



## Welcome to the VBA training course from “Excel for Managers”.

This VBA training workshop has been designed for people who need to know how to create robust, professional VBA code, but have no intention of becoming full time programmers.

It is a rigorous, fast-track course for consultants, analysts and accountants with proven Excel skills and concentrates on the skills and knowledge needed to create professional solutions to real world problems. It is NOT a course for junior programmers.

After grasping the core VBA skills, you will be challenged with a start-to-finish real world scenario. As you undertake the project, and we discuss the various challenges, trade-offs and options, you will learn a simple, logical and methodical nine-point plan that will enable you to design, program or manage Excel VBA projects with full confidence.



# Contents

Course objectives .....	1
VBA – An overview .....	2
VBA vs. Standard Excel Functionality .....	2
Should I Use Excel Functions or VBA Code? .....	2
When should I use VBA? .....	3
Should I use Excel at all? .....	4
The Macro Recorder .....	5
Personal Macro Workbook .....	5
Viewing your code in the VBE .....	6
The Visual Basic Editor’s Code Window .....	7
Making your code generic .....	8
Intellisense .....	11
Using Intellisense for Parameters & Constants .....	12
VBA fundamentals: Objects .....	13
The Workbook, Worksheet and Range Objects .....	13
Defining Objects .....	14
Defining Workbook and Worksheet Objects .....	16
Defining a Worksheet Range .....	18
Implicit and Explicit declaration of objects .....	20
Method & Property Examples .....	21
Worksheets .....	21
Ranges .....	23
Workbooks .....	24
Variables, Arrays & Constants .....	25
Standard Variables .....	26
Object Variables .....	26
Why bother pre-defining Variables? .....	27
Setting & Using Variables .....	28
Arrays .....	31
Declaring Arrays .....	31
Populating Arrays .....	31
Control Structures & Program Flow .....	33
IF (Elseif, Else) .....	34
Select Case() .....	35
For Next Loop .....	36
For Each Loop() .....	37
Do Until Loop and Do While Loop .....	38
With, End With .....	39
Nesting Code .....	39
Scope & Code Location .....	40
Functions, Routines & Arguments .....	42
Definitions .....	42
Passing Parameters to a Function .....	43

Multiple and Optional Parameters .....	43
Passing Parameters back and forth between Routines.....	44
Error Handling .....	45
The VBE in detail .....	47
Manipulating Text .....	49
Triggering code with Events.....	51
Events with Parameters.....	52
Interacting with the user.....	53
Msgbox .....	53
Application.StatusBar .....	54
Using the generic toolbar and userform workbook.....	55
Legal Stuff.....	55
Toolbars / Toolbar buttons.....	56
Adapting the generic userform to your requirements .....	59
Modal / Modeless Userforms.....	62
Looking at projects.....	63
The three elements of every Excel VBA project .....	63
How to spot a well (or badly) designed system.....	64
The Nine Point Project Plan .....	67
Before you start ... ..	68
Planning .....	69
1. Best approach?.....	69
2. How can that be done? .....	69
Confirming .....	72
3. Proof of Concepts.....	72
4. User Interface.....	73
Coding & Designing.....	74
5. Output .....	74
6. Input .....	74
7. Data Manipulation.....	74
Finalising.....	75
8. Final User Testing .....	75
9. Handover .....	75
VBA & Charts.....	76
Getting additional help .....	77
Top 5 Excel Resources .....	78
Designing for Bob.....	80

## Course objectives

By the end of this course you will be able to ...

- Understand Excel VBA's strengths, weaknesses and limitations.
- Plan, write, debug and extend well structured VBA code.
- Follow a simple methodical process to design, code and implement an Excel VBA system.
- Recognise tell-tale signs of well and badly designed workbooks and VBA projects.
- Appreciate the factors that impact the complexity of a VBA project.
- Design or specify projects for other people.
- Know where to get additional help.

You will NOT be expected to remember specific details of the functions, routines or syntax covered in the workshop.

### **A few words about this documentation...**

Whilst this document contains many of the pointers, examples and discussion items we will be working with throughout the workshop, it has not been designed as a course book.

In order to give you the maximum exposure to Excel VBA and promote discussion on the topics covered we will be working directly with the Visual Basic Editor throughout the course.

This guide has simply been designed as a reminder of the topics discussed and for reference during or after the workshop.

### **A note to readers, *not* undertaking this course from a qualified trainer ...**

This document is available freely online from several locations, and you are welcome to peruse its contents, though not to copy or change them. Although we would ideally recommend you undertake this course under the instruction of a qualified trainer and experienced programmer, we sincerely hope that it is of use in your understanding of VBA in a commercial environment. ExcelForManagers.com is a consultancy specializing in spreadsheets and spreadsheet risk management, and we are committed to promoting the use of VBA in a safe and responsible way. Please visit our website at [www.ExcelForManagers.com](http://www.ExcelForManagers.com) or contact us for assistance in your Excel issues or training requirements including VBA for Excel 2007.

### **A note to qualified trainers or VBA programmers ...**

If you are interested in presenting this course yourself (or an updated 2007 VBA version) we can license it to *suitably qualified* personnel including the accompanying spreadsheets and documentation. Please contact us for details at [Enquiries@ExcelForManagers.com](mailto:Enquiries@ExcelForManagers.com).

## VBA – An overview

VBA is the programming language that resides within Excel. It enables the automation of complex tasks and the construction of complete applications that can make use of the full power of Excel.

VBA also resides in other host applications including Word, PowerPoint and Access, so many of the skills learnt here can be applied to other Microsoft Office software.

## VBA vs. Standard Excel Functionality

When faced with an Excel project, experienced Excel users who are not programmers often use complex spreadsheets, hundreds of formulae and little-known tricks to create their solution, whilst professional programmers tend to drop straight into VBA and use Excel simply to display the solution.

So which approach is best?

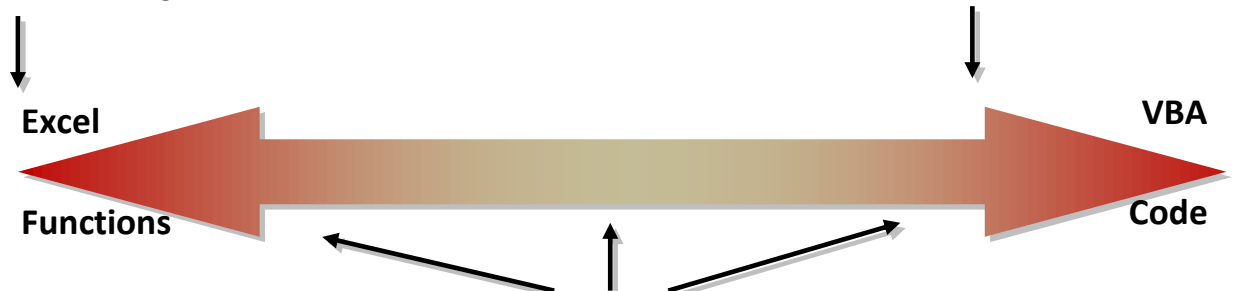
The answer is that it depends on the requirement. However, the most effective solution is *usually* one that combines the built-in power and functions of Excel with the automation and control of VBA.

Using VBA or Excel inefficiently is one of the primary sources of project failure, and the decision on where to pitch your solution on an imaginary Excel functions vs. VBA code line is worthy of active and deliberate consideration, before finally deciding upon your approach.

### Should I Use Excel Functions or VBA Code?

Non programmers have no choice but to design their solutions using standard Excel.

Computer professionals usually go for a mostly programmed solution.



True Excel professionals pitch their solution at the most appropriate mix of VBA code and Excel functionality for the requirement.

## When should I use VBA?

VBA can be made to perform most tasks in Excel. It gives complete control of what is displayed and calculated and can be made to be fast, efficient and effective. However, with a few exceptions, standard Excel functionality can be coerced (often in long or roundabout ways) to perform most of the functionality available to a VBA programmer. Therefore you will often be faced with the personal choice of using Excel or VBA for each aspect of your project. Detailed below is an advisory list of how you should tend towards using either Excel functions or VBA for various parts of a project.

**Note: This is a generic list of recommendations, and the right decision will depend on many factors. Base your decision on the specifics of your project and your relative Excel vs. VBA skill levels.**

### Use Excel functionality for ...

- Holding data (Unlike VBA variables, Excel data persists if the workbook is closed and re-opened)
- Simple data validation.
- Looking up associated data: e.g. `VLookup()`, `Offset()`, `Match()`
- Sorting data.
- Calculating data.
- Displaying and formatting results or calculated information.
- Pivot table functionality.
- Charts.

### Use VBA for ...

#### Interacting with the data ...

- Moving, copying or deleting data based on set conditions.
- Performing different calculations based on the data's content.
- Manipulating graph views and source data.
- Opening pre-specified data files and extracting known information.
- Checking complex data logic or validity.

#### Interacting with the data and the user combined ...

- Preventing the user continuing until defined criteria have been met.
- Showing and hiding data or sheets of data according to the user's function or ability.
- Wizard style applications where a defined process needs to be followed.
- Situations where several user decisions need to be met (especially when decision combinations can be incongruous)
- Manipulating the user's environment (e.g. adding sheets, selecting the relevant data)

#### Interacting with other applications ...

- Exporting graphs to PowerPoint.
- Importing database information, for example from SQL or Oracle.

***Pro-Tip***

Power users like to be able to 'see' what is going on in their application .... They like to be able to dig down into a worksheet and have the ability to see, understand and investigate each stage of the calculation process. For them seeing an Excel formula calculate their data is more important than the speed of calculation. Using worksheet functions instead of (usually faster and often more efficient) VBA to calculate data gives users a crucial level of comfort with the system, and gives you as the designer a superb opportunity to check the system is performing correctly at each stage of the calculation.

**Should I use Excel at all?**

Before we leave the topic of combining VBA and Excel's functionality to best effect, we really should consider when Excel might not be the right solution at all. Here are the three main factors that would give warning signs for the need to consider solutions outside of Excel / VBA.

**Security**

Excel is NOT secure. Passwords are useful and there are numerous clever ways of disguising and hiding data within Excel, but if security is of paramount importance then you should talk to your IT department about progressing to a client-server application where source data can be stored remotely and securely.... You might still be able to use Excel to open, manipulate and display the information.

**Multi-User Applications.**

Excel now has several increasingly useful (and complex) methods to update, share and control data between multiple users. Whilst these may meet your requirements, and/or you can certainly use VBA to create and maintain data integrity, a multi-user Excel system is a far-from-ideal solution. If your project is a linear process, (one where several users work on the data in turn and pass it on to the next user) then you may consider using a single Excel workbook as a data file that acts as an intermediary between the users, i.e. each user has their own different 'system' workbook that manipulates a single data workbook ready for manipulation by the next user. However if all the users need to manipulate the same data at the same time then you should be disinclined to use Excel, at least for the database aspect of the requirement.

**Web-Applications**

A question that often arises is whether Excel can be used effectively over an intranet or internet. Excel CAN be used directly in a browser such as Internet Explorer, but no VBA is available and there are other severe limits on its functionality. Our view is that Microsoft will need to address this issue over the next few years as the demand for more and more applications that run directly through a browser increases. Right now there is no simple, perfect solution. If you need web-based or intranet based interactivity with users, discuss this with your IT department, BEFORE starting your solution. Prompt them about 'Excel Services' which may be appropriate in some circumstances.

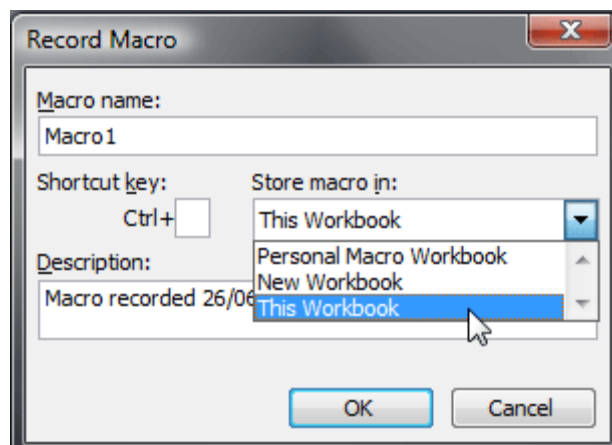
## The Macro Recorder

### *Pro-Tip*

Unless you develop solutions full time, it is likely that the macro-recorder will be the cornerstone of your programming syntax. Once you have learned to make this Excel generated code generic there is no problem whatsoever with this approach.

Note: All public macros are always available to run via Tools / Macro / Macros.

Most Intermediate Excel users have heard about or used the macro recorder. It is available via Tools / Macro / Record New Macro.



A variety of options is presented, one of which is the location of the code that will be created.

### **Personal Macro Workbook**

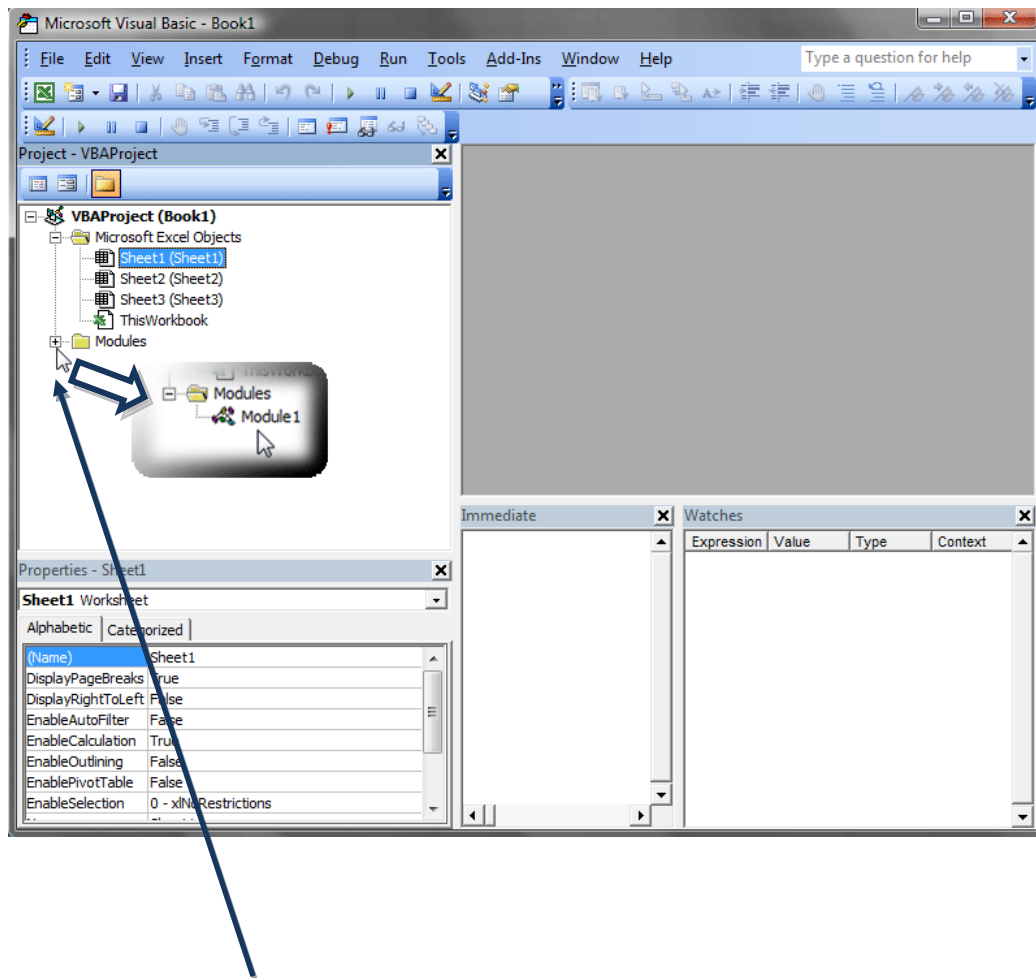
The Personal Macro Book is an Excel workbook that can be used to contain macros you use personally. If it has been created it is opened automatically alongside Excel and can be unhidden at any time via Windows /Unhide / Personal.xls. The Personal Macro Workbook has the disadvantage that the macros on which your work relies are only available to you. It can be messy to share these macros with your colleagues (naming issues).

For the duration of the workshop, and it is suggested in general, save the macro to 'This Workbook', which will be the active workbook at the time. Any saved code will then be kept with the workbook and available to all users when the workbook is shared.

## Viewing your code in the VBE

In one of your first workshop exercises, you will have created a macro that fits the worksheet contents onto a single page of A4 portrait paper, and includes the date in the centre of the footer. However to see this, or any other code we must enter into the world of the Visual Basic Editor (VBE).

Start the VBE from Tools / Macro / Visual Basic Editor. Your screen should look something like this.



The workbook name(s) will appear at the top right hand side of the screen, along with a tree structure of items. Right now all we are interested in is the code we have just written. Double click the + sign next to the word 'Modules' to reveal the tree structure underneath. From there double-click 'Module 1'. Your code will appear in the Code Window.

### Speed-Tip

Press Alt-F11 to start the VBE or alternate between VBA and Excel

## The Visual Basic Editor's Code Window

The VBE code window (shown in white) is your main interface with VBA. It is used to write, read and debug code.

Here, a dropdown list of subroutines available in the code module is available. Select one to view the code.

The colour of the writing in the VBE indicates how Excel is interpreting the text ...

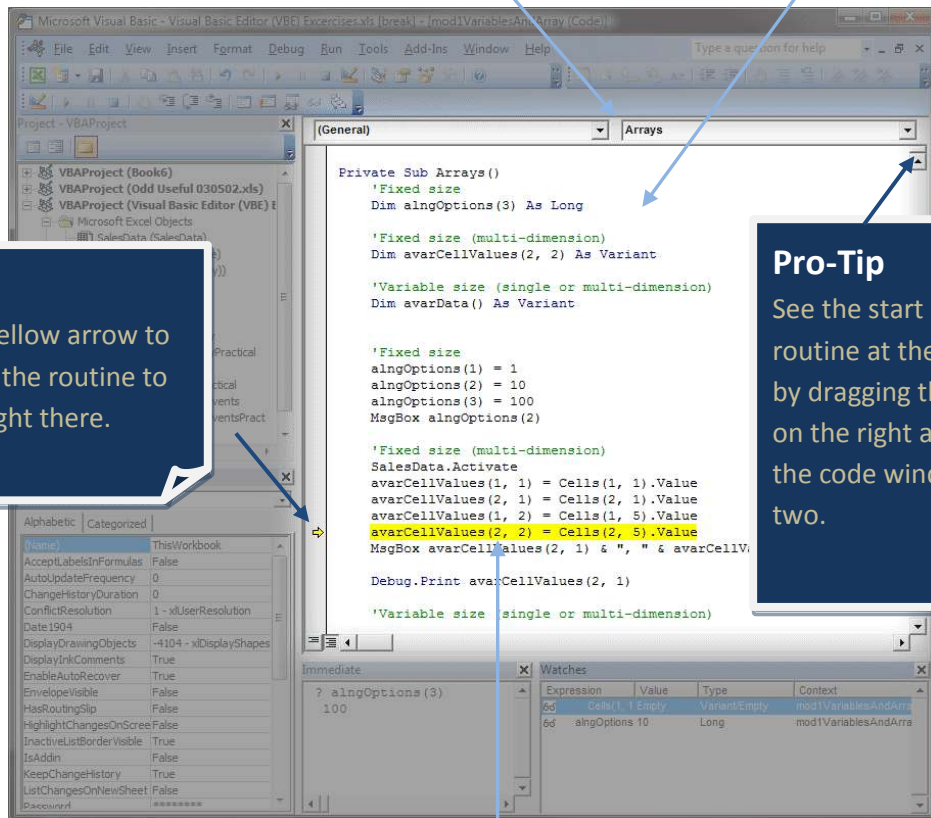
- Green: Comments (Start with a ' and are ignored by VBA)
- Blue: Reserved words with special meaning to VBA
- Black: User instructions analysed by VBA at run time

### Pro-Tip

Drag the yellow arrow to any line in the routine to jump straight there.

