
        .AddItem "London"

    End With

End Sub
```

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

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

Example B

```
Private Sub UserForm_Initialize()

    Me.txtLocation.Value = "London"
    Me.chkPrint.Value = True

End Sub
```

DISPLAYING A USERFORM

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

```
Sub DisplayForm()

    frmDataEntry.Show

End Sub
```

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

Procedures

1. Launch the VB Editor.
2. Identify the workbook containing the form that you want to show.
3. Insert a module sheet in the workbook.
4. Type the following code:

   ```
   Sub ShowForm
           <formname>.Show
   End Sub
   ```

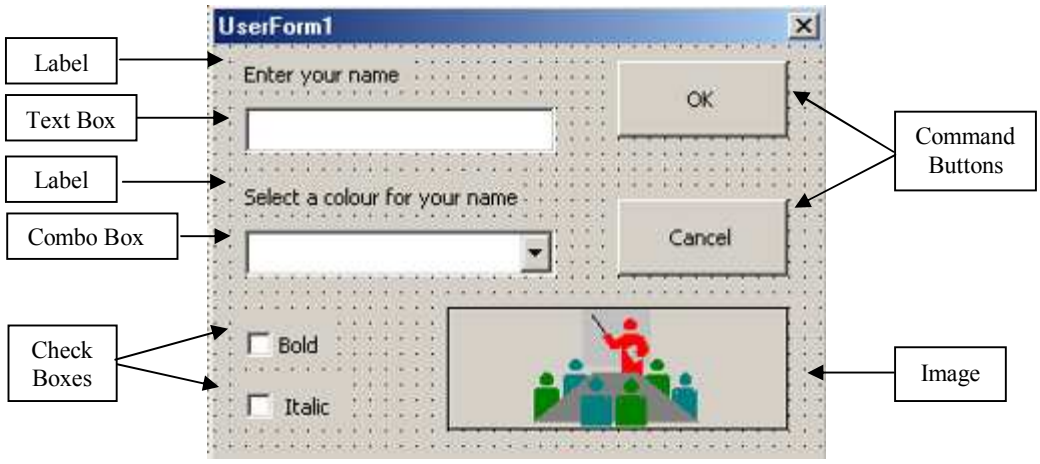
5. Run the userform

EXERCISE

CREATING A USERFORM

Task 1 - To create a UserForm to prompt for Text, ComboBox and CheckBox information.

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



3. Set properties to the objects as follows:

Control	Property	
CommandButton 1	Name:	btnOK
	Caption:	OK
CommandButton 2	Name:	btnCancel
	Caption:	Cancel
Label 1	Caption:	Enter your name
Text Box	Name:	txtInput
Label 2	Caption:	Select a colour for your name
ComboBox	Name:	comColour
CheckBox 1	Name:	chkBold
	Caption:	Bold
CheckBox 2	Name:	chkItalic
	Caption:	Italic
UserForm	Name:	frmDataEntry

Caption: Data Entry

4. Assign an "on-click event" (what will happen when the control is clicked) to the Cancel buttons by double clicking it and typing the following code:

```
Unload frmDataEntry
```

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

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

```
With frmDataEntry.comColours
        .AddItem "Red"
        .AddItem "Blue"
        .AddItem "Green"
End With
```

7. Return to the form design window.

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

```
Range("A1").Value = frmDataEntry.txtInput.Value
Select Case frmDataEntry.comColours
    Case Is = "Red"
    Range("A1").Font.Color = vbRed
    Case Is = "Blue"
    Range("A1").Font.Color = vbBlue
    Case Is = "Green"
    Range("A1").Font.Color = vbGreen
End Select
If frmDataEntry.chkBold = True Then
    Range("A1").Font.Bold = True
    Else
    Range("A1").Font.Bold = False
End If
If frmDataEntry.chkItalic = True Then
```

```
        Range("A1").Font.Italic = True
        Else
        Range("A1").Font.Italic = False
    End If
    Unload frmDataEntry
```

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

```
Sub EnterData()
        frmDataEntry.Show
End Sub
```

10. Create a custom button on your toolbar to run the **EnterData** macro and check correct data entry in cell A1. Correct any code, if necessary.

11. Save and close the file.

APPENDIX I – CREATING AN ADD-IN FOR SUB PROCEDURES

Discussion

In order for a workbook containing sub procedure(s) to be used as an Add-in, there must be a way for the Add-in to add a menu or buttons to run the procedure(s) when it is installed.

In order to achieve this, the following two (or similar) procedures needs to be added to the **ThisWorkbook** Excel Object module sheet as **Private Subs** (one that is not seen from the Excel side because it is not relevant to run it from there).

The Add-in can then be protected, saved and installed as described in lesson 6 on page 59.