### Pro-Tip

See the start and end of a routine at the same time by dragging the scroll bar on the right and splitting the code window into two.



The yellow line indicates the line of code currently being processed. Start or re-start the code by selecting anywhere in the routine, then...

- F8: Run the code, one line at a time
- F5: Run the code to the end

## Making your code generic

The macro recorder is very accurate; however it is NOT very efficient. The code you generated will look something like this, which can be quite daunting to a non programmer

```

Sub Macro1 ()
    With ActiveSheet.PageSetup
        .PrintTitleRows = ""
        .PrintTitleColumns = ""
    End With
    ActiveSheet.PageSetup.PrintArea = ""
    With ActiveSheet.PageSetup
        .LeftHeader = ""
        .CenterHeader = ""
        .RightHeader = ""
        .LeftFooter = ""
        .CenterFooter = "&D"
        .RightFooter = ""
        .LeftMargin = Application.InchesToPoints(0.75)
        .RightMargin = Application.InchesToPoints(0.75)
        .TopMargin = Application.InchesToPoints(1)
        .BottomMargin = Application.InchesToPoints(1)
        .HeaderMargin = Application.InchesToPoints(0.5)
        .FooterMargin = Application.InchesToPoints(0.5)
        .PrintHeadings = False
        .PrintGridlines = False
        .PrintComments = xlPrintNoComments
        .CenterHorizontally = False
        .CenterVertically = False
        .Orientation = xlPortrait
        .Draft = False
        .PaperSize = xlPaperA4
        .FirstPageNumber = xlAutomatic
        .Order = xlDownThenOver
        .BlackAndWhite = False
        .Zoom = False
        .FitToPagesWide = 1
        .FitToPagesTall = 1
        .PrintErrors = xlPrintErrorsDisplayed
    End With
End Sub

```

... so let's break it down to something more manageable!

## Macro Name

```
Sub Macro1()
```

All recorder macros start with the word 'Sub' followed by a default name or one specified by you, and then a set of parenthesis. If you didn't give the macro a meaningful unique name when you recorded it you can change it now. The name must start with a letter and can include letters & numbers. An underscore must be used in place of any spaces. The name is not case sensitive.

```
With ActiveSheet.PageSetup  
    .PrintTitleRows = ""  
    .PrintTitleColumns = ""  
End With
```

The phrases 'With' and 'End With' combined, tell VBA that everything listed between them applies to the same 'thing'. The 'thing' in question is the item (object) that follows the initial 'With', which in this case is the page setup of the active sheet.

This code snippet tells Excel to set the title rows and columns (those rows and columns that are always printed at the top or left of EVERY printed sheet when there is more than one sheet top print) to nothing.

We will be covering the very useful 'With – End With' in more detail later.

## Eliminating code

Whilst we were recording the macro, we went into the Page Setup userform and the (rather zealous) macro recorder decided to take a note of EVERY page setup setting. For example in the With – End With code snippet above we did not tell VBA anything about PrintTitleRows, however it still recorded this setting for us. As we didn't want to 'change' anything to do with this aspect of the page setup, it is safe to delete those entire four rows of code.

### Pro-Tip

Eliminating the code that we didn't actually request will often cut a recorded macro's size by 80%-90%

Once you see how the macro recorder operates, it becomes easy to extend this logic ...

```
ActiveSheet.PageSetup.PrintArea = ""
With ActiveSheet.PageSetup
    .LeftHeader = ""
    .CenterHeader = ""
    .RightHeader = ""
    .LeftFooter = ""
    .CenterFooter = "&D"
    .RightFooter = ""
```

We didn't request to set the print area of the worksheet (we want to leave it how it was) so it is safe to delete that line too.

### Pro-Tip

When the superfluous code is deleted and the macro is run, the settings relating to the deleted code are left unchanged.

The same is true of the **.LeftHeader**, **CenterHeader** and so on..

Once we have renamed the macro and deleted the superfluous code we are left with something a lot more manageable. Note the comments (in green) have also been amended to make it easier to understand later. You can test run the new macro from Tools / Macro / Macros.

```
Sub SinglePortraitPageWithDateFooter ()

    'Sets up a single page portrait A4 sheet ready for printing
    'and inserts the system date into the footer.

    With ActiveSheet.PageSetup
        .CenterFooter = "&D"
        .Orientation = xlPortrait
        .PaperSize = xlPaperA4
        .FitToPagesWide = 1
        .FitToPagesTall = 1
    End With
End Sub
```

### Pro-Tip

When you start out in VBA, it can often be a process of trial and error which lines it is safe to remove: Make a copy of the original recorded macro before you amend it so you can easily re-insert lines of code until you get the required result.

## Intellisense

You may have noticed the liberal use of dots in the code created by the macro recorder. These dots are crucial to VBA (more on that when we get to Objects) but importantly to us they are the pre-cursor to one of the most helpful, time-saving, and clue-giving aspects of VBA .... Intellisense.

Intellisense is Microsoft's way of helping us programmers remember all the settings, operands and syntax available to us whilst we are coding.

This is best described by way of an example, so let's start by writing our own brand new macro.

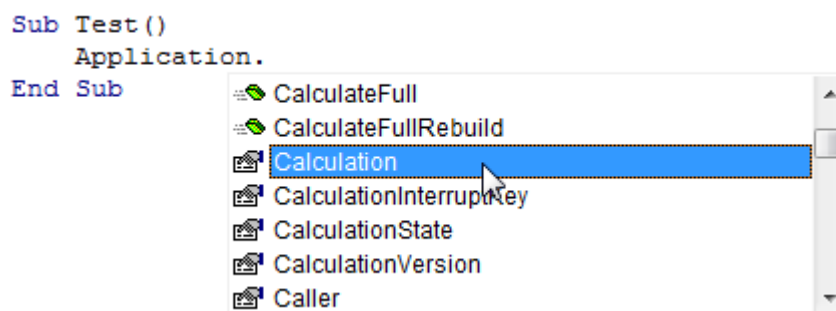
Underneath any other macros in Module 1, enter the exact text: `Sub Test` then press Return. You will see that VBA has entered parentheses after the word `Test` and also included a gap followed by the words 'End Sub':- This is not Intellisense at work, but it is helpful nevertheless. You should see something similar to this ...

```
Sub Test ()
End Sub
```

All our code for this routine must now be written *between* these two lines.

`Application` is VBA's word that describes its Excel environment. We are going to change the Excel Application's calculation setting from `Automatic` (its standard setting) to `Manual`, just as if we had changed it directly from the Excel options.

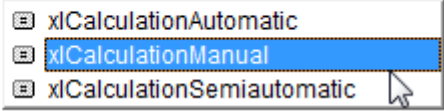
Start by inserting the word 'Application' on the line below our routine title, and follow it with a dot (.)



You will see that on entering the dot Intellisense immediately presents us with a list of all the possible settings and actions that can be viewed, performed or set on the Application. This list is very extensive. We want to set the 'Calculation' setting, so either scroll down to it or start by entering the first few letters 'CAL'. Once you see the word, double-click on it to let VBA insert the word for you.

We now want to set the calculation to the 'manual' setting, so add an equals ( = ) sign to the line. This time Intellisense immediately offers us the three options that are available. It is a simple matter to double click the required option and press return to complete our line of code.

```
Sub Test()
    Application.Calculation = =
End Sub
```



That's it! We used Intellisense to create a fully functional macro with little effort. You may wish to write another macro to set calculation to Automatic and run it (or reset it from Excel's options menu).

```
Sub Test()
    Application.Calculation = xlCalculationManual
End Sub
```

## Using Intellisense for Parameters & Constants

As well as informing us about settings and actions (in VBA these are known as properties and events), we can also use Intellisense to tell us about parameters and constants.

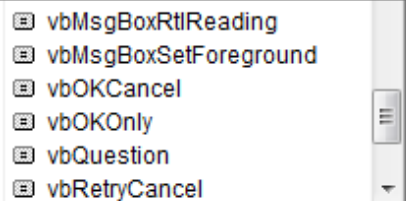
Msgbox is one of the ways to interact with the user, and whilst it can be used to display simple text, it can take several optional parameters to enhance the message.

In our test sub, start a new line with the word `Msgbox` then the first parenthesis ( `(` You should see the following Intellisense caption which informs you that you are about to enter the prompt (message) ...

```
Sub Test()
    msgbox (
End   MsgBox(Prompt, [Buttons As VbMsgBoxStyle = vbOKOnly], [Title], [HelpFile], [Context]) As VbMsgBoxResult
```

Type a message in quote marks (e.g. "Hello") and enter a comma

```
Sub Test()
    msgbox ("Hello",
End   MsgBox(Prompt, [Buttons As VbMsgBoxStyle = vbOKOnly], [Title], [HelpFile], [Context]) As VbMsgBoxResult
```



This time you are informed that you are entering the (type and text of) buttons the user will see, and presented with a list of options. Press escape and come out of the routine as we're finished for now, but carry on experimenting with Intellisense and use it wherever possible ... it is real time saver.

## VBA fundamentals: Objects

VBA is an object orientated language. That is to say that most projects can be described, and the solution programmed through a hierarchy (tree structure) of things (objects) and their properties.

Although there are hundreds of objects in Excel, there are just a handful that are key to over 90% of most programming requirements, and of these, there are three core objects; Workbook, Worksheet and Range.

We will work extensively with these objects during this workshop. It is essential that you are comfortable with them and how to manipulate and control these objects.

### The Workbook, Worksheet and Range Objects

These three objects are intrinsic to VBA programming. They are available as a simple hierarchy ...

- The Application (Excel) contains one or more Workbooks.
- Each Workbook contains one or more Worksheets
- Each Worksheet contains a range of cells (Anything from 1 row by 1 col to 65536 rows by 255 cols).

Once we have defined the workbook or worksheet or range in which we are interested, we can then tell VBA what needs doing ...

#### Properties

When we want to know or set something about our object, such as its name, its colour or its height we do this through the use of 'Properties'. Properties are analogous to nouns.

#### Methods

When we want to perform an action on an object, such as to create a new one, delete it, activate it or copy it we do this through the use of 'Methods'. Methods are analogous to verbs.

## Defining Objects

Excel VBA object code starts (either implicitly or explicitly) with the word 'Application'. An object 'tree' is then followed down to the specific required object using a dot at each level and once the required object is reached we can finally invoke a method or view/set a property.

We select or define specific Objects, invoke Methods and view or set Properties all via a single simple programming instruction ... the humble dot!

The following line of code shows the syntax used to define a range and show its 'Value' property by using a dot to separate each object and the final property. This message box displays the contents of cell A1 on the second worksheet of the first (or only) open workbook in Excel

```
MsgBox Application.Workbooks(1).Worksheets(2).Range("A1").Value
```

It follows that in order for VBA to work on a range of cells, we need to tell VBA not only the range, for example A1:C4, but also the Workbook and Worksheet (the hierarchy) to which the range belongs. Luckily Excel VBA offers us lots of useful ways in which to specify the required object.

### Pro-Tip

Using VBA to manipulate ranges that include merged cells in a worksheet is tricky (you need to treat the single cell as a contiguous range, and there are other cut/paste limitations). Avoid merging cells in any workbook where you intend to manipulate worksheets using VBA if at all possible.

### Explicitly specifying an object

You will be writing code that works at all levels in the object hierarchy ...

#### Application Level

E.g. Change the calculation method, maximise or refresh the screen.

#### Workbook Level

E.g. Saving, activating or protecting a workbook.

#### Worksheet Level

E.g. Add, rename or delete a worksheet.

... or you may wish to work with objects that sit within worksheets, for example

#### Cell Ranges

E.g. Change the values in a range, insert a row, copy & paste cells or change cell formatting.

#### ChartObjects()Charts()

E.g. Show or hide a chart or reset its source range.

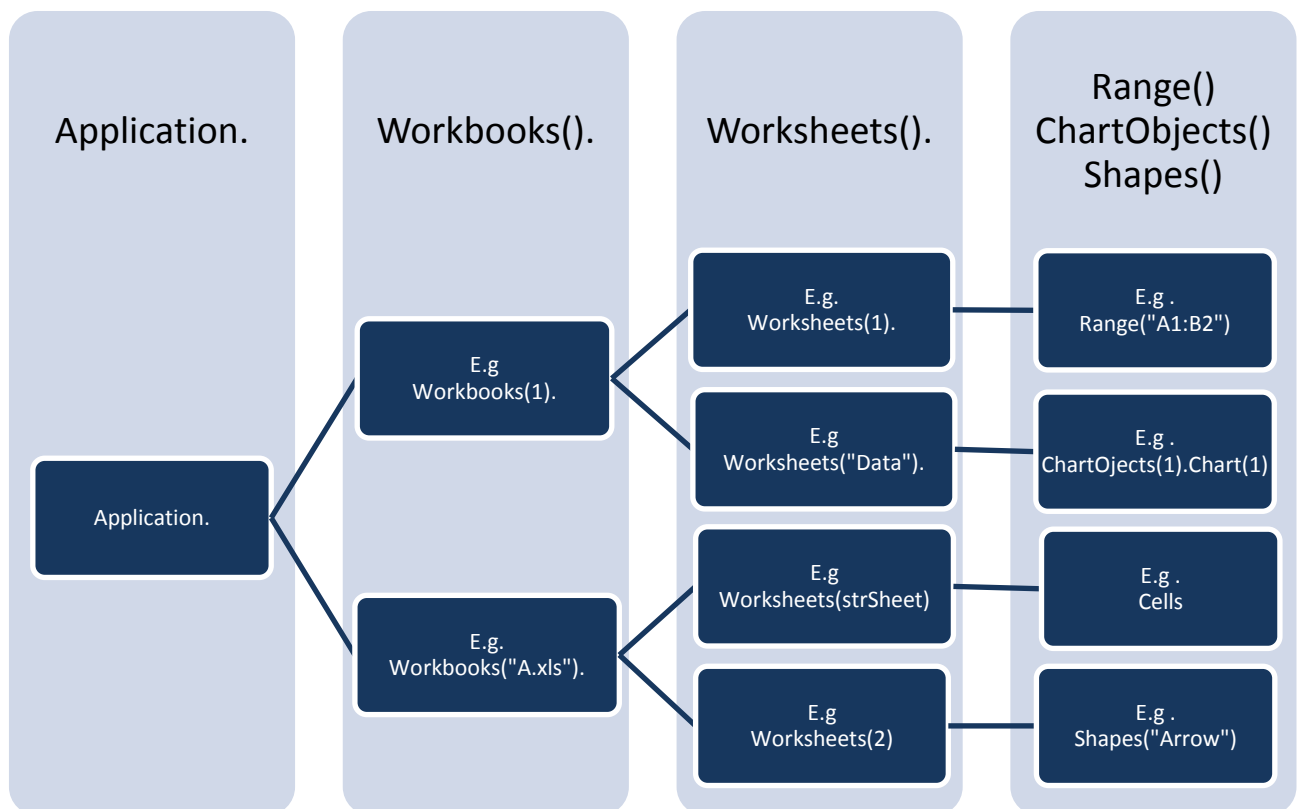
#### Shapes or Pictures

E.g. Add, hide, resize or delete pictures and shapes.

Simply remember to start at the top of the hierarchy and 'stop' at the appropriate place, e.g.

#### **Application.Workbooks(1).Worksheets(1).Calculate**

The above line of code stops at the Worksheet level and then invokes the 'Calculate' method. It therefore ONLY recalculates WorkSheet(1)



## Defining Workbook and Worksheet Objects

### ActiveWorkbook, ActiveSheet.

The 'Active' object is the one currently being viewed or worked upon in the main Excel window. The open workbook name (including its .xls extension) will be unique, as will be the sheet tab name within that workbook, and can be used to uniquely define the object.

#### Example;

**ActiveWorkbook. ActiveSheet.**

#### Good for ...

Everything!

#### Beware...

Nothing really, until you get to advanced programming (*when speed, and event handling issues can come into play*)

### Pro-Tip

If you use `ActiveWorkbook` and `ActiveSheet` in your code, it is good practice to enter the first line as ...

```
ThisWorkbook.Activate
```

This will prevent your code attempting to run on another workbook.

### ThisWorkbook

Referencing the workbook in this way ensures that the code is acting upon the workbook containing the VBA itself.

#### Example;

**ThisWorkbook** (There is no Worksheet equivalent)

#### Good for ...

Times when you think the user may open or activate another workbook and your code is not designed to work on other workbooks.

### Using its index

Multiple objects of the same type are referred to as a 'Collection'. For example a typical Excel Application may contain a collection of three Workbooks. Each workbook in the collection is assigned a unique index and can be referenced using this number.

#### Example;

**Workbooks (1) .Worksheets (variable\_number) .**

#### Good for ...

When cycling through all the open workbooks, the index can be used in a For-Next loop

#### Beware ...

The index may change; **Workbooks (2)** may become **Workbooks (1)** if the first workbook is closed.

### Using its name

The open workbook name (including its .xls extension) will be unique, as will be the sheet tab name within each workbook, and can be used to uniquely define the object.

**Example;**

**Workbooks ("Data.xls") . Worksheets (variable\_string) .**

**Good for ...**

Not much really ... It's OK if the workbook/worksheet name never changes. It's simple to understand but see 'Object Name' for a much better alternative. It is mentioned here really only because you are likely to see it used by less experienced programmers.

**Beware...**

The user may change the workbook or worksheet name, and invariably does!

### Use its OBJECT name

The BEST way to reference Worksheets (and most other object you will encounter such as Charts, Pivot Tables, Userforms and Shapes) is to set and then use an object name. *(not applicable to workbooks)*

**Good for ...**

Everything

**Speed-Tip**  
 Giving meaningful names to your code modules only takes a few seconds and significantly speeds up navigation around your project.

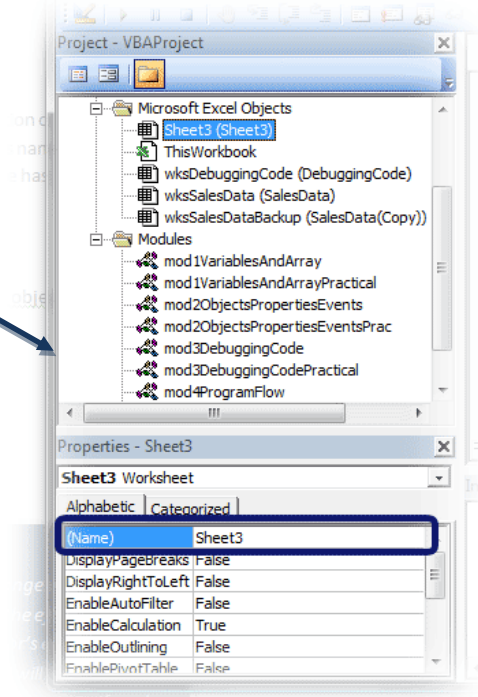
**Example;**

Set an appropriately descriptive name in the 'Name' section of the Properties Window. In this example both the object's name AND sheet tab name are still set to 'Sheet3'. The word must be alpha-numeric and contain no spaces. Once a name has been set it can be used directly in code, for example ...

**Msgbox wksSalesData.Name**

...shows the TAB name of the worksheet object.

**Pro-Tip**  
 Setting up appropriate object names and range names may appear time consuming and not worthy of the effort. However this is time well spent. It is the decorators' equivalent of sanding down properly before painting and will ALWAYS result in a better, more reliable and more extensible system. If you're going to cut corners DON'T do it here!



## Defining a Worksheet Range

### Pro-Tip

There are numerous ways to define a range object in VBA, many of which are listed below. Each have their own advantages and disadvantages and you will often have a choice of which to use. Experiment with different methods, however the following four methods (all described more fully below) are in our experience the most flexible and powerful in VBA. The first two can be used to describe the starting (or actual) position and the second two the range's relative position/size.

<code>Range ("Data")</code>	Use a previously defined Excel named range
<code>Cells (X, Y)</code>	Using variables is a powerful, generic way to determine a start position
<code>.CurrentRegion</code>	Setting a contiguous range around a known starting position
<code>.Offset.Resize</code>	Working on a relative position to a known cell and/or resizing the range

### Defining a single cell

<code>Cells (2, 3)</code>	The second row down the third column across (Cell C2)
<code>Range ("A2")</code>	A2
<code>ActiveCell</code>	The upper left most cell in the Selection (see below)
<code>Intersect (Rows (3) , Columns (2) )</code>	The intersection of Row 3 and Column 2 (Cell B3)

### Defining a range of cells

<code>Range ("A2 : B3")</code>	The contiguous range from the top left to the bottom right (i.e. A2, A3, B2, B3).
<code>Range ("Data")</code>	The range defined by the range name in Excel, surrounded by quotes (e.g. Data).
<code>Rows (2)</code>	The entire 2 <sup>nd</sup> row.
<code>Columns (1 : 3)</code>	All cells in Columns A, B & C.
<code>Selection</code>	One or more cells representing the selected cells on the active sheet.
<code>UsedRange</code>	The range starting at Cell A1 and ending at the intersection of the last populated row and the last populated column.
<code>CurrentRegion</code>	The contiguous range starting at any single named cell.
<code>Cells</code>	All the cells in the worksheet or the specified range (XL 2003 = A1:IV65536)

### Pro-Tip

It is entirely possible to define multiple (non contiguous) ranges as a single object – However this can get complex and is subject to limitations. It is usually better to enter a single range and use a loop to act upon each range in turn. See Excel help under UNION if you really need to use multiple ranges in your code.

**Offset & Resize**

Defining a range objects can often powerfully be combined with Offset and Resize ...

A cell reference X cells to the right and Y cells below the specified range (x and y can be positive or negative)

**.Offset (X, Y)**

A range of cells X rows high and Y columns wide (irrelevant of the previous size)

**.Resize (X, Y)**

**Cells (1, 1).Offset(1, 0)**

A2

**Range ("B2").Resize(1, 3)**

Cells B2, C2 & D2

**ActiveCell.Offset(-1, 0).Resize(2, 1)**

The active cell and the cell above it

**Pro-Tip; Range Names & VBA**

In the workshop you will have been shown many ways to specify and manipulate worksheet ranges ... it is a fundamental VBA skill.

However if there is ANY chance the worksheet structure will change, such as the inserting or deleting of rows or columns you should only EVER specify or manipulate cells with reference to (or an 'Offset' from) a named range, by using DYNAMIC (self-calculating) formula.

E.g. The following lines of code both delete the last row in the range 'Data'

```
Range ("Data").Offset (Range ("Data").CurrentRegion.Rows.Count-1, 0).EntireRow.Delete
Range ("A5").EntireRow.Delete
```

(Don't worry if right now that first line looks complicated ... you'll understand it soon enough)

Unlike Excel, VBA will NOT amend its references to adapt for a new or deleted rows or columns. So for example if you (or someone else) EVER needed to amend the worksheet structure by (say) inserting a row, the second line of code would immediately be wrong. Worse, you may have to review every single line of code to find the problem. If you only take away one 'Tip' from this workshop, please make sure it's this one ...

**RANGE NAMES ARE THE MOST RELIABLE, CONSISTANT AND EXTENSIBLE WAY TO  
MANIPULATE WORKSHEET STRUCTURE AND WORKSHEET DATA IN EXCEL**

## Implicit and Explicit declaration of objects

Until now we have EXPLICITLY declared our objects, starting at the top (Application) level. However, by default VBA will implicitly 'assume' that you are referring to the ACTIVE object of the parent hierarchy. For example if the user is currently in Worksheet( 2) of Workbook( 1) then ...

**MsgBox Range ("A1")**

will be interpreted by Excel exactly as if it was written ...

**MsgBox Application.ActiveWorkbook.ActiveSheet.Range ("A1") .Value**

or (if for example the user is currently in Worksheet(2) of Workbook(1)) ...

**MsgBox Application.Workbooks (1) .Worksheets (2) .Range ("A1") .Value**

It therefore follows that when declaring any object EXPLICITLY it does NOT need to be active (selected), but when you omit the hierarchy then the next level up MUST be the active (or selected) object.

### Pro-Tip

The base object 'Application' is only explicitly required when controlling Excel from another software product (e.g. Access or Word) or for Application level events (such as Application.Calculation). It is perfectly safe and acceptable to drop the syntax 'Application.' from your object definitions if you intend to run everything from within Excel.

Use this feature to your advantage.... Mix Implicit & Explicit declarations

The following line of code copies cell A1 from the active sheet to A1 on WorkSheet(1) ...

**Worksheets (1) .Range ("A1") .Value = Range ("A1") .Value**

The following line of code copies the value from the active cell in another workbook into the corresponding cell on Worksheet(1) of the workbook containing the code ...

**ThisWorkbook.Worksheets (1) .Range (ActiveCell.Address) .Value = ActiveCell.Value**

## Method & Property Examples

Once you have defined your object, you can define what you want to do with it (Method) or what you want to know or set on it (Property).

Common Methods	Common Properties
.Add	.Text
.Clear	.Format
.Copy	.Name
.Paste	.Height
.Select	.Value
.Sort	.Address
	.Index

The following pages list typical samples of the syntax of some worksheet, range and workbook objects to assist you with the style of syntax used.

### Speed-Tip

Methods and Properties vary according to the object, and its status. Use Intellisense to help you quickly define objects, properties, and events permissible for any specific situation.

## Worksheets

Creates a new worksheet as the last tab in the active workbook, and sets it as the active sheet.

**Sheets.Add**

Makes a copy of the active worksheet and places it in a new workbook

**ActiveSheet.Copy**

Makes a copy of the worksheet named "Sheet1" and places it as the last worksheet

**Sheets("Sheet1").Copy After:=Sheets(Sheets.Count)**

Deletes sheet index 1 (requires confirmation by the user)

**Sheets(1).Delete**

Deletes sheet index 1 (doesn't require confirmation by the user)

```
Application.DisplayAlerts = False  
Sheets(1).Delete  
Application.DisplayAlerts = True
```

Deletes any chart on the active sheet

```
ActiveSheet.ChartObjects.Delete
```

Activates the last worksheet

```
Sheets(Sheets.Count).Activate
```

Clears all the formats and contents of worksheet index 2

```
Sheets(2).Cells.Clear
```

Clears all the contents but leaves the formatting of worksheet index 2

```
Sheets(2).Cells.ClearContents
```

In manual calculation mode, (re)calculates the active sheet, leaving the rest of the workbook un-calculated

```
ActiveSheet.Calculate
```

Hides the active sheet (the 2nd line unhides it)

```
Sheets(1).Visible = xlSheetHidden  
Sheets(1).Visible = xlSheetVisible
```

Hides the active sheet so it can only be un-hidden using VBA (The sheet name doesn't show up in the list of hidden sheets you can un-hide)

```
ActiveSheet.Visible = xlSheetVeryHidden
```

Protects the active sheet with the password 123 (the following line unprotects it again)

```
ActiveSheet.Protect DrawingObjects:=True, Contents:=True, Scenarios:=True, Password:="123"  
ActiveSheet.Unprotect Password:="123"
```

Selects all the charts and shapes on sheet index 1

```
Sheets(1).Shapes.SelectAll
```

How many chart objects are in the active sheet?

```
Debug.Print ActiveSheet.ChartObjects.Count
```

What is the last used row on the active sheet?

```
Debug.Print ActiveSheet.UsedRange.Rows.Count
```

What is the name and index of the active sheet?

```
Debug.Print ActiveSheet.Name, ActiveSheet.Index
```

Set the column width for every column to 5

```
Sheets(1).Cells.ColumnWidth = 5
```

## Ranges

Copies B2 to C3 on the active worksheet

```
Cells(2, 2).Copy Cells(3, 3)
```

Copies the contiguous range around cell A1 of the active sheet to A1 on Sheet index 2

```
Range("A1").CurrentRegion.Copy Sheets(2).Cells(1, 1)
```

Clears the formatting of column B

```
Range("B1").EntireColumn.ClearFormats
```

Clears the contents of column C, except (for example a header) in Row 1

```
Cells(2, 3).Resize(ActiveSheet.UsedRange.Rows.Count, 1).ClearContents
```

Adds a comment in cell A1 representing the number of cells in the contiguous range around A1 (Only works when there is no comment already in A1):

The second line deletes the comment, value & formatting

```
Cells(1, 1).AddComment Cells(1, 1).CurrentRegion.Cells.Count & " contiguous cells"
```

```
Range("A1").Clear
```

Adds a new range name (or replaces the existing range) entitled 'NewRange' at cell B3 of the active sheet.

```
Names.Add Name:="NewRange", RefersTo:=Cells(3, 2)
```

Adds a new range name 3 rows down from 'NewRange'

```
Set rngOffset = Range("NewRange").Offset(3, 0)
```

```
Names.Add Name:="NewOffsetRange", RefersTo:=rngOffset
```

Deletes the first column (also see comments below)

```
Cells(1, 1).EntireColumn.Delete
```

When you delete an entire row or column, remember Excel will shift the remainder of the cells up or to the left, usually resulting in a re-calculation of the worksheet. If deleting lots of rows/columns, this can slow things down dramatically, so it is best to start from the bottom right and/or control the way Excel re-calculates the sheet.

This example does both ...deleting any odd rows between 1 and 100 on the active sheet

```
Application.Calculation = xlCalculationManual
```

```
For lngRow = 100 To 1 Step -1
```

```
    If lngRow Mod 2 = 0 Then
```

```
        Cells(lngRow, 1).EntireRow.Delete
```

```
    End If
```

```
Next
```

```
Application.Calculation = xlCalculationAutomatic
```

The address of the last used cell

```
Debug.Print Cells(ActiveSheet.UsedRange.Rows.Count, ActiveSheet.UsedRange.Columns.Count).Address
```

The address of the cells 1 column to right of the contiguous region around cell A1

```
Debug.Print Cells(1, 1).CurrentRegion.Offset(0, 1).Address
```

Enters a Sum() formula in the active cell that sums the 5 cells below it

```
ActiveCell.Formula = "=sum(" & ActiveCell.Offset(1, 0).Resize(5, 1).Address & ")"
```

Sets row 2 to cell formatting of 2 decimal places

```
Range("A2").EntireRow.NumberFormat = "0.00"
```

Sets the used range to green

Note: Colours & colour indexes are best obtained by using the macro recorder

```
ActiveSheet.UsedRange.Interior.ColorIndex = 4
```

## Workbooks

Create a new workbook and set it as the active workbook.

```
Workbooks.Add
```

However, without saving this new workbook with a known name, it can be complex to refer back to it as an object later. The following 3 lines demonstrate a method to add a new workbook, and then switch between the calling workbook and new workbook.

```
Set wkb = Workbooks.Add
ThisWorkbook.Activate
wkb.Activate
```

Activates the workbook that contain this code

```
ThisWorkbook.Activate
```

Gives the complete path of the active workbook

```
Debug.Print ActiveWorkbook.Path
```

Tells us if active workbook has been saved after any changes

```
Debug.Print ActiveWorkbook.Saved
```

### A NOTE ABOUT DELETING WORKBOOKS

Deleting saved workbooks is fraught with danger, and it is STRONGLY advised against. Whilst VBA can manipulate / rename / delete files, it is not well suited to this type of functionality. It is usually acceptable to simply close the workbook without saving...

```
ActiveWorkbook.Close savechanges:=False
```

Note: If the workbook has not yet been saved with a file name, this will also effectively delete it.

# Variables, Arrays & Constants

## Variables, Constants & Arrays – A One Page Summary

A variable is a way of storing a value for future use

### Variables

- Variables are named by the user and can be any alpha-numeric string starting with a letter.
- Giving Variables meaningful and consistent names makes it far easier to follow your code.
- Always start your variable with the 3 characters representing the data type (e.g. **str**, **obj**)
- Spaces aren't allowed in variable names, so keep code easy to read by using an underscore (**str\_user\_name**) or mixed case (e.g. **strUserName**) variable names.
- Use the keyword **Dim** to dimension a variable ready for use at the beginning of a subroutine or function.
- Variables are NOT objects but can contain object(s)

### Constants

- A Constant is a special type of variable that is useful for numbers that aren't likely to change such as a column number or number of list items.
- A Predefined Constant is an Excel defined figure which can be useful when setting system or object parameters e.g. **Application.Calculation = xlManual**. Intellisense is useful for showing these predefined constants when they are applicable.

### Arrays

- An Array is a variable with multiple compartments, each holding one variable or item of data.
- Any variable can be defined as a variable sized array by appending `()` or a fixed size array by appending `(x)` where x is the array size. Prefix the name with an 'a' so you know it's an array.
- Use **Redim()** to (re)size a variable sized array before or whilst using it. Use the Preserve keyword, **Redim Preserve()** to prevent the current array contents being lost.
- Arrays can have multiple dimensions (e.g. **ReDim astrName(2,3)** will have 2 dimensions... you can re-dimension these arrays as normal, but only the last dimension can be resized).

### Advanced

- Always declare the data TYPE of variable or constant, for example String, Boolean or Variant.
- Wherever possible use variables, constants or range names in your code, instead of direct numbers and cell references (e.g. the range **Taxate** instead of **Range(C7).Value**)
- Only use a Variant data type if you cannot be sure what data type will be required.
- Range names can be used to good effect as variables or constants within VBA.
- Some programmers use module or public (all module) level variables. However, it is advisable to pass local variables as arguments where feasible – Keep the scope as tight as possible.
- The keyword **'Set'** is required when setting the value of any object variable (e.g. range, chart).

## Standard Variables

### String

Strings are used to hold text variables and may contain up to around 2 billion characters.

```
Dim strText As String
```

### Long

Longs are integers whose maximum is about +/-2 billion.

```
Dim lngNumber As Long
```

### Double

Doubles are any floating point number.

```
Dim dblNumber As Double
```

### Date

Dates and times are contained within the date serial number. Whole numbers represent a day and the decimal represents the decimal part of a whole day (e.g. 0.75 = 6pm).

```
Dim datDate As Date
```

### Boolean

TRUE or FALSE

```
Dim blnTrueFalse As Boolean
```

### Variant

Variants can be used to hold any standard variable.

It should ONLY be used for inputs and error handling when you do not know the input type.

```
Dim varInput As Variant
```

There are other standard data types including Integer, Single, Currency, Byte & Decimal.

## Object Variables

Objects are used throughout VBA and are key to its powerful use. The standard object variable describes an ANY entire object such as workbook, worksheet, range or graph.

Do NOT use the generic object type if you know the object type with which you are working.

```
Dim objAnyObject As Object
```

Always use the specific object if possible, for example ...

<b>Dim wkbBook As Workbook</b>	A workbook
<b>Dim wksSheet As Worksheet</b>	A worksheet
<b>Dim rngCells As Range</b>	A worksheet range of any size (contiguous or non-contiguous)
<b>Dim chtChart As Chart</b>	A chart
<b>Dim wndWindow As Window</b>	An Application window

## Why bother pre-defining Variables?

Why not set all variables simply as object or variant?

These two valid questions are commonly asked by new programmers. After all you don't have to declare variables before you use them and Excel will automatically work with different data types, even converting them automatically wherever possible. ... so why bother?

Let us start by saying that the world won't cave in before your eyes if you don't declare your variables, but also that we have never met a full time, experienced & competent programmer that doesn't do it, even those who started out on the 'Why bother?' side of the fence. Here are the main reasons.

- **It simplifies and in some cases negates the need for error handling.**

Imagine a function that adds VAT to an input. You declared your input as a variant and in your testing you tried it with lots of different numbers and it worked perfectly. When you release your system a user accidentally enters a letter O instead of a number zero - Your function crashes. If you are in the habit of explicitly declaring and passing the input as a number (with error handling) the problem would have been averted.

- **It avoids problems with misspelled variable names**

At the start of your routine, the variable LastColumnPosition is set to the number 2. Further down your code the LastColumnPosition is used to determine the next column position .... Except LastColumnPosition is actually still zero and it takes you an hour to find out why your code doesn't work properly. How quickly did you spot the error?