```
Option Explicit
Dim cControl As CommandBarButton
-----------------------------------------------------------------
Private Sub Workbook_AddinInstall()
' This procedure adds a menu command to run the procedure when the Add-in is installed
On Error Resume Next 'Just in case
   'Delete any existing menu item that may have been left
   Application.CommandBars("Worksheet Menu Bar").Controls("My Code").Delete
   'Add the new menu item and set a CommandBarButton variable to it
   Set cControl = Application.CommandBars("Worksheet Menu Bar").Controls.Add
   'Work with the variable
      With cControl
         .Caption = "My Code"
         .Style = msoButtonCaption
         .OnAction = "MyGreatMacro"
         'Macro must be stored in a standard module in the Add-in file
      End With
On Error GoTo 0
End Sub
```

```
Private Sub Workbook_AddinUninstall()
' This procedure removes the menu command when the Add-in is un-installed
On Error Resume Next 'In case it has already gone.
   Application.CommandBars("Worksheet Menu Bar").Controls("Super Code").Delete
   On Error GoTo 0
End Sub
```

APPENDIX II – LIST OF TRAPPABLE ERRORS AND THEIR CODES

Source:

http:/ / msdn.microsoft.com/ en-us/ library/ ms234761(VS.80).aspx

Code	Message
3	Return without GoSub
5	Invalid procedure call
6	Overflow
7	Out of memory
9	Subscript out of range
10	This array is fixed or temporarily locked
11	Division by zero
13	Type mismatch
14	Out of string space
16	Expression too complex
17	Can't perform requested operation
18	User interrupt occurred
20	Resume without error
28	Out of stack space
35	Sub, Function, or Property not defined
47	Too many code resource or DLL application clients
48	Error in loading code resource or DLL
49	Bad code resource or DLL calling convention
51	Internal error
52	Bad file name or number
53	File not found
54	Bad file mode
55	File already open
57	Device I/ O error
58	File already exists
59	Bad record length

61	Disk full
62	Input past end of file
63	Bad record number
67	Too many files
68	Device unavailable
70	Permission denied
71	Disk not ready
74	Can't rename with different drive
75	Path/ File access error
76	Path not found
91	Object variable or With block variable not set
92	For loop not initialized
93	Invalid pattern string
94	Invalid use of Null
97	Can't call Friend procedure on an object that is not an instance of the defining class
98	A property or method call cannot include a reference to a private object, either as an argument or as a return value
298	System resource or DLL could not be loaded
320	Can't use character device names in specified file names
321	Invalid file format
322	Can't create necessary temporary file
325	Invalid format in resource file
327	Data value named not found
328	Illegal parameter; can't write arrays
335	Could not access system registry
336	Component not correctly registered
337	Component not found
338	Component did not run correctly
360	Object already loaded
361	Can't load or unload this object
363	Control specified not found
364	Object was unloaded
365	Unable to unload within this context

368	The specified file is out of date. This program requires a later version
371	The specified object can't be used as an owner form for Show
380	Invalid property value
381	Invalid property-array index
382	Property Set can't be executed at run time
383	Property Set can't be used with a read-only property
385	Need property-array index
387	Property Set not permitted
393	Property Get can't be executed at run time
394	Property Get can't be executed on write-only property
400	Form already displayed; can't show modally
402	Code must close topmost modal form first
419	Permission to use object denied
422	Property not found
423	Property or method not found
424	Object required
425	Invalid object use
429	Component can't create object or return reference to this object
430	Class doesn't support Automation
432	File name or class name not found during Automation operation
438	Object doesn't support this property or method
440	Automation error
442	Connection to type library or object library for remote process has been lost
443	Automation object doesn't have a default value
445	Object doesn't support this action
446	Object doesn't support named arguments
447	Object doesn't support current locale setting
448	Named argument not found
449	Argument not optional or invalid property assignment
450	Wrong number of arguments or invalid property assignment
451	Object not a collection
452	Invalid ordinal

453	Specified code resource not found
454	Code resource not found
455	Code resource lock error
457	This key is already associated with an element of this collection
458	Variable uses a type not supported in Visual Basic
459	This component doesn't support the set of events
460	Invalid Clipboard format
461	Method or data member not found
462	The remote server machine does not exist or is unavailable
463	Class not registered on local machine
480	Can't create AutoRedraw image
481	Invalid picture
482	Printer error
483	Printer driver does not support specified property
484	Problem getting printer information from the system. Make sure the printer is set up correctly
485	Invalid picture type
486	Can't print form image to this type of printer
520	Can't empty Clipboard
521	Can't open Clipboard
735	Can't save file to TEMP directory
744	Search text not found
746	Replacements too long
31001	Out of memory
31004	No object
31018	Class is not set
31027	Unable to activate object
31032	Unable to create embedded object
31036	Error saving to file
31037	Error loading from file

APPENDIX III – DEBUG.ASSERT

Discussion

In Excel 2000 and later, you can use **Debug.Assert** statements to cause the code to break if a condition is not met. The syntax for Debug.Assert is:

Debug.Assert (condition)

...where condition is some VBA code or expression that returns True (any numeric non-zero value) or False (a zero value). If condition evaluates to False or 0, VBA breaks on that line (see Breakpoints, page 76). For example, the following code will break on the Debug.Assert line because the condition (X < 100) is false.