- **It prevents unexpected 'features' of automatic variable handling**

Your number variables entitled 'UserValue1' and 'UserValue2' are set to "10" and "2" respectively. Does the following surprise you?

```
UserValue1 * UserValue2 = 20
UserValue1 + UserValue2 = 102
```

- **It allows Intellisense to help you**

Intellisense is a superb time-saver and sense checker. Dimensioning object variables as specific objects (such as chart or range) allows Intellisense to let you know which properties & methods are applicable (along with their often 'American' syntax) in any given situation.

- **It's faster, smaller and more efficient for the computer**

Of less importance these days, but nonetheless worthy of note is the speed and size advantages of using specific variables. Explicit variables take up less memory, and Excel does not have to figure out what data type they are before using them. This can make a difference in speed of up to 3x in your code.

## Setting & Using Variables: Examples

### String

```
strText = "Hello" & Chr(10) & "How are you?"
MsgBox strText
```



```
strText = "2"
MsgBox strText & strText
```



### Long

```
lngNumber = 1000
```

Setting Long numbers gives some (perhaps) unexpected results...

```
lngNumber = 0.3      Yields 0
lngNumber = -0.3     Yields 0
lngNumber = 0.5      Yields 0
lngNumber = -0.5     Yields 0
lngNumber = 0.7      Yields 1
lngNumber = -0.7     Yields -1
```

### Double

Excel displays accuracy up to 15 decimal places

```
dblNumber = 1000
MsgBox dblNumber / 3
```



Excel happily accepts and displays scientific notation

```
dblNumber = 1000 ^ 6
MsgBox dblNumber
```



### Boolean

```
blnTrueFalse = False
MsgBox Not (blnTrueFalse)
```

True has a mathematical value of -1 and False has a mathematical value of 0, so it is also possible to use Boolean values in equations.



## Date

Take great care whilst working with dates, especially if you work in an international organisation. For absolute safety it is advised to keep the date as a serial number throughout your code and worksheet calculations, then simply formatting any cells that need to be viewed by a user.

**datDate = 36526**      Yields 31st December 1999

**datDate = "1/3/2000"**      Yields, EITHER 1st March 2000 or 3rd January 2000 depending on your international date settings.

The Date type is fairly flexible when it comes to its construction. The following example would be understood perfectly well by VBA. As well as using text in its construct, also note the use of different separators used for the month & year.

**strText = "2"**

**datDate = strText & strText & "/" & strText & "-" & strText & "007"**

**MsgBox datDate**



Using the built in Now() function can be useful in messages, calculations or entered in cells as part of an audit trail feature.

**datDate = Now()**

**MsgBox Format(datDate, "d mmm yyyy, h:mm am/pm")**



### Pro-Tip

Date handling is the cause of problematic issues in many Excel systems, so test your date routines thoroughly. Also check out these two very useful functions...

**DateAdd()**      Adds (or subtracts) whole days, months or years from a date

**DateDiff()**      Returns the time interval between two dates

## Variant

A variant can be used to hold any standard variable but cannot hold object variables.

```
varInput = 5
varInput = "5"
varInput = True
varInput = Now() + 1
```

## Objects

All object variables require the SET word in order to be instantiated.

```
Set objAnyObject = ThisWorkbook
```

```
Set wkbBook = ActiveWorkbook
```

```
Set wkbBook = Workbooks(1)
```

```
MsgBox wkbBook.Worksheets.Count [ [The number of worksheets in the first workbook]]
```

```
Set wksSheet = ActiveSheet [ [The active worksheet]]
```

```
Set wksSheet = Sheets("SalesData")
```

```
MsgBox wksSheet.UsedRange.Address
```



```
Set rngCells = Range("A1:B3") [ [The range of cells A1:B3 on the active sheet]]
```

```
Set rngCells = Cells(1, 1).Resize(2, 3)
```

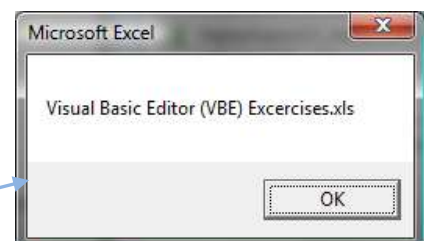
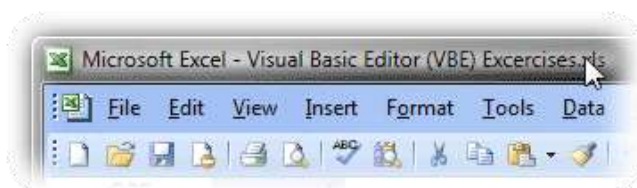
```
MsgBox rngCells.Address
```



```
Set rngCells = Selection [[The selected cells on the active worksheet]]
```

```
Set wndWindow = ActiveWindow
```

```
MsgBox wndWindow.Caption
```



## Arrays

An array can be thought of as multiple variables, all of the same type and with the same name. We determine the number of variables in the array, either by declaring the number at the same time we declare the variable (static) or at run time (dynamic). We can even re-size dynamic arrays as we go along.

We use the prefix 'a' as a convention to indicate to ourselves that the variable is an array and the suffix () is used to tell VBA that the variable is an array.

### Declaring Arrays

#### Declaring a fixed size array

```
Dim aLngOptions(3) As Long
```

When an array is dimensioned like this VBA actually starts the array size at 0 and finishes at the number specified, so in the above example there will be 4 variables available.

```
aLngOptions (0)
aLngOptions (1)
aLngOptions (2)
aLngOptions (3)
```

However for the sake of simplicity it is perfectly acceptable to ignore the first (0) variable if you wish.

#### Declaring a fixed size (multi-dimension)

```
Dim avarCellValues(2, 2) As Variant
```

#### Declaring a variable size array (single or multi-dimension)

(The actual size of the array is set at run time)

```
Dim avarData() As Variant
Dim avarSizeTest() As Variant
```

### Populating Arrays

#### Fixed size array

```
aLngOptions(1) = 1
aLngOptions(2) = 10
aLngOptions(3) = 100
MsgBox aLngOptions(2)          [[10]]
```

#### Fixed size (multi-dimension)

```
avarCellValues(1, 1) = Cells(1, 1).Value
avarCellValues(2, 1) = Cells(2, 1).Value
avarCellValues(1, 2) = Cells(1, 5).Value
avarCellValues(2, 2) = Cells(2, 5).Value
MsgBox avarCellValues(2, 1) & ", " & avarCellValues(2, 2)
[[The value of cells A2 & E2 on the Sales Data sheet]]
```

**Variable size (single or multi-dimension)**

Requires REDIM before use

```
ReDim avarData (2, 2)
avarData (1, 1) = Cells (1, 1) .Value
avarData (2, 1) = Cells (2, 1) .Value
avarData (1, 2) = Cells (1, 3) .Value
avarData (2, 2) = Cells (2, 3) .Value
MsgBox avarData (2, 1)
[[The value of cells A2 on the active worksheet]]
```

**Resizing a variable sized array**

ONLY the last dimension of an can be resized ....use the PRESERVE word to keep current contents  
The following line will re-dimension that second dimension of avarCellValues()

The following example resizes the second (last) dimension of 'avarData' from 2 to 3 whilst keeping the previously populated values.

```
ReDim Preserve avarData (2, 3)
avarData (1, 3) = Cells (1, 4) .Value
avarData (2, 3) = Cells (2, 4) .Value
MsgBox avarData (2, 1) & ", " & avarData (2, 3)
[[The value of cells A2 & E2 on the Sales Data sheet]]
```

```
ReDim avarData (2, 3)
[[Without Preserve' the data will be lost, the answer below will be null]]
MsgBox avarData (2, 1)
```

**Obtaining the size of an array**

LBound() :The lowest index usable in the array

UBound() :The highest index usable in the array

```
ReDim avarSizeTest (2)
Debug.Print LBound(avarSizeTest)           [[0]]
```

This example makes the array 1 item bigger

```
ReDim Preserve avarSizeTest(UBound(avarSizeTest) + 1)
Debug.Print UBound(avarSizeTest)          [[3]]
```

**Pro-Tip**

Always use 'Option Explicit' at the start of every code module. It forces you to declare your variables and prevents the agony of wasted hours looking for code errors that simply aren't there, when the problem is caused solely by a mis-spelled variable.

## Control Structures & Program Flow

So far all the routines we have looked at have started at the beginning and run one line at a time until they get to the end. Whilst this is simple, it is far from efficient as we would need a different routine for every possible scenario and variation.

A far more efficient method would be one which could handle various scenarios according to their status or automatically repeat the instructions for each applicable variable or object.

Listed on the next few pages, Excel gives us many different ways in which to achieve this functionality.

### Pro-Tip

You may notice that when in many of the cases below the code is indented with tabs when the program flow is being changed, and different sections of the subroutine are separated with line breaks and 'white space'.

This is not a requirement of VBA but helps us as programmers follow the flow of the code through the subroutine. As your code becomes more complex (or if you are following 'someone else's code) you will realise the difference in readability this simple procedure makes.

## IF (ElseIf, Else)

IF is one the most basic and widely used branching instructions.

```
If (condition) Then
    (do something)
End if
```

```
If Cells(1, 1).Value <> "Date" Then
    MsgBox "It appears the worksheet structure has changed."
End If
```

```
If (condition) Then
    (do something)
Elseif (condition) Then
    (do something)
End if
```

```
If Cells(1, 1).Value <> "Date" Then
    MsgBox "It appears the worksheet structure has changed."
ElseIf Cells(2, 1).Value = "" Then
    MsgBox "It appears the sales sheet contains no data"
End If
```

```
If (condition) Then
    (do something)
Elseif (condition) Then
    (do something)
Else
    (do something)
End if
```

```
If Cells(1, 1).Value <> "Date" Then
    MsgBox "It appears the worksheet structure has changed."
ElseIf Cells(2, 1).Value = "" Then
    MsgBox "It appears the sales sheet contains no data"
Else
    MsgBox "Data found"
End If
```

## Select Case()

```
Select Case (expression)
Case (expression)
    (do something)
Case (expression)
    (do something)
End Select
```

```
Select Case Range("F2").Value
Case "Smith"
    MsgBox "The first order was for Smiths, the corner shop"
Case "Clarke"
    MsgBox "The first order was for Mrs Clarke, the market trader"
End Select
```

Select Case does not check further cases once it has found a condition that is true

```
Select Case Range("J2").Value
Case Is > 15
    MsgBox "The first order was a large order"
    [[the following 2 select case statements are also true, but not executed]]
Case Is > 10
    MsgBox "The first order was a medium sized order"
    [[the following select case statements is also true, but not executed]]
Case Is > 0
    MsgBox "The first order was a small order"
Case Is < 0
    MsgBox "The first order was a REFUND!"
End Select
```

Select case can look at more than one expression simultaneously

```
Select Case (expression)
Case (expression or expressions)
    (do something)
Case (expression or expressions)
    (do something)
Case Else
    (do something)
End Select
```

```
Select Case Range("D2").Value
Case "Andy", "Dave"
    MsgBox "The first order was taken by a man"
Case "Sarah", "Emily"
    MsgBox "The first order was taken by a lady"
Case Else
    MsgBox "The first order was taken by " & Range("D2").Value
End Select
```

## For Next Loop

```
For (counter) = (start) To (finish)
    (do something)
Next (counter)
```

```
[[This example will show 3 message boxes in turn]]
For lngRow = 1 To 3
    MsgBox "Order " & lngRow & " was for " & Cells(lngRow + 1, 10).Value
Next
```

When deleting rows of data it is often easier to start at the bottom. This particular example counts backwards. (the For Next loop can use any positive or negative integer).

```
For (counter) = (start) To (finish) step (integer)
    (do something)
Next (counter)
```

```
lngRows = ActiveSheet.UsedRange.Rows.Count
[[The total number of rows used in the worksheet]]
For lngRow = lngRows To 2 Step -1
    Debug.Print "Reviewing row " & lngRowNote
    [[the row in the immediate window]]
Next
```

For next loops are often useful when running through ranges of data. This example displays the addresses of all the cells in column A of the contiguous range around cell A1.

```
[[Set the range as the contiguous region starting at A1]]
Set rngData = Sheets(1).Cells(1, 1).CurrentRegion
lngRows = rngData.Rows.Count [[count the rows]]

For lngRow = 1 To lngRows
    Debug.Print rngData.Cells(lngRow, 1).Address
Next
```

## For Each Loop()

The For Each Loop is used to cycle through each and every object in a collection of objects. It is useful because you don't have to count the number of objects first.

```
For Each (object) In (collection of objects)
    (do something)
Next
```

This example gives the cell address of each cell in the current selection.

```
For Each rngCell In Selection
    Debug.Print rngCell.Address
Next
```

This example gives the cell address of each cell in the contiguous range starting at cell A1.

```
For Each rngCell In Cells(1, 1).CurrentRegion
    Debug.Print rngCell.Address
Next
```

This example gives the name of every worksheet in the calling workbook.

```
For Each shtWorksheet In ThisWorkbook.Worksheets
    Debug.Print shtWorksheet.Name
Next
```

This example gives the number of worksheets in every open workbook.

```
For Each wkbBook In Application.Workbooks
    Debug.Print wkbBook.Name & ", " & wkbBook.Worksheets.Count & " sheets."
Next
```

## Do Until Loop and Do While Loop

As their names suggest these loops carry on (indefinitely) while or until a condition is met.

```
Do Until (Condition)
    (do something)
Loop
```

This example moves down the 2<sup>nd</sup> column until it finds a cell containing the word 'Pear'

```
Do Until Range("B1").Offset(lngRow, 0).Value = "Pear"
    lngRow = lngRow + 1
Loop
MsgBox "The first order for pears is in row " & lngRow
```

```
Do
    (do something)
Loop Until (Condition)
```

This example keeps presenting an input box for the user until they enter 6 characters or more.

```
Do
    strPassword = InputBox("Please enter 6 or more characters")
Loop Until Len(strPassword) >= 6
```

If you prefer, the syntax **WHILE** in place of **UNTIL** may also be used.

E.g. the two lines below would work in a similar way ...

```
Do While x <= 10
Do Until x > 10
```

## With, End With

This instruction allows you to perform multiple instructions on a single object whilst referring to it only once. Please note the use of the dot.

```
With (object)
    .(do something)
End With
```

Take care when using With / End With

The use of the dot (.) can be used to your advantage or create unexpected errors. The following example may or may not display the same value, depending on which is the active sheet.

```
With Worksheets("SalesData")
    Debug.Print .Cells(1, 1).Value
    [[A1 on the worksheet 'SalesData']]
    Debug.Print Cells(1, 1).Value
    [[A1 on the active worksheet]]
End With
```

## Nesting Code

The power of branching code becomes apparent when you start combining these features. ... this is also when you really appreciate the indenting of code!

```
If Cells(1, 1).Value <> "Date" Then
    MsgBox "It appears the worksheet structure has changed."
ElseIf Cells(2, 1).Value = "" Then
    MsgBox "It appears the sales sheet contains no data"
Else
    Do
        lngRow = lngRow + 1
        Select Case Cells(lngRow, 2).Value
            Case "Apple"
                lngApples = lngApples + Cells(lngRow, 7).Value
            Case "Pear"
                lngPears = lngPears + Cells(lngRow, 7).Value
        End Select
    Loop Until Cells(lngRow, 2).Value = ""
    MsgBox "We sold " & lngApples & " apples, and " & lngPears & " pears."
End If
```

## Scope & Code Location

Scope describes which parts of Excel and VBA can 'see' your variables, constants, objects, functions and routines.

In VBA variable and constant scope can be defined at 3 levels ...

- Subroutine (or Function)
- Module
- Workbook

Here is an example of subroutine level variable scope ...

```
Private Sub SetMessage_1()  
    Dim strMsg As String  
    strMsg = "Hello"
```

```
    [[Call the second test subroutine, which also defines and sets strMsg to 'Goodbye']]  
    SetMessage_2
```

```
    [[However, when we ask to 'see' strMsg it is still 'Hello']]
```

```
    MsgBox strMsg
```

```
    [[That is because the scope of strMsg is limited to the routine in which is it was dimensioned]]
```

```
End Sub
```

```
Private Sub SetMessage_2()  
    Dim strMsg As String  
    strMsg = "Goodbye"
```

```
End Sub
```

Programming best practice is to keep scope as tight (small) as possible. That is to say where 'possible keep scope to the routine or function.

Occasionally it may be useful to set constants (and even less occasionally variables) at module level. This means that *any* routine or function written in that module has access to that variable. A module level variable is set in exactly the same way, but is defined OUTSIDE the routine, placed at the top of the code module. We also use the prefix 'm' to remind us it is a module level variable or constant.

### Pro-Tip

Use module and public level variables and constants sparingly. Variables required in other functions or routines should usually be passed to the code as an argument. There is usually little genuine need for many of them outside the title of the project or an item you expect to change in the future (e.g. the column number of imported database)

Defining a Module Level Variable and constants at the top of a code module ...

```
Dim mstrUserAnswer As String
Const NUM_USERS As Long = 5
```

The example below sets the module level variable in one routine and displays it in the second.

```
Private Sub ModuleVariableText1 ()
    [[Set the module level variable]]
    mstrUserAnswer = InputBox("Enter user name")
    [[Call the second test subroutine]]
    ModuleVariableText2
End Sub

Private Sub ModuleVariableText2 ()
    MsgBox mstrUserAnswer
End Sub
```

Public variables work in the same way as module level variables but it is available to any routine or function in the workbook modules or userforms. They are also defined at the top of the module, with the word 'Public' and to help remind us we use the 'p' prefix.

```
Public Const pPROJ_NAME As String = "My Project"
Public pstrErrors As String
```

#### Scope also applies to routines and functions

In VBA, variable and constant scope for routines and functions are defined at 2 levels ...

- Module
- Workbook

The keywords **Private** and **Public** are used to designate whether or not code can be 'seen' outside the code module in which it resides.

#### Pro-Tip

It is considered good practice to use 'Private' functions and routines where possible.

This subroutine example could only be called from within the module in which it resides.

```
Private Sub ScopeTest1 ()
    MsgBox "Hello"
End Sub
```

Using Public, this subroutine could be called from any module in the workbook.

```
Public Sub ScopeTest2 ()
    MsgBox "Hello"
End Sub
```

# Functions, Routines & Arguments

## Definitions

### Routines (aka subroutines or subs or procedures)

A routine is a set of instructions that perform a task.

### Functions

A function is set of instructions that return a single value. They are similar to Excel worksheet functions.

### Arguments (aka parameters)

An argument is the data that is sent to a function or routine in order for it carry out its instructions.

Use a routine to carry out your instructions. Whilst achieving this, the routine may need to call one or more functions to calculate specific results.

The use of routines and functions is best described by way of an example ...