```
Dim X As Long
X = 123
Debug.Assert (X < 100)
```

Debug.Assert is a useful way to pause code execution when special or unexpected conditions occur. It may seem backwards that Debug.Assert breaks execution when condition is False rather than True, but this peculiarity traces its roots back to early C-language compilers.

Your end users do not want the code to enter break mode under any circumstances, so be sure to remove the statements before distributing your code, or use Conditional Compilation to create "release" and "debug" versions of your project. Note that Debug.Assert is not available in Excel97 or earlier versions.

Conditional Compilation

While not directly part of debugging code, conditional compilation allows you to create "debug" and "release" versions of your code. For example, you may want to include message boxes, or Debug.Print or Debug.Assert statements while you are developing and testing your code, but you do not want those to be active when you release the code to users. VBA allows you to include or exclude blocks of code with a technique called conditional compilation. Conditional compilation uses If, Then, and Else statements to include or exclude a block of code. First, you want to create a compiler variable called, for example, DEBUG_ . Use the #CONST directive to create the variable.

```
#CONST DEBUG_ = True
```

Then, delimit blocks using the compiler directives to include various blocks of code. For example,

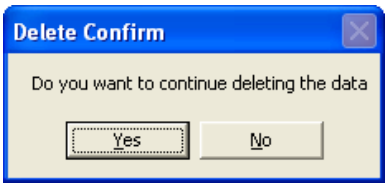
```
#If DEBUG_ Then
   Debug.Assert (X<100)
#End If
```

Note the use of the # character. In your development version, keep the value of DEBUG_ equal to True. When you are ready to release the code to end users, set this one constant value to False to prevent the Debug.Assert statement from even being included in the compiled code.

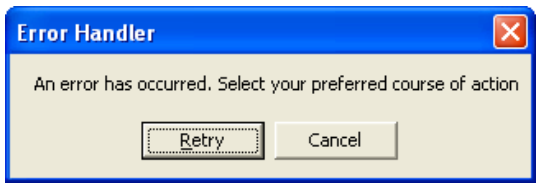
APPENDIX IV – ADDING INTERACTIVITY TO A MESSAGE BOX

Discussion

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



Example A



Example B

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

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

Prompt | Buttons | Title

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	Icon

vbCritical	16	Critical Message icon.	
vbQuestion	32	Warning Query icon.	
vbExclamation	48	Warning Message icon.	
vbInformation	64	Information Message icon.	

Default Button:

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

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

vbYesNo + vbQuestion + vbDefaultButton2

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

4 + 32 + 256

It is easier, however, to remember the vb constants, viz. vbYesNo + vbQuestion + DefaultButton2.

Procedures

1. Position the cursor in the sub procedure code where you want to place the statement.

2. Type a variable name to store the value of whichever button is clicked in the message box, eg. response.	
3. Type **=**.	
4. Type **MsgBox**.	
5. Type an opening bracket **(** .	
6. Type a speech mark (**Shift 2**).	
7. Type a prompt for the message box, ie. the message that you want it to display, eg. Do you want to continue?	
8. Type a speech mark (**Shift 2**).	
9. Type a comma.	
10. Type the necessary value to indicate which buttons you want the message box to display, eg. vbYesNo.	
11. If you wish to add an icon and/ or default button to the message box, type a plus symbol (**+**).	If you do not wish to add an icon and/ or a default button to the message box, go to 15 below.
12. Type the necessary value to indicate which icon to display, eg. vbQuestion	
13. Type a plus symbol (**+**).	