### Simple Routine & Function Example

This example routine calls the function SimpleFunctionTimeTomorrow, which in turn calculates the date and time 1 day in the future from now. It passes the answer back to the calling routine, which displays the date. Note that the function has a data type defined (in this case the date) which must co-inside with the variable that has been set up to hold the answer.

```

Private Sub Simple Routine()
    Dim datTomorrow As Date

    [[Call the function here]]
    datTomorrow = SimpleFunctionTimeTomorrow

    MsgBox datTomorrow
End Sub

Private Function SimpleFunctionTimeTomorrow() As Date

    [[Tomorrow is the date today + 1]]
    SimpleFunctionDateTomorrow = Now() + 1

End Function

```

## Passing Parameters to a Function

Sometimes you need to pass arguments to the function to enable it to work.

Arguments can be variables or constants (of the right data type) or fixed data such as a number or string. Note that the parameters are passed in brackets, and that the parameters also have their data type defined.

This subroutine example uses a function that is passed a workbook path value to get the drive letter of the workbook. One of the key advantages of using a function is that it is re-usable. Without any changes this function could just have easily be called by another routine that wanted to know the path letter of the active workbook or even the Excel application itself.

```
Private Sub SimpleArgmeuntRoutine ()
    Dim strPath As String
    Dim strDrive As String

    strPath = ThisWorkbook.Path
    strDrive = GetDrive (strPath)
    MsgBox "This workbook is on drive " & strDrive
End Sub

Private Function GetDrive(strPath As String) As String
    [[The path is the first letter of the path]]
    SimpleArgumentFunctionDrive = Left(strPath, 1)
End Function
```

## Multiple and Optional Parameters

You can pass multiple arguments and/or make some of them optional. In this example the function is called twice, with differing results...

```
Private Sub CallFunction_1 ()
    Dim datNextDate As Date

    [[Call the function, and pass two arguments]]
    datNextDate = FutureTime (Now(), 36)
    MsgBox datNextDate

    [[This time call the function, but ignore the optional parameter]]
    datNextDate = FutureTime ("24/12/2004")
    MsgBox datNextDate

End Sub

Private Function FutureTime(datStartDt As Date, Optional lngHrs As Long = 24) As Date
    [[If the optional parameter was NOT passed, it takes on the default value assigned to it in the
    parameter list]]
    GetDateAndTimeInFuture = datStartDate + lngHours / 24
End Function
```

## Passing Parameters back and forth between Routines

You can also pass arguments between routines. The called routine can use the value, or set or change the value if it is a variable. Unlike a function, a routine does not HAVE to pass a value back to the calling sub.

### Pro-Tip

Functions & Routines should be separated logically, with each one performing a single aspect of the requirement. As a guide, when printed, functions should take up no more than 1 sheet, and routines no more than 2 sheets of paper.

If this is the case you should consider fragmenting your code into one core routine, calling primary and secondary routines and/or functions. This will likely create the need for lots of parameters, but don't worry .... this is one of the signs of a competent, professional programmer.

This example uses a main sub to call a second sub. If the parameter meets a certain condition then the second sub creates a new worksheet and changes the parameter. The value of the parameter is then checked in the calling sub to decide what message to display.

```
Private Sub CallingSub()
    Dim strCreate As String

    strCreate = "Sheet"
    [[Call the routine and pass the argument]]
    CreateItem strCreate

    If strCreate <> "" Then
        MsgBox strCreate & " has been added"
    Else
        MsgBox "Sorry, nothing has been added"
    End If
End Sub
```

```
Private Sub CreateItem(strCreate As String)
    [[Check the value of the argument and take an action]]
    If strCreate = "Sheet" Then
        [[Add a worksheet]]
        Worksheets.Add
        [[Amend the value of the argument]]
        strCreate = "Worksheet: " & ActiveSheet.Name
    Else
        strCreate = ""
    End If
End Sub
```

## Error Handling

VBA offers us several ways in which we can handle errors through 'On Error'

The most applicable, common and useful are ...

- On Error Goto** (label)      VBA goes straight to the label
- On Error Resume Next**    VBA ignores the error
- On Error GoTo 0**            Resets VBA to any previous error handling

If your system is used by non-programmers, you should always use a minimum of a generic error handler at the point of code entry (i.e. the first subroutine called)

- VBA can tell you or the user about any error via the **Err** object, which is created automatically by VBA if an error occurs.
- VBA can be instructed to ignore errors by using **On Error Resume Next**.

### Pro-Tip

Error handling and testing go hand-in-hand, they are NOT mutually exclusive. It should be second nature for you to test your routines and functions at every stage of development. (For good reason many large companies force their IT departments to record all their test procedures and results).

Once you have tested your code to the best of your ability, an error handler of some description should still be in place to handle anything unexpected or that has escaped your attention.

This routine example sets a generic handler as a catch-all for any unexpected problems.

```

Private Sub GenericError()
    Dim lngBigNumber As long
    [[Here we tell VBA to go to the label ErrorHandler if it encounters a problem]]
    [[We put the label followed by a colon at the bottom of the routine]]
    On Error GoTo ErrorHandler

    [[Create an artificial error]]
    lngBigNumber = 1 / 0

    MsgBox "Everything is fine"

    [[Note the use of Exit Sub at the end of the normal code, before the label]]
    [[This stops the program executing the error handling message box if there was no problem]]
    Exit Sub

ErrorHandler:
    [[Err is a VBA specific object that is automatically created if an error occurs]]
    [[Here, we use it to tell the user about the problem. In this case it would be 'Division by zero']]
    MsgBox "Sorry, an error has occurred." & vbCr & Err.Description
    [[On Error Goto 0 'resets' the ErrorHandler to its previous setting]]
    On Error GoTo 0
End Sub

```

The generic handler will even work for any subroutines or functions called from the main sub

```

Private Sub CalledSubError ()

    Dim lngBigNumber As Long

    On Error GoTo ErrorHandler

    [[Here we call another subroutine, that doesn't have its own error handling]]
    CreateAnErrorRoutine

    MsgBox "Everything is fine"

    Exit Sub

ErrorHandler:
    [[In this case Err.Description would be 'Type Mismatch' - a mismatch between text and numbers]]
    MsgBox "Sorry, an error has occurred." & vbCrLf & Err.Description
    On Error GoTo 0
End Sub

Private Sub CreateAnErrorRoutine ()
    Dim lngBigNumber As Long
    lngBigNumber = "Hello"
End Sub

```

### On Error Resume Next

Sometimes you may want to ignore an error and let VBA carry on as if nothing had happened. This example replaces all the cell values (X) in the selection with 1/X

```

Sub ReplaceNumbers ()
    Dim rngCel As Range

    [[The following line will let VBA ignore any errors]]
    On Error Resume Next

    For Each rngCel In Selection
        rngCel.Value = 1 / rngCel.Value
    Next

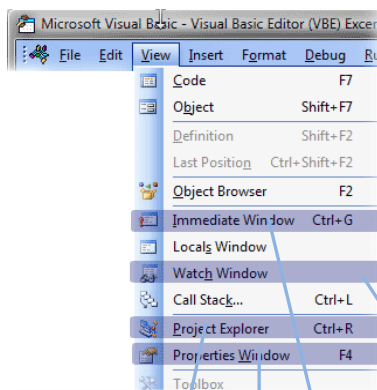
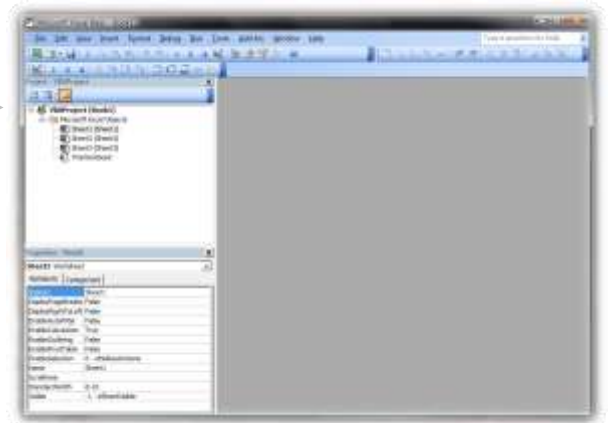
    [ [Don't forget to reset the error handling]]
    On Error GoTo 0

End Sub

```

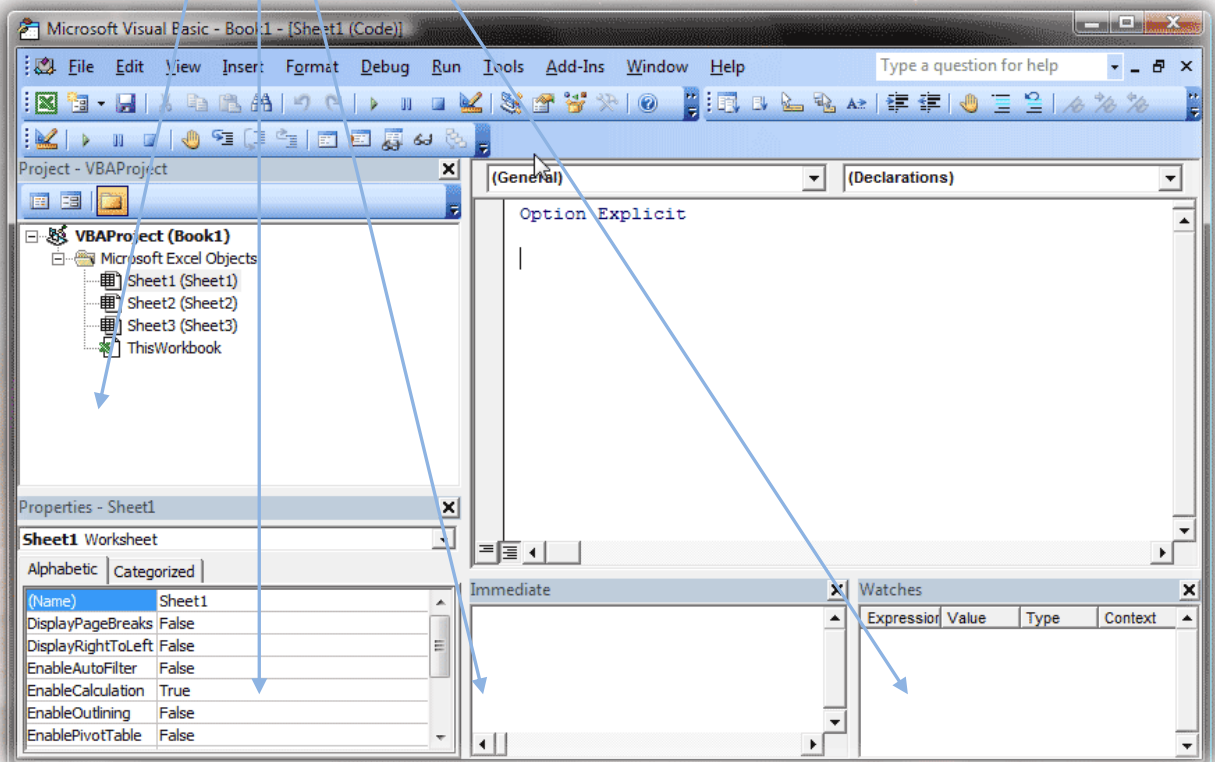
## The VBE in detail

When you first use the Visual Basic Editor, it will most likely look something like the screenshot on the right. You can change the layout, tools and views shown but this screen will soon become second nature to you. Whilst display options are very personal, the layout used in the workshop (shown below) is the one found to give the best balance between clarity, ease of use, and quality of information. It is suggested you start with this, and then experiment for yourself.



**Pro-Tip**

If you intend to use VBA extensively, go for a computer with a high screen resolution (or 2 monitors linked together). This will enable you to view the code AND the results of the code in Excel simultaneously, making debugging much quicker & easier.



### Pro-Tip

By default VBA shows an error message box EVERY time you make any syntax or logical error and move away from the line, even if you're only half way through it!

You can prevent this zealous error trapping in VBA Options, by DE-SELECTING the 'Auto Syntax Check' checkbox. Any syntax errors will still be highlighted in red so can be picked up easily.

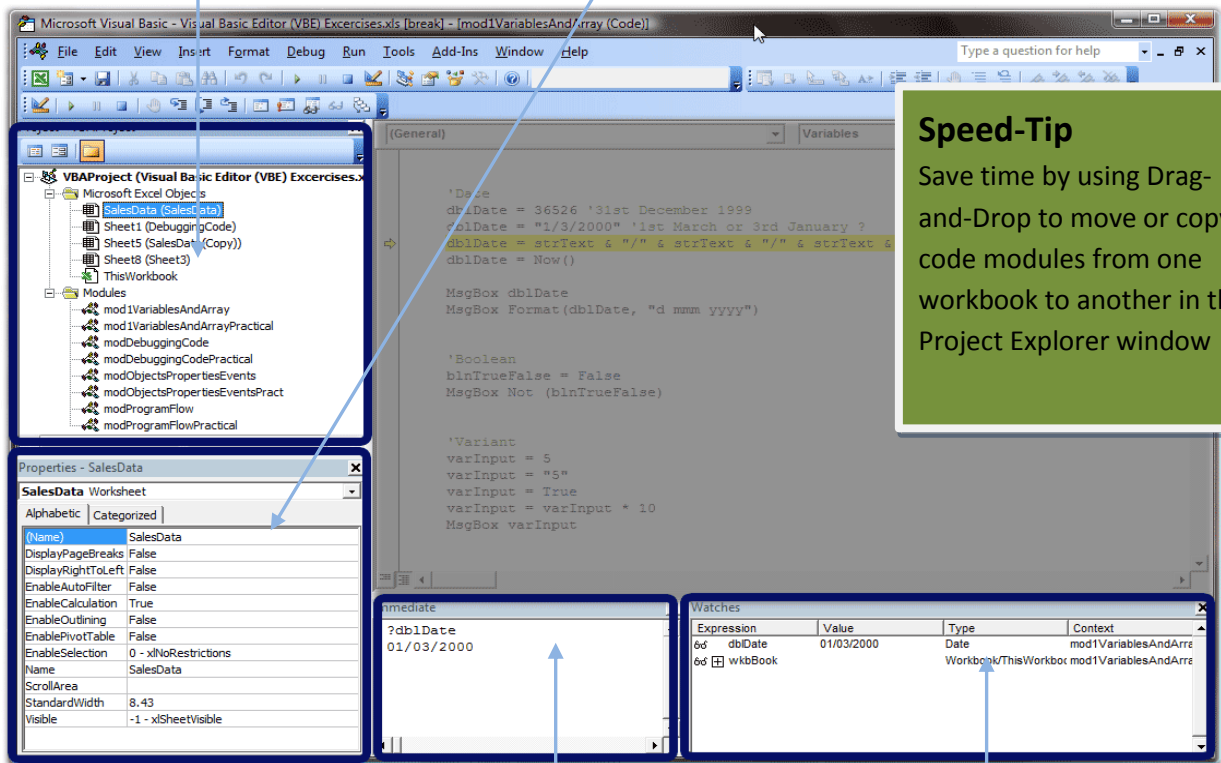


### Project Explorer Window

Used to select the workbook, worksheet, module or userform viewed in the code window

### Properties Window

View or change the properties of the object selected in the project Explorer or the selected Userform Control worksheet, module or userform to be viewed in the code window



**Speed-Tip**  
Save time by using Drag-and-Drop to move or copy code modules from one workbook to another in the Project Explorer window

### Immediate Window

Run a routine: Type the routines name  
 Check a variable's value: Enter a ? then the variable  
 Set a variable's value. E.g. `strText = "Hello"`  
**Debug.Print** will display values during code execution

### Watch Window

Right click and add a watch to a variable or object. Check its value or have your code stop if a certain condition is met.

## Manipulating Text

There are just a few relatively simple VBA text functions. Combining them together (often in a loop that changes a small portion of the text) allows you to manipulate text in advanced ways.

### CSTR()

CStr converts variables to a string if possible.

```
strText = CStr (lngNumber)
```

### LEFT() & RIGHT()

```
strNewText = Left (strText, 3)
```

```
strNewText = Right (strText, 3)
```

No error is caused if you request too many characters ...

```
strNewText = Left (strText, 1000)
```

```
strNewText = Right (strText, 1000)
```

### LEN()

The Len function gives the length of the string.

```
lngLength = Len (strText)
```

### UCASE(), LCASE()

Ucase and Lcase change the text to upper or lower case.

```
strNewText = UCase ("Abc") [[ABC]]
```

```
strNewText = LCase ("Abc") [[abc]]
```

### TRIM()

Trim removes spaces from the beginning and end of the text.

```
strNewText = Trim (" A ") [[A]]
```

### Pro-Tip

A very common requirement is to check or compare a user input against a list of items. It's a good habit to remove the possibility of erroneous spaces and upper/lower case issues upsetting the check.

```
If UCase (Trim (strText)) = UCase (Trim (strNewText)) Then
    MsgBox "They are the same"
End If
```

**MID()**

The Mid function returns a number of characters starting at a given position.

This example returns 2 characters starting at character 4.

```
strNewText = Mid(strText, 4, 2)
```

**INSTR()**

The Instr() function returns the first position of a character within the string starting from any given position.

Here it is returning the position of '4' starting at the beginning.

```
lngPosition = Instr(1, strText, "4")
```

Here it is returning the position of 'ABC' starting at character 3.

```
lngPosition = Instr(3, "ABCDABCD", "ABC")
```

Instr() is (by default) case sensitive, so the answer returned here would be 0 (Not found).

```
lngPosition = Instr(3, "ABCDABCD", "a")
```

**REPLACE()**

Replace all instances of one set of characters with another.

```
strNewText = Replace("ABC 123 DEF", " ", "-")
```

**LINEFEED CHARACTER (CHR(10)/VBLF)**

The linefeed character (Chr(10) is used to emulate a carriage return.

```
Cells(1, 1).Value = "Hello" & Chr(10) & "It's sunny today"
```

In VBA you may also see programmers use the equivalent Excel constants, **vbLf** or **vbCr**.

```
MsgBox "Hello" & vbLf & "Today is " & Format(Date, "dddd")
```

**QUOTE MARKS**

Occasionally you may wish to add quote marks to text.

This is achieved by wrapping the quote mark in quote marks!!!

```
MsgBox """"Help!""""
```

## Triggering code with Events

Once you've written your code, somehow it needs to be run!

Excel & VBA offer us many ways to start your code. These are all linked to events.