14. Type the necessary value to indicate which default button you wish to set, eg.vbDefaultButton2	
15. Type a comma.	
16. If you wish to add a title to the message box, type a comma	If you do not wish to add a title to the message box, go to 20 below.
17. Type a speech mark (**Shift 2**).	
18. Type the title text.	
19. Type a speech mark (**Shift 2**).	
20. Type a closing bracket **)** .	
21. Press **Enter**.	
22. Add additional code as necessary.	

Responding to an Interactive Message Box

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

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

In the following example, the message box offers a yes/ no response. If the user clicks *yes*, then the procedure will delete all the data from the sheet. If the user clicks *no*, the procedure will terminate.

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

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

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

Procedures

1. (The procedure below assumes a message box containing Yes and No buttons. It can be adapted, however, to respond to any set of buttons (eg. vbOKCancel, vbRetryCancel etc.))

2. Position the cursor in the sub procedure code where you want to place the statement.
3. Create an interactive message box as described in the previous topic of this lesson.
4. Press **Enter**.
5. Type **If**.
6. Type a **space**.
7. Type the variable name that you have used to store the response from the message box.
8. Type a **space**.
9. Type = vbYes.
10. Type a **space**.
11. Type **Then**.
12. Type the statements that you want the sub procedure to perform if the user has clicked the **Yes** button.
13. Press **Enter**.
14. Type **Else**.
15. Press **Enter**.
16. Type the statements that you want the sub procedure to perform if the user has clicked the **No** button.
17. Press **Enter**.
18. Type End If.
19. Press **Enter**.
20. Add additional code as necessary.

APPENDIX V – SOLUTIONS TO EXERCISES

Review Exercise (Page 5)

```
Sub CreateTable()

Sheets.Add

' Add data
        Range("B1").Value = "USA"
        Range("C1").Value = "Europe"
        Range("A2").Value = "Sales"
        Range("A3").Value = "Costs"
        Range("A4").Value = "Profit"

' Add formulas
        Range("B4").Formula = "=B2-B3"
        Range("C4").Formula = "=C2-C3"

' Add formatting
        Range("B1:C1").Font.Bold = True
        Range("A2:A4").Font.Italic = True
        Range("A4:C4").Interior.ColorIndex = 15

End Sub
```

```
Sub TestProfit()

' Declare variable
Dim cel As Variant

' Start looping through each variable in range
   For Each cel In Range("B4:C4")

' Test each variable
      If cel.Value >= 15000 Then

         cel.Font.Bold = True

      Else

         cel.Font.Color = vbRed

      End If

   Next cel

End Sub
```

Working with the Range Object (Page 18)

```
Sub LayoutTable()

' Select a cell on the table before running this macro

        ActiveCell.CurrentRegion.Select
        Selection.ColumnWidth = 12
        Selection.Columns(1).Font.Color = vbBlue
        Selection.Rows(1).HorizontalAlignment = xlCenter

End Sub
```

```
Sub FormatNumbers()

' Select a cell on the table before running this macro

        ActiveCell.CurrentRegion.Select
        Selection.Offset(1, 1).Select
        Selection.Resize(Selection.Rows.Count - 1, Selection.Columns.Count - 1).Select
        Selection.NumberFormat = "£#,##0.00"

End Sub
```

```
Sub LastCell()

' Select a cell on the table before running this macro

        Selection.CurrentRegion.Select
        Selection.Cells(Selection.Cells.Count).Select
        ActiveCell.Font.Size = 14
        ActiveCell.Interior.Color = vbYellow
        ActiveCell.EntireColumn.AutoFit