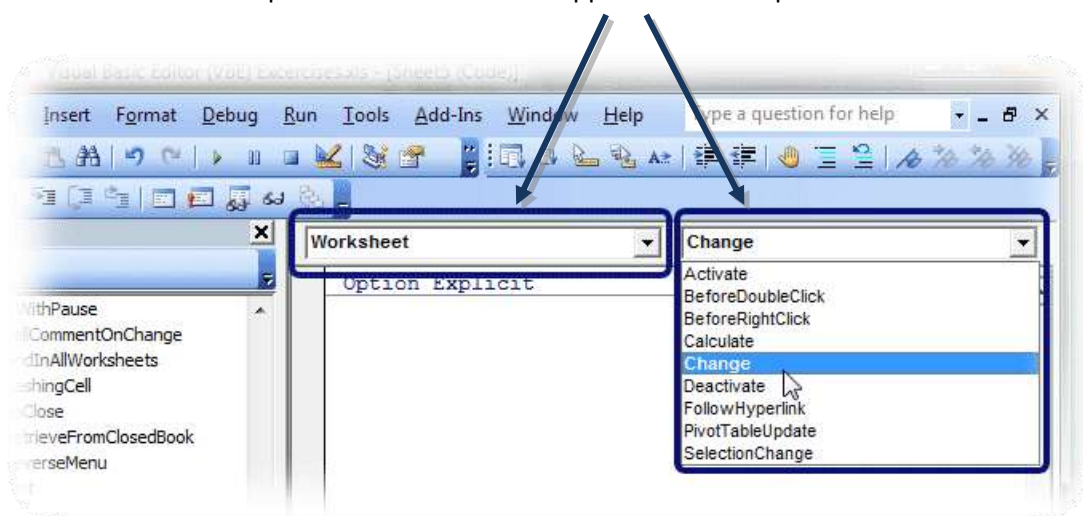
Some events are self-evident, such as the clicking of a button on a userform, or selection of an item from a menu bar, but VBA monitors many other occurrences and interactions in Excel. You can take advantage of almost any occurrence to start your code or react to the situation. You won't be surprised to hear that events are based around objects and when you combine an object with a verb, you will usually have an event that can be captured. This could be anything from resizing a window to clicking a shape.

For example, in addition to the normal clicking of buttons, some common events to start code include ...

<b>Workbook_Open</b>	The workbook is opened
<b>Workbook_Activate</b>	The workbook has been activated
<b>Workbook_BeforeClose</b>	The workbook is just about to be closed
<b>Worksheet_Activated</b>	A different worksheet has been selected
<b>Worksheet_Change</b>	A cell or set of cell's value has changed
<b>Worksheet_SelectionChange</b>	A different or specific cell has been selected

Until now, we have been placing our code in code modules. You may have wondered why VBA lists all the worksheets, userforms and an object entitled **ThisWorkbook** in the Project Explorer. You can probably now guess that in order to tell VBA what code should be run (say) when a particular object (for example a worksheet) is activated, we should put the code within *that* particular object.

From the project window in the VBE, double-click one of the worksheet objects that reside above the code modules. At the top of the code window select 'Worksheet'. In the right hand drop-down you will now see a list of the 9 possible events that are applicable to that particular worksheet.



By clicking on the relevant event on the right hand side a new subroutine is automatically created for us with a specific name. This name is understood by VBA and automatically associated with the event. So for example, if we clicked on the Worksheet's **Activate** drop down a subroutine would automatically appear, ready for us to complete ....

```
Private Sub Worksheet_Activate()

End Sub
```

The following example displays the message that the (sheet tab name) has just been activated every time the worksheet is activated either by a user or through code.

```
Private Sub Worksheet_Activate()
    MsgBox Me.Name & " has just been activated"
End Sub
```

## Events with Parameters

Sometimes the event contains a parameter. If for example we selected **Change** from the right hand dropdown, we would be face with the following routine that includes a parameter ...

```
Private Sub Worksheet_Change(ByVal Target As Range)

End Sub
```

The **Worksheet\_Change** event is invoked when any user or code changes any value on any cell(s) within the worksheet. The **Target** parameter is a range object used to define which cell(s) have changed.

This example uses an IF() statement to only show a message if the change is made on a cell in row 1

```
Private Sub Worksheet_Change(ByVal Target As Range)
    If Target.Row = 1 Then
        MsgBox "Please reflect this header change in the report"
    End If
End Sub
```

Finding out if a particular cell or range of cells has changed is a little trickier. This example uses the IF() function and the INTERSECT() function to show a message, only when a change is made on a cell within the range name 'DataRange'.

```
Private Sub Worksheet_Change(ByVal Target As Range)
    If Not (Intersect(Target, Range("DataRange")) Is Nothing) Then
        MsgBox "A value in the data range has changed"
    End If
End Sub
```

## Interacting with the user

Gathering facts from the user and providing them with feedback information is a key aspect of ANY system. With a simple system this could be as easy as a message box telling the user (or you) that the processing is complete, through to a complex project with completely dynamic and interactive wizard-style userforms, messages, and interactions.

### Pro-Tip

Informing the user of the system's progress is simple, and absolutely essential for the user to have confidence in your system.

### Msgbox

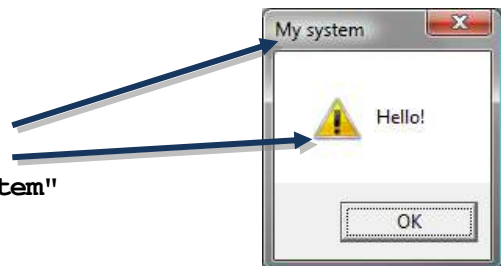
The simplest interaction with the user can be used to give or get information.

#### MsgBox "Hello"



Using some optional parameters, a better impact can easily be achieved.  
(Use Intellisense to help you with the parameters)

#### MsgBox "Hello!", vbExclamation, "My system"



```
lngAnswer = MsgBox("Continue?", vbQuestion + vbYesNo + vbDefaultButton2, "System")
If lngAnswer = vbYes Then
    MsgBox "Continuing!"
End If
```

Use a **long** variable and place the parameters in parenthesis to get the user's answer, which can then be used to act upon. Notice the use of **vbDefaultButton2** to set the button focus.

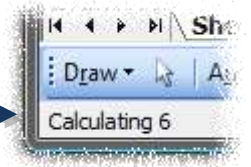


## Application.StatusBar

This is an excellent non-obtrusive method to communicate subtly to the user. It is simply text that is shown to the user at any time at the bottom left of the Excel screen, and can inform the user of progress of your code.

The following example gets the initial status bar value then keeps the user up to date with the progress of its 1000 loops. At the end the status bar is reset to its original value.

```
strStatusBar = Application.StatusBar
For lngCounter = 1 To 1000
    'Put your main looping code here
    Application.StatusBar = "Calculating " & lngCounter
Next
Application.StatusBar = strStatusBar
```



## Other Communication Methods

Two other input methods are listed below. You will probably find them not as useful as a simple message box or more flexible as userforms but will probably find them used in other people's code and so they have been included briefly here for reference. It is suggested you refer to Google Groups or Excel help for additional information if required.

### Input Box

The (Excel) input box offers a more extensive way of getting information from the user. It includes many parameters that you are unlikely to need or use (hence the large number of commas in the examples) but it automatically handles input validation and can be used to force the users to enter text, numbers or a range value. It also permits you to set a default value.

This example requests the user for a price (number) and sets the default value to 1.25

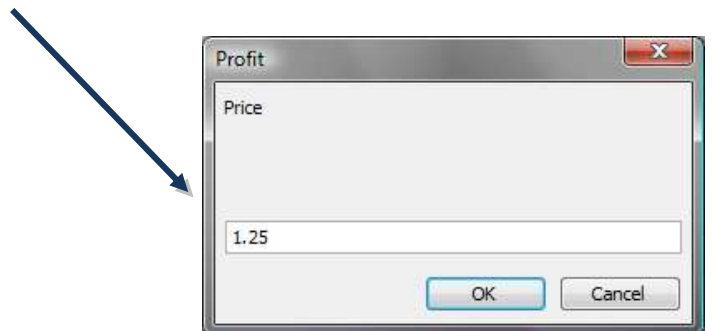
```
lngDataType = 1 [[Type 1 = Request a number from the user]]
dblPrice = Application.InputBox("Price", "Profit", 1.25, , , , lngDataType)
```

Forced Input Data Types

1 = Number

2 = Text

8 = A cell reference or range



### GetOpenFileName

Use `Application.GetOpenFileName` to request a file name and location from the user.

## Using the generic toolbar and userform workbook

### Discussion

The userform is the most powerful and communicative method in VBA from which to obtain data from a user. Unfortunately userforms and the command buttons or menu items usually required to invoke them are relatively complex to code to a professional level and training to this level would take up a great deal of the workshop time. A generic toolbar has therefore been written, together with a set of toolbar buttons and userform containing the most common controls that can easily be adapted to your own projects.

#### **Generic Command Bar and Userform.xls**

You will have been shown how to work with this tool in the workshop.

Excel 2007 completely replaced the menu bar and menu item features with an entirely new concept called the Ribbon. It is generally understood that the Ribbon is a positive step forward but it is substantially more complex for the part-time VBA programmer to adapt (it uses a different language called XML as its construct). Due to the expected demise of menu items over the forthcoming years (as Excel 2007 is gradually introduced to the corporate market), it has been decided not to invest time on instructing delegates how to work with menu items in this workshop.

It is recommended that you limit your user interactions to Message Boxes and the Status Bar described above and the Generic Toolbar, toolbar buttons and the Userform described in detail below. Additional help on menus and menu items is widely available should you deem their use to be essential.

In this section you will be shown the most common controls available in Excel VBA, and how to use them in your code. It is advised that you keep the read-only version of the generic workbook unchanged and make copies of it under different names as the starting point for all your projects.

#### **Legal Stuff**

The Generic Command Bar and Userform workbook have been designed especially for use in this workshop and as an aid in delegates' subsequent Excel VBA projects. All participants in this workshop are granted free use of the workbook and may alter any part of the workbook or code therein with the exception of the userform title (ExcelForManagers.com), the toolbar title (Excel for Managers) and the copyright notice (the ExcelForManagers module). No warranty of the accuracy, availability or functionality of the workbook or code contained within the workbook is made or implied.

## Toolbars / Toolbar buttons

In Excel 2003 (and earlier), toolbars are situated just below the menu bars and can be detached and floated around the screen if required. Each toolbar contains one or more buttons (or other controls) that can be used to invoke VBA code or standard Excel functions.

A toolbar with five generic and one userform button has been created for you in the Generic Command Bar and Userform workbook. The toolbar and buttons are visible whenever the workbook is open.



**Use the toolbar buttons to initiate any userforms in your project and/or start macros.**

### Pro-Tip

Many Excel users put buttons and other controls directly onto the worksheet. There are technical differences to this type of button, and for reasons of stability and consistency I suggest you keep all your controls on the toolbar or (preferably) userforms.

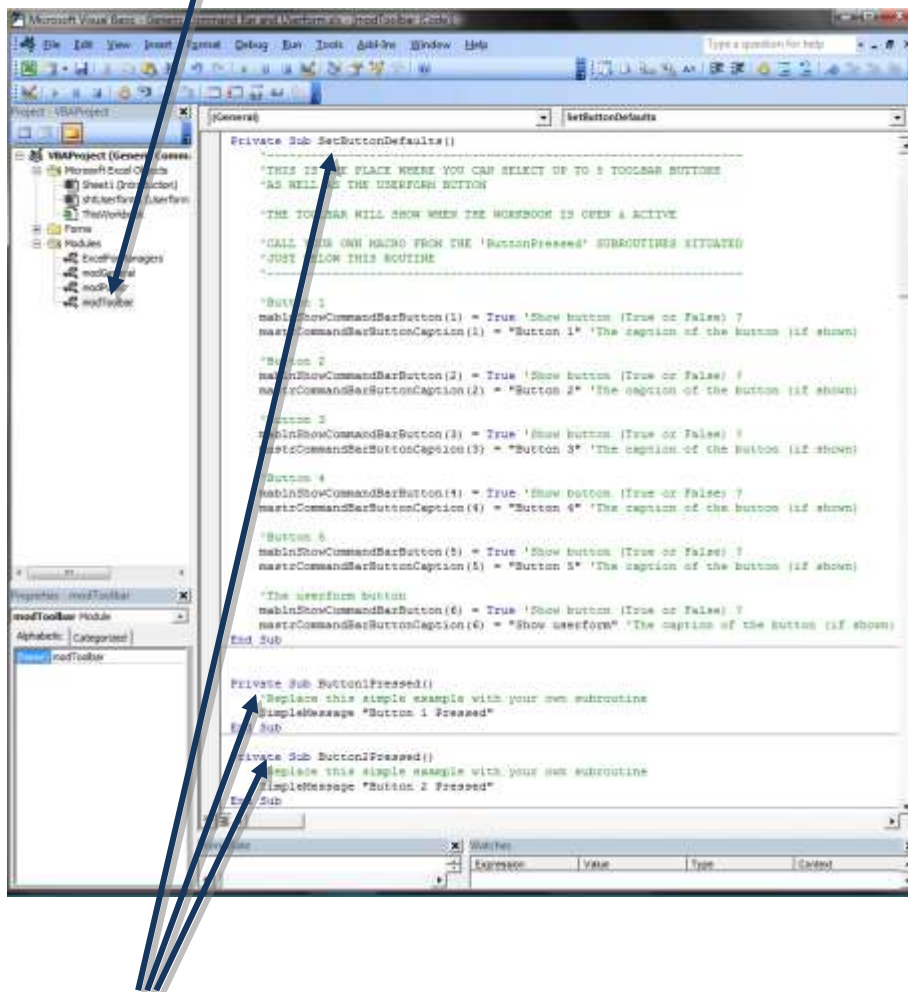
## Adapting the generic toolbar to your requirements

The generic toolbar can easily be adapted to your needs.

You can very easily ...

1. Hide or remove any of the 6 buttons you do not need.
2. Change the text shown on the buttons.
3. Instruct the button which routine to run when it is pressed.

Open the generic tool workbook and go into the VBE (Alt-F11). Double-click the **modToolbar** module from the project explorer window to show the toolbar code in the code window.



Feel free to check out at all the code in this module; however most of it is there to create the toolbar and its buttons when you open the workbook and to remove them when you close it!

You only NEED to be concerned with the first seven routines.

The first routine tells Excel which of the six buttons should be visible.

Routines 2:7 tells Excel which macro should be run when the button is pressed.

In the first routine (**SetButtonDefaults**) toggle which of the six buttons are visible and which are not using the Boolean, **TRUE** or **FALSE**– Do not make ALL the buttons invisible!

```
'Button 1
mablnShowCommandBarButton(1) = True 'Show button (True or False) ?
mastrCommandBarButtonCaption(1) = "Button 1" 'The caption of the button (if shown)

'Button 2
mablnShowCommandBarButton(2) = True 'Show button (True or False) ?
mastrCommandBarButtonCaption(2) = "Button 2" 'The caption of the button (if shown)
```

In the six following routines, replace the entire line of the **'SimpleMessage'** routine with the name of your macro that should be run when the relevant button is pressed. Note there is no need to change or delete any of the subs that will not be visible on your toolbar.

```
Private Sub Button1Pressed()
    'Replace this simple example with your own subroutine
    SimpleMessage "Button 1 Pressed"
End Sub

Private Sub Button2Pressed()
    'Replace this simple example with your own subroutine
    SimpleMessage "Button 2 Pressed"
End Sub
```

**That's it!**

In order to see your changes you will need to re-run the **'CreateToolbar'** routine (a little down the page) or re-open the workbook.

### Pro-Tip

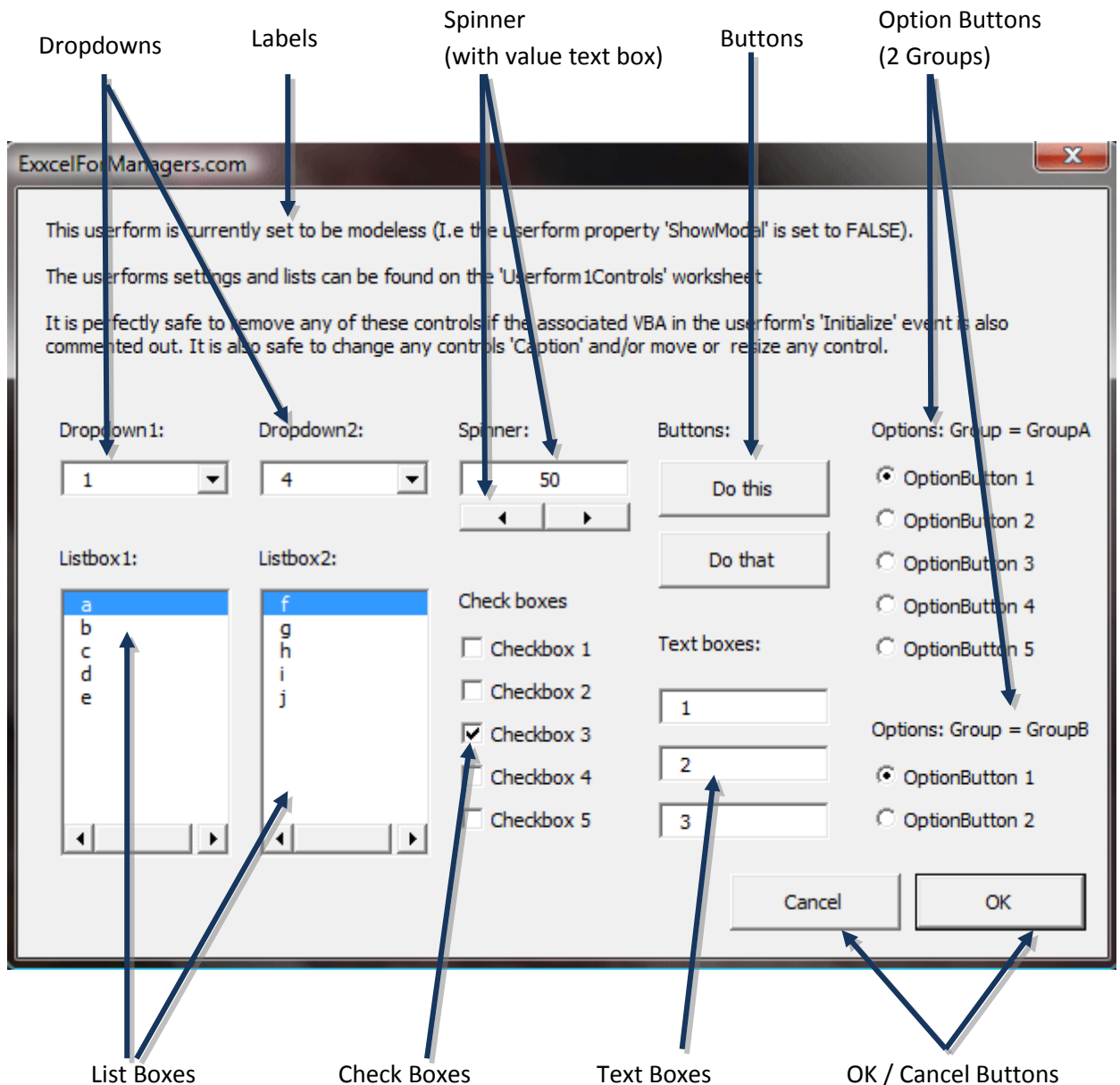
*Generally speaking, once you enter the VBE and change any code, VBA 'forgets' any variables that have been set (for example the variable name of the toolbar). This means that whilst you are changing or playing with the toolbar settings the toolbar will not always be deleted as it should be, and you can end up with several toolbars all looking the same - This is normal!*

*Once you have finalised your toolbar, close and re-open Excel (not just the workbook) and all will be restored, just as it should be.*

## Adapting the generic userform to your requirements

The generic userform can be invoked at any time by pressing the 'Show userform' button on the generic toolbar. It is suggested you change the text to give this button a more meaningful name for your userform tool (see previous section) before you implement your project.

When you first press the button, you will see a userform similar to the one below, that should contain most or all of the controls you need. Each of these controls are described in detail in the following pages, together with how they may easily be adapted to your own project.



The userform has been designed to be simple to use, and link in with standard Excel functionality.

**Pro-Tip**

This easiest way to 'combine' VBA with Excel functionality is to use VBA to set values on worksheet cells. These values can then be used to drive IF() or OFFSET() functions within the worksheet, and provide a seamless interface between the two.

As an additional benefit the values can be used as seed or default values next time the workbook or userform is opened.

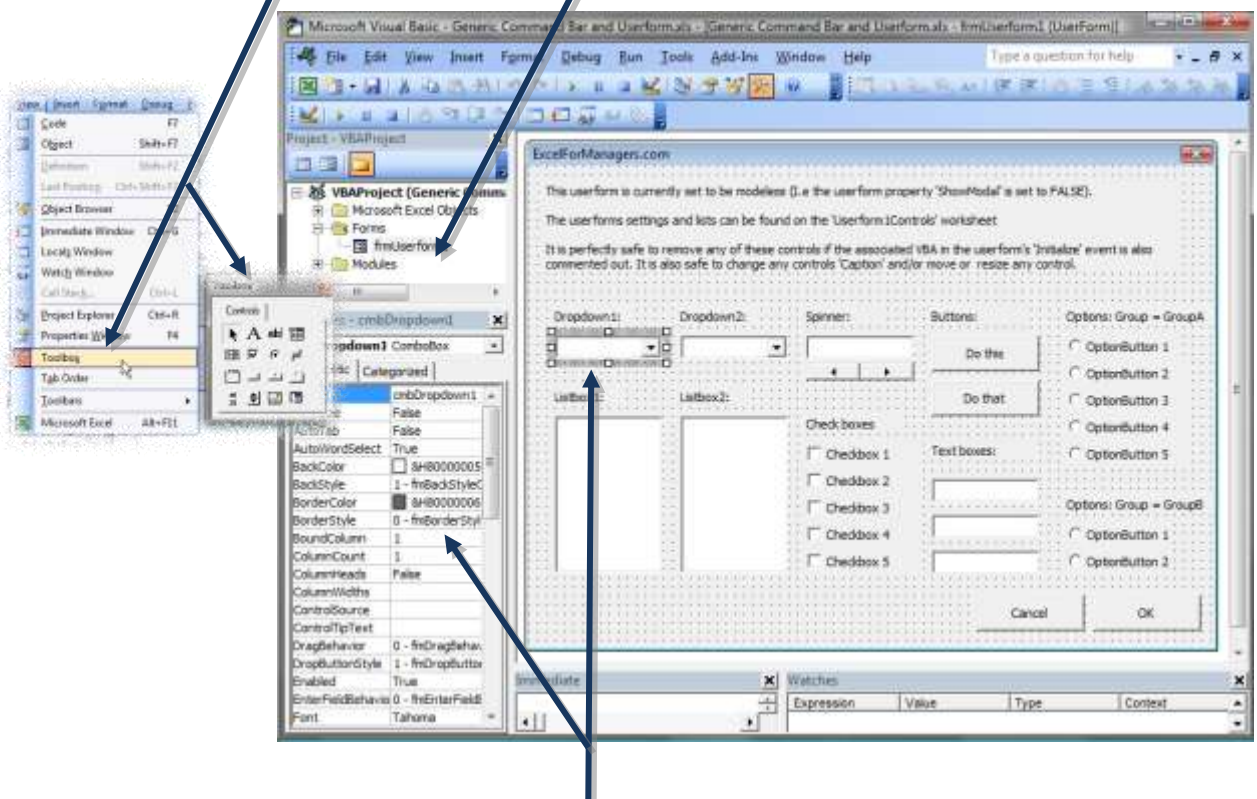
Activate the worksheet entitled **UserForm1Controls**. You will see a worksheet similar in content to the one shown below. Now open the userform and change some of the figures or values. You will see that each change is immediately shown on the worksheet, in one of the purple cells.

	A	B	C	D	E	F	G
1	<b>Userform Values &amp; Lists</b>						
2	<b>These are the control values on the userform 'frmUserform1'</b>						
6							
7	<b>Spinner</b>	<b>Value</b>					
8	Spinner: Min	-10					
9	Spinner: Max	100					
10	Spinner: Increment	1					
11	Spinner: Value	25					
12							
13	<b>Checkbox Name</b>	<b>Value</b>					
14	Checkbox 1	TRUE					
15	Checkbox 2	FALSE					
16	Checkbox 3	FALSE					
17	Checkbox 4	FALSE					
18	Checkbox 5	FALSE					
19							
20	<b>Option Buttons</b>	<b>Selected button</b>					
21	Group A	optGroupAOption1					
22	Group B	optGroupBOption2					
23							
24	<b>Text Box Name</b>	<b>Value</b>					
25	Text box 1	Box 1					
26	Text box 2	Box 2					
27	Text box 3	Box 3					
28							
29	<b>Dropdowns</b>				<b>Listboxes</b>		
30	<b>Dropdown 1</b>	<b>Dropdown 2</b>		<b>Listbox 1</b>	<b>Listbox 2</b>		
31	<b>Default Item</b>	<b>Default Item</b>		<b>Default Item</b>	<b>Default Item</b>		
32	1	6		a	f		
33							
34	<b>List (leave next row blank)</b>	<b>List (leave next row blank)</b>		<b>List (leave next row blank)</b>	<b>List (leave next row blank)</b>		
35							
36	1	4		a	f		
37	2	5		b	g		
38	3	6		c	h		
39				d	i		
40				e	j		
41							

Some of the cells are coloured in lemon. These are for you (or VBA) to set the default settings or list contents before the userform is displayed.

To see the userform in design mode double-click the `frmUserForm1` Form in the project explorer.

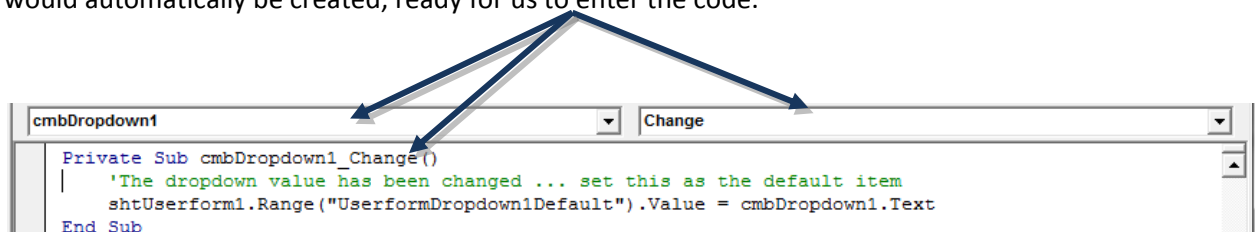
To see the list of controls you can drag onto the userform, press View / Tools.



Userform design mode enables you to add and remove controls, and add, remove, resize or change the properties of the userform or any controls contained therein. Select any control and its properties will immediately be shown in the properties window. Any of the Control's properties can be changed at any time during development (such as size or position) or with code during run time (such as enabled, visible, or value).

Double-click the control to switch to the code window and to show the code that will be run when the control is clicked, selected, changed etc.

This screenshot shows that when the value `cmbDropdown1` is changed, the corresponding value on the spreadsheet is set to the dropdown's value. If we selected a different event from the right hand dropdown list (say `dblClick` which would capture the double click event) a new blank subroutine would automatically be created, ready for us to enter the code.



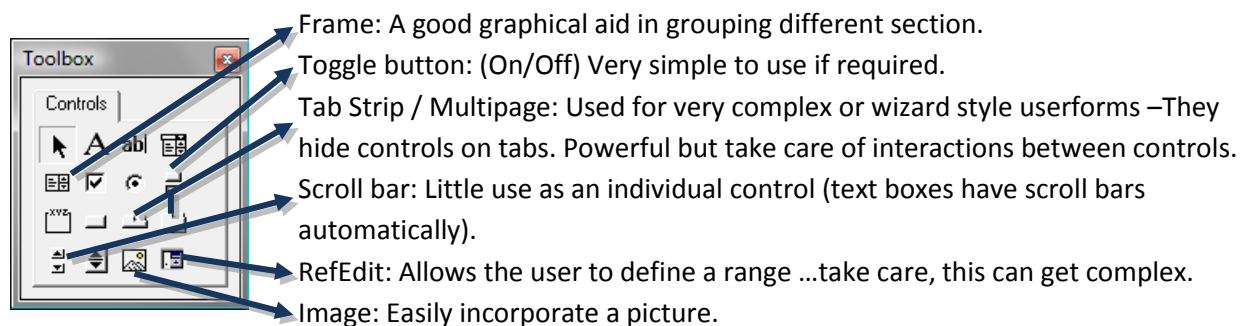
When you have finished any code changes, get back to the userform by double-clicking the userform in the project explorer again.

## Modal / Modeless Userforms

A modal form is one where the user must complete the form by pressing OK or Cancel BEFORE they are permitted to continue. A modeless form is one where the user may continue working on the spreadsheet or elsewhere in Excel whilst the form remains open. The generic userform is initially set to modeless, however you can change this to modal, by changing the **ShowModal** parameter in the userform properties within the VBE.

## Missing Controls

There are a few controls that have been purposefully omitted from the generic userform, either because they are slightly more complex to work with, or not used with such regularity. However you should be aware of them and be able to incorporate them in your project if required.



## Project Quick Start

Here is a quick guide to amending the userform to fit your needs ...

- Take a copy of the generic workbook and save it as read-write with a meaningful name.
- Enlarge the userform, move (do not delete) any controls you will not need off to the bottom or right hand side of the userform.
- Reposition the remaining controls, and change / add the label controls as required.
- If required, set (or write code to set) the lists for the dropdown(s) and listbox(es).
- Once you're happy with the layout, resize the userform (leaving the un-used controls hidden) and check/test your userform.
- If you are happy with the result enlarge the userform and delete the un-used controls.

## Looking at projects

### The three elements of every Excel VBA project

Every computer program ever written performs the same basic function...

- It takes an input of information.
- It processes the information.
- It outputs a result.

Good spreadsheet designers and programmers always separate out these three distinct elements of their systems, for good reason ...

- Less (or more controlled) impact if part of the system requires change.
- The individual elements are more likely to be re-usable.
- Easier to follow and understand.
- Independent testing can be carried out on any of the three elements.

#### Pro-Tip

For those who are aware of or need to know about SOX compliance, Sarbanes Oxley has recognised the importance of the separation of input, calculations and output in designing accountable, readily auditable and structured workbooks. It is one of the key elements of a compliant Sarbanes-Oxley implementation.

## How to spot a well (or badly) designed system

In our travels at some of the world's most prominent companies, we have seen and worked with hundreds of other people's workbooks, databases and VBA systems. There are numerous aspects to each system design, and dozens of personal judgement calls that their designers made along the way. Often, there are several completely different approaches that would have met the physical requirements of the project, making the system design almost as much of an art as a science. Without a complete knowledge and understanding of the numerous defining factors of the requirement, such as time available, project-life-expectancy and designer's skill, it is unfair to say for sure whether a specific spreadsheet or VBA tool was well designed and executed. However, whether it be a single page template or a complex multi-user VBA tool, the same features crop up again and again in respected, reliable tools, and conversely several factors always seem to be present in troublesome projects. Ensuring your project has the all the features associated with the best systems is an excellent first step towards well designed, manageable, extensible and reliable systems. Here they are ...

### Inputs, Outputs & Calculations kept apart

Excel handles multiple spreadsheets very efficiently. There is absolutely no need to cram a single worksheet with lots of different inputs and calculations. Separate user inputs, data inputs, constants (such as VAT or tax thresholds), and calculations in different sheets or different parts of a worksheet. If the workbook includes graphs, report pages or a dashboard then use cell references back to the data and calculation sheets. If your project includes VBA, create different modules for each aspect of your code. Workbooks set out using this method are much easier to understand, change or extend.

### Verified Inputs

Whether it be strict adherence to a defined external database structure, or a simple cell-drop-down validation, good tools *always* control their input of data. Rubbish in – Rubbish out ... Always!

### Complex problems, simplified

Once you get 'into' a project it is very easy to get carried away with clever or complex VBA code or cell formulae. That's all very well until someone else has to follow your work or you want to make a change to the logic and have to re-create the thought process you were following when you wrote it. Without a single exception all good workbooks and VBA systems distil complex problems down to their component parts and build the solutions up in logical, simple-to-follow steps ... even when this is at the cost of some extra columns of data, a less efficient (slower) sub-routine or a larger file. These trade-offs are miniscule compared with the benefits of being able to follow your own logic at 2am when the boss needs your report on his desk first thing in the morning!

**Intuitive**

Good tools are always clear and logical to the user, and follow a defined path to achieve their goal. It has been found users appreciate colour coded worksheet tabs, cell ranges or fonts, or different cell formatting to differentiate inputs from outputs and always try to emulate that feature in our own systems.

**Built in sense checks and/or Error handling**

Unexpected things happen ... there's no getting away from it no matter how well you design your system and test your code. Good systems anticipate problems and handle them. That is not to say that your project should necessarily skip over the issue or correct the data, but should alert the user when a potentially damaging situation or data problem occurs. This may be something as simple as a sense-check with conditional cell formatting to highlight a potential issue, or a complete and full error handling routine built into the code.

**Known limits**

Systems that have well defined and thought out scope are far more likely to be successful. Tools that attempt to be all things to all people are extremely difficult to design and even more difficult to implement and maintain. Ascertain your requirements and stick to them.

**Pro-Tip**

Most Excel systems are either tools for reporting or data analysis ...  
You need two systems for these very different functions.

**Well named, well described**

Good systems invariably have a standardised or logical naming convention. Sheet names, Range names, Title headers, Variable names (if VBA is used) and intermediate calculations will all be consistent and descriptive from first input through to final output.

Note: It is impossible to say for sure how much thought went into any completed tool *before* it was built (people will *always* tell you they thought everything through first), but a solid naming convention is a genuine tell-tale sign of good planning and being able to visualize the entire solution *before* it was created.

### No duplication

Primary keys and data integrity are terms more akin to full-time programmers, but the consequences of not adopting some of the rules of professional computing standards can wrong-foot otherwise superbly designed tools with frustrating consequences ... here is an example.

Let's say we are working with our database of 100 different employees. We want to display the hours worked for any particular employee, so we include a drop-down of each employee's surname plus initial and use a **VLOOKUP ()** function to display that employees' information .... Fair enough?

Modern relational databases all include a primary key ... an identifier that **UNIQUELY** identifies a row of data, and this is for a very good reason... What happens in our new system when a second John Smith starts work for you?

This may sound obvious, but if you need to uniquely identify an item - use a *unique* identifier.

### Playing to Excel's strengths

Excel is a superb product. It is capable of any number of tasks from number crunching to picture editing. However, like any product it is best suited to its primary task - Numbers. There is no shame in having (for example) three stages to your automatic procedures .... A database system to query your data, Excel to manipulate it and PowerPoint to display it. All the best systems use Excel for its core purpose. Two areas where Excel is often used and its implementation is never truly successful are when Excel is used as a (badly designed) relational database, and when it is programmed to handle file management. Don't be afraid to rule Excel VBA out as a project option.

#### Pro-Tip: Problem System Checklist

These are tell-tale signs of a problems system. If a tool you know has half or more these pointers, it's probably heading for trouble...

- Input data mixed in with calculations.
- Lots of links to other workbooks.
- Overly-Automated (the 'Press-one-button' system).
- No input or data validation.
- No data integrity checks.
- Overly complex or long cell formulae.
- No version-control when updates or changes are made.
- Different headers/names used for the same figure (e.g. Profit, Margin).
- Only one person understands the tool.
- No documentation.

# The Nine Point Project Plan

**Fail to plan – Plan to fail!**

This saying is very close to us, because it could not be more applicable than in the world of workbook development and programming.

Below is the nine point plan that is strongly advised you use EVERY time you undertake an Excel project. Spending 5 or 10 minutes, even on the smallest of projects and asking yourself the right questions gives you the confidence of an end result before you start and WILL save you time. It has the added benefit of avoiding the very realistic chance of getting half way through the project only to realise it was more complex than you thought. This realisation has been seen on £-multi-million projects on more than occasion by professionals who should know better. - They had to throw away months of work.

## Planning

1. **Best approach?**
2. **How can that be done?**

## Confirming

3. **Proof of concepts**
4. **User Interface**

## Coding & Designing

5. **Output**
6. **Input**
7. **Data Manipulation**

## Finalising

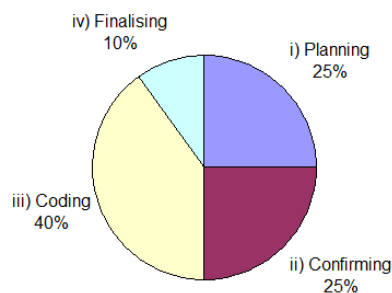
8. **Final User Testing**
9. **Handover**

### Pro-Tip

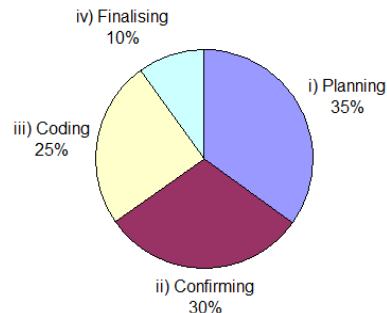
You WILL be tempted to dive straight into a solution and blissfully start coding. DON'T!