End Sub
```

```
Sub RunAllMacros()

' Select a cell on the table before running this macro

        Call LayoutTable
        Call FormatNumbers
        Call LastCell

End Sub
```

Variables and Arrays (Page 41)
Task 1

```
Sub CreateNewForecast()

Dim arrMonthName As String
Dim arrVals(1 To 4) As String

   Sheets("Template").Copy After:=Sheets(Sheets.Count)

   arrMonthName = InputBox("Enter the name for the sheet")
   ActiveSheet.Name = arrMonthName

   For i = 1 To 4

      arrVals(i) = InputBox("Enter value for week " & i)

   Next i

   For i = 1 To 4

      Cells(2, 1 + i).Value = arrVals(i)

   Next i

End Sub
```

Task 2

```
Const PURPLE As Byte = 29
Const ORANGE As Byte = 45
Const ROSE As Byte = 38
Const BLUE As Byte = 5

Sub ApplyColours()

' Select a cell on the table before running this macro
        ActiveCell.CurrentRegion.Rows(1).Interior.ColorIndex = PURPLE
        ActiveCell.CurrentRegion.Rows(1).Font.ColorIndex = ROSE

        ActiveCell.CurrentRegion.Columns(1).Interior.ColorIndex = ORANGE
        ActiveCell.CurrentRegion.Columns(1).Font.ColorIndex = BLUE

End Sub
```

Task 3

```
Sub TransferData()

Dim arrData(1 To 6, 1 To 4)
Dim iRows As Byte
Dim iCols As Byte


For iRows = 1 To 6

   For iCols = 1 To 4

      arrData(iRows, iCols) = Cells(iRows + 1, iCols + 1).Value

   Next iCols

Next iRows

Sheets("Net").Activate

For iRows = 1 To 6

   For iCols = 1 To 4

      Cells(iRows + 1, iCols + 1).Value = arrData(iRows, iCols) * 0.8

   Next iCols

Next iRows


End Sub
```

User-Defined Functions (Page 57)
Task 1

```
Function DegF(TempC)

   DegF = TempC * 9 / 5 + 32

End Function
```

```
Function DegC(TempF)

   DegC = (TempF - 32) * 5 / 9

End Function
```

```
Function Hypot(x, y)

   Hypot = Sqr(x ^ 2 + y ^ 2)

End Function
```

```
Function Gasbill(UnitsUsed, LowUnits, HighRate, LowRate)

Dim highUnits As Integer

   highUnits = UnitsUsed - LowUnits
   Gasbill = (LowUnits * LowRate) + (highUnits * HighRate)

End Function
```

Task 2

```
Function CountText(RangeToCount)


For Each numb In RangeToCount

   If Not IsNumeric(numb) Then

      CountText = CountText + 1

   End If

Next numb

End Function
```

Dealing with Potential Errors (Page 95)

```
Sub CircData()

Const PI = 3.142

Radius = InputBox("Type the radius of your circle in centimetres", "Circle Data")

   If IsNumeric(Radius) Then

      CircArea = PI * (Radius ^ 2)
      CircCircumf = 2 * PI * Radius

      MsgBox "The area of the circle is: " & CircArea & vbCrLf & vbCrLf & _
         "The circumference of the circle is: " & CircCircumf

   Else

      MsgBox "Input not valid"

   End If

End Sub
```

```
Sub DeleteOldFile()

Dim fileToRemove As String

On Error GoTo errhandle

fileToRemove = InputBox("Type name of file to delete from the current folder", "Delete File")

If fileToRemove = "" Then

   Exit Sub

Else


   Kill ThisWorkbook.Path & "\" & fileToRemove & ".xls"
   MsgBox "File successfully deleted"

End If


Exit Sub
errhandle:

   If Err.Number = 53 Then
```

```
        response = MsgBox("File not found. Do you want to try again?", vbYesNo)

        If response = vbYes Then

            fileToRemove = InputBox("Type name of file to delete from the current folder", "Delete File")
            Resume

        End If

    End If

    If Err.Number = 70 Then

        MsgBox "File is currently open. Close the file and run the macro again"

    End If

End Sub
```

INDEX

V