The charts below gives *general* indications of the time you should allocate to each stage of your project. Note how the percentage planning time increases with the project's size and complexity. TESTING (whether it be your theories, assumption or code) should be integrated into EVERY aspect of the project.

**Small Projects**



**Large Projects**



## Before you start ...

Before we consider the nine point plan in detail, let us consider the most fundamental question of them all .... Should we undertake the project?

This most simple of questions is all too often completely overlooked, especially when the project is expected to be small enough that no escalation of costing approval or acceptance is required. Before starting or agreeing to ANYTHING, ask yourself some fundamentals questions...

- Is it worthwhile automating or coding this project?
- What is its expected lifetime?
- How easy is it to achieve some or all of the results manually?
- Is it cost effective to take on this task?
- What are the expected time & costs savings of a solution?
- Do I have the time and resources to undertake the requirement?
- Can it be done to Budget and/or Deadline?
- Is this best achieved in Excel / VBA?
- Are there security issues or a need for a web interface or multi-user features?
- What are the risks or consequences of non-delivery?
- What are the consequences if the system gives incorrect results?
- Do I need to, or am I able to maintain the solution after its completion?
- What will be the process if bugs or problems come up after delivery?

Only you can make a judgement call on these questions, and their answers may not be simple. For example your code may only save you an hour a month and take 2 weeks to write, but if you only have 2 hours to turn-around a critical report each month, a one hour saving is very worthwhile.

Once you have posed these questions, you may decide not to undertake the project at all or to look for other solutions.

### Pro-Tip

You considered the project or automating the workbook thoroughly and decided an alternative approach was better.

Whilst, as an Excel specialist that may be disappointing news, it is definitely good management on your part!

Ok, the project can be justified and we want to do it Excel/VBA. Let's start in earnest ...

## Planning

1. **Best approach?**
2. **How can that be done?**

The first two actions in our plan are listed together for good reason ... they are asked together (or sequentially) and form a loop.

Consider the solution in its totality and start simply. Describe to yourself (or write down) your requirement in a non-programming way. For example ...

*"I need to take sales data each week and produce a table of the best performing products that week"*

By the end of this planning stage of the project we want to change our non-programming statement into a series of logical computing routines and functions by breaking down the requirement piece by piece.

In our example, our requirement can be broken down into two parts ...

1. Get the Sales Data each week
2. Produce a table of the best performing products

Shortly, we will carry on breaking down each aspect of our requirement, asking the two planning questions until we get right down to a subroutine or function level, but note that whilst the above breakdown is still at the non-programming stage it is not too early to start considering issues and posing questions. For example, when we 'Get the sales data each week' ...

- Will the data arrive in a form readily accessible to Excel?
- Will the data arrive in one file, or are there ever additions / amendments?
- Will the data always contain 5 weekdays worth of sales?
- Will the data structure (column headers) be the same each week?
- Will it fit on one spreadsheet (255 columns by 65k rows)?
- Is there a password on the data file?

### Pro-Tip

Don't dismiss (or allow somebody else to dismiss) a routine or function as easy unless you have either done something similar already or have thought it through properly ... remember the sentence-to words function?

## Discussion Page

Only you, as a subject expert can ask and answer the questions from the previous page. Many of your self-posed questions can often be discounted in a moments thought. For example it might not matter to you whether the data file contains 5 or 7 days worth of sales; however some answers are of more consequence...

Let us say that occasionally the sales team work Saturday mornings. When this happens they send through the standard week's sales file on Monday, but also an additional file for the Saturday sales on Tuesday. You are now faced with several options, each of which have their own consequences ...

1. Ignore Tuesday's file and be explicit in your report it includes Monday-Friday sales only.
2. Write your system in such a way that it can handle the additional file.
3. Manually concatenate the data into one file whenever this happens (or write an additional automated tool to do this for you).

So, something that started out as simple as, 'Get the data' now has far reaching consequences on the data that will be contained in your report, the day in the week on which you deliver it, and its overall relevance and accuracy.

### Pro-Tip

Take your time and ask yourself the difficult and challenging questions at the beginning of the project - It will save you much deliberation and problem solving towards the end!

In this imaginary situation, whichever decision you made would have impacted your solution in some way, and will probably have placed some limitations on your project.

### Pro-Tip

It is essential you understand and if necessary communicate the limitations of your project before you start coding. Re-assessing or amending a project in light of the need to remove a limitation can have massive consequences.

Making and balancing decisions with their inherent implications and limitations are a key aspect of an analyst's skill. Not only do these decisions need to be made at a high level (such as the example above) but at EVERY level of your project. For example when you undertook the (simple!) routine to separate out words in a sentence you had to make decisions about what constituted a word. This skill comes with business understanding and practice, but asking yourself questions whilst you consider the solutions will give you the best possible opportunity to get it right first time.

We will now follow this process through one 'path' of the project ...

*(You will need to follow this logic through many paths in order to analyse the WHOLE system)*

'Get the Sales Data each week' ...

The question of the best way of doing that is more one of business process (should the data arrive weekly instead of monthly data etc.) so we will take it that this has already been decided and concentrate on the more technical programming aspects of this point. Here is a typical thought process.

#### **How can that be done?**

Copy the required fields from the data workbook into the report workbook.

#### **Best approach?**

We don't have to copy the data, but that would mean we would have to re-open the data file if we wanted to check the results, and would mean that we would have to work with two files (the data file and the report workbook file) during the data manipulation stage, so on balance it probably is best to open the data file, extract the data and close it again.

#### **How can that be done?**

We will need to open the file from a defined name and location, or we could ask the user for the file name and location, or we could force the user to open the file for us first.

#### **Best approach?**

The easiest for us would be to make the user open the data file for us, then invoke the macro. A defined path and file name is too restrictive so we'll rule that one out. Asking the user is the most flexible but that will mean we have to write any code to delimit the database file into Excel format and also that alternative won't work if the data is emailed unless the attachment has been saved... On balance we'll get the user to open the file in an Excel format and then get them to run the macro.

#### **How can that be done?**

We need to ensure an 'import data and start report' button is shown when the user has opened the data file, and check that the active file is the correct data file.

*(Note that is two actions so we would eventually need to follow both path's to their conclusion ... we will stick with the first required action, making a button available)*

#### **Best approach?**

A menu item or a button on a command bar would be ideal as the user would be able to access it no matter which workbook was active. The generic userform and toolbar contains a command bar and button we could re-name and use. The macro would need to check the validity of the active worksheet on the active book and import the data into a blank data worksheet on the report

*(Again we have multiple paths ... this time we'll follow the 'validate' data path)*

#### **How can that be done?**

A function will need to check that the first row header titles match the data headers we expect, alert the user which header(s) do not match if required and tell the user the macro can't continue.

**Continue this process until all your paths have ended in a detailed description of a function or sub-routine.**

You can see that this planning process is long but methodical. It may take you many hours, days or even weeks to complete.

**Pro-Tip**

Don't be afraid to 'jump' around the numerous paths that will be required for you to analyse the system entirely, or even jump into the 'Proof of Concept' stage of the project if you are unsure if a promising approach is possible or not.

Especially in larger systems it can work well when you haven't finalised ANY path, but worked down the whole project one 'layer' at a time ... you won't have finalised anything but you will have a good idea how the project will work as a whole. You can then be more confident there are no nasty surprises that will require a re-think of your system.

In smaller projects the entire planning process can usually be done in your head. For larger or more complicated projects (say ones that will take more than a few days) it is always worthwhile documenting your thoughts and/or creating simple process diagrams.

**Confirming****3. Proof of Concepts**

A proof of concept is a process by which you prove how complicated it would be, or indeed if it is possible to achieve a given solution. It is strongly advised that you advocate of this approach, which is easily skipped, but can save days or even weeks of wasted programming time.

Whilst you were in the planning stage and asking yourself 'How can that be done' you would hopefully have given yourself an idea of just how complicated that approach, function, routine, formula, response or calculation would be. Are you SURE it can be done that way? .... If you're not 100% sure of whether it can be done, or you are sure it can be done but not sure ROUGHLY HOW DIFFICULT it will be, now is the time to prove it!

Let us say you have decided that you want to show a chart with two different axes ... you're sure you've seen something similar but you've never created one yourself. Use an extract of real data then open a blank workbook and try to create the chart yourself.

If you find the path not possible, not feasible or not practical go back to your planning stage and re-consider other options...

***THAT IS GOOD ANALYSIS - NOT A FAILURE ON YOUR PART!***

By the end of this phase you should be 100% confident that your proposed solution is definitely, categorically possible and feasible. You should also have a good idea (say within 25%) of the total time that the solution will take to develop...

**IF NOT - CONDUCT MORE PROOFS OF CONCEPTS**

#### 4. User Interface

The final section of the project, before you get down to coding or creating workbooks, is to design the User Interface (UI). The UI is ANYWHERE where the user interacts with the data or your code.

In simple solutions this may be non-existent, or simply a matter of a drop-down list with data validation, however in larger tools this is likely to be one or more userforms.

#### Pro-Tip

Creating the userforms just as the user will see them in the final solution gives both you and the user a good opportunity to consider any issues, anomalies or potential problems BEFORE you start coding.

There is no need at this stage to write any of the code behind the userform controls, simply to create the look and feel of the finished product.

Take care in the design of your UI's.

- Align controls and group them together.
- Keep the UI simple and intuitive
- Use multiple userforms rather than clutter controls

Once you have finished designing your UI consider its use carefully. Take time to think through the logic of the UI, and work through how it will be used. Ask yourself questions such as ...

- ...if I select from this list, how does it effect that list?
- ...is it valid for the user to combine that option with that request?
- ...should that check box always be available?
- ...is it logical for the user to request that under this condition?

If you are not the final or only user of the system, go back to the user(s) at this stage of the project and show / discuss the UI and project with them. It is a great opportunity to discuss the limitations that you have accepted, and how you both see the tool being used. Don't do this as a matter of course, but genuinely engage the users and don't be afraid to go back to the planning stage and re-consider or re-design to meet any genuine needs or resolve any misunderstandings.

**This stage is your last chance to re-consider your solution without having to make major changes.**

By the end of this phase you should be 100% confident not only that your solution will work, but that you have not misunderstood or mis-communicated the project and it will work well. If you have done your job properly through to this stage then there will be NO big or nasty surprises through the rest of the project. You have done 90% of the hard work!

## Coding & Designing

We are now into the meat of the project. If you recall back to the three aspects of any computer program (and SOX compliance) then we need to differentiate the Input, Processing and Output stages of our tool. The order in which you produce these three discreet sections is more one of personal choice and not really of consequence. Many programmers like to start at the beginning and work methodically to the end ... and that's fine!

The reason it is suggested to do the output, then input, then calculations, is that occasionally whilst constructing the output layer, and very occasionally whilst writing the input layer you may come across extra features, extra requirements or ways to improve the system which can easily be incorporated, or small incongruities that need to be addressed. 99% of the time, these small changes can be incorporated simply into the tool with no additional work except for small future adjustments in the data manipulation layer.

### 5. Output

Unless your output is another file, this will be the part of the workbook that the user will see. Keep your designs simple and logical. Try to use no more than three main colours. It usually goes down well with the boss to use company logos, fonts and colours if possible. Keep output sheets separate from the input and calculation sections, and use names ranges abundantly.

### 6. Input

This is usually code to handle one or more input files, which should be copied to an 'internal' database sheet or sheets. If the tool includes user inputs remember to store them on a separate sheet again.

#### Pro-Tip

Remember to clear the data and reset the user settings before new data is imported. Bare in mind the user may import the data more than once.

### 7. Data Manipulation

This is usually the most intense and time consuming aspect of your coding. Remember to keep your code and intermediate data separate from any input and output data, and to use a combination of Excel functions and VBA where possible. Use simple, methodical steps even if this is less efficient or slightly slower and show/comment your workings!

## Finalising

### 8. Final User Testing

It goes without saying that you should have been testing your functions and routines at **every stage** of your project. Inevitably however, there are aspects of testing that can only be carried out once the project has been fully assembled. Try to work through as many different end-to-end scenarios as possible. If you are not the only user then get the other user(s) to also test your project and accept that the project is fine or convey to you what errors need to be corrected. (This is known as UAT or User Acceptance Testing)

#### Pro-Tip

Do not be tempted at this stage to go back and re-design chunks of your system. This stage is **ONLY** to correct errors.

If you do find a critical design flaw at this stage then you **MUST** go back to the Planning Stage (with all the painful time and cost consequences that involves) or design a completely separate and discreet 'bolt-on' that will work around the issue.

### 9. Handover

If you have designed the system for your own use, this is simply a matter of saying to yourself "right, that project is now finished" and **NOT** being tempted to keep re-visit it making minor enhancements (That is much more difficult than you might think!)

If others are also using the system it is human nature for them to start asking for enhancement or changes immediately. It is recommended you resist the temptation to oblige straight away, but instead get them to list all their requests down together for group discussion in (say) a month's time.

(also see the section 'Designing for Bob' for other issues on completely handing over a project)

***Job Done !***

## VBA & Charts

Creating and manipulating charts with VBA was not covered in the workshop!

VBA and charts are unhappy bed-fellows and whilst it is indeed possible to achieve most charting requirement directly through VBA, a disproportionate amount of time is often required to achieve professional results... Problems include handling bad or missing data / data series, poor colouring, graph re-sizing issues and difficulties when series data cross 0 on the y axis amongst others. It is often far more convenient to create the graphs manually and use VBA simple to show, hide or possibly set the source data range ... nothing more!

### *Pro-Tip*

If you have attended the Advanced Excel Users workshop you will know that Graphs can be made to look professional and work completely dynamically without the need for VBA. In summary ...

- Use a reference formula in the Chart Title to allow it change dynamically
- Use OFFSET() to show different data sets without changing the source range
- Hide Rows of Columns of Data to prevent them being shown in the graph

*(There is a very simple example of a dynamic chart in the Generic Toolbar and Userform workbook)*

If you really need to use VBA to manipulate charts, it is suggested you do this as a proof-of-concept and allow plenty of time for coding and testing of the solution.

### Chart Sheets

Chart sheets have the disadvantage do not have the same level of control of aspect ratios and what can be printed alongside it, and it is suggested that displaying charts of standard Excel worksheets offers more flexibility. However, if there are occasions when should you wish to use Chart Sheets, please be advised ...

They are not part of the **Worksheets ()** collection

(ChartSheets & Worksheets *are* both part of the **Sheets ()** collection)

ChartSheets are essentially chart objects and do not require the ChartObjects object

The example below shows how the charts on chart sheets and worksheets are referenced differently ..

A chart on Sheet 1

```
Msgbox worksheets("Sheet 1").Chartobjects(1).Charts(1).Name
```

A chart on ChartSheet 1

```
Msgbox sheet("Chart 1").Chartobjects(1).Charts(1).Name
```

## Getting additional help

By the end of the workshop you should have the understanding, knowledge and confidence to answer the BIG Excel VBA questions. Questions such as ...

- Should I do this in Excel, or VBA, or use both?
- Should I attempt this at all?
- What problems might I face if I do it this way?
- What will be the impact if the data headers change next month?
- How much testing do I need to undertake?

These questions involve value judgements, and often do not have a single correct answer. As your experience grows you will be able to answer them with increasing knowledge and confidence but what is for sure is that you will not find any answers to these sorts of real-world issues in any VBA book or help file.

You may have noticed that whilst numerous examples of VBA were introduced throughout the course, time was prioritised towards discussions of the bigger issues, not on practicing the intricacies of syntax.

This was for two very good reasons ...

- Unless you program full time, you are unlikely to remember it all.
- It is relatively quick and easy to get answers to this sort of problem:- You can easily copy other people's code, obtain similar-syntax routines or get help from readily available (and free) sources.

These relatively simple (and to me, unimportant) problems can be summarised with questions such as ...

- Can this be done?
- How do I achieve this?
- Why wont this work?
- What is the correct syntax to do this?

This chapter is dedicated to helping you find this type of information quickly and easily.

### Pro-Tip

Never be afraid to copy other people's code (copyright permitting). You may need to amend it to suit your own specific needs or style but it is usually the fastest way to get things done....why re-invent the wheel?

However, ALWAYS thoroughly test any code you have acquired from other programmers as if it were written from scratch. Never assume other people's code will just work .... it very often doesn't!

## Top 5 Excel Resources

### Google Groups

This should be the first, and probably will be your last port of call for all problems Excel, VBA or otherwise. Millions of answers and discussions from the most mundane beginner's problems to the most technical and esoteric professionals' debate are available right here.

Start with standard Google search site and press 'Groups'. From the resultant web page select 'Advanced Groups Search'

Enter the salient word(s) to your problem or question in the relevant search boxes, and limit your answers to the Excel newsgroups by entering the phrase \*Excel\* (make sure you include the asterisks) in the 'Group' box. Finally press the 'Google Search button to view the results.

Find messages with **all of the words**  10 messages Sort by relevance

with the exact phrase

with at least one of the words

without the words

Group   (Examples: groups.google.com/group/Google-SMS, comp.os.7

Google Search

#### Pro-Tip

Microsoft operates a 'reward' system for suitably knowledgeable individuals who give up their own time to help others, known as MVP (Most Valuable Professionals). If you ever see these letters after the name of anyone in the Excel Newsgroups (or occasionally on a website or book), you can be sure to trust the information contained therein.

### **Excel Help & the Microsoft Website**

There is additional VBA help available from within the VBE. It is not particularly comprehensive (and the help is not always installed automatically) however pressing F1 whilst a keyword is selected can give useful help on syntax, plus often a simple example of how it is used. The Microsoft website is both a goldmine and minefield. It has answers to many technical questions (it certainly includes most of the numerous known features (aka bugs), but the search engine leaves much to be desired. Put aside plenty of time to trawl through half relevant information! The site also includes some good tutorials.

<http://support.microsoft.com/gp/hublist>

### **John Peltier - The graph guru!**

If it can be graphed in Excel ...this guy has done it. You will be in awe at what can be done with an Excel chart once you've reviewed this site. Some of his examples require VBA but many do not. If you want to know how to achieve the graph you have in the back of your mind, this is the ONE time that it is recommended that you use a specific website over and above any Google groups search.

<http://peltiertech.com/Excel/Charts/index.html>

### **John Walkenbach**

Author of the best Excel books available for the Advanced Excel user. The site lists his books and also includes some good developer tips.

<http://j-walk.com/ss/books/index.htm>

<http://j-walk.com/ss/excel/tips/index.htm>

### **Rob Bovey : XY Chart Labeller**

Making the label on a chart say something more useful than the point's value or series name can be really tricky. Rob's free program is probably the most useful Excel utility available anywhere.... and best of all, it's free.

Some other good free utilities (particular the VBA code cleaner) and references to other useful sites.

<http://www.appspro.com/>

## Designing for Bob

*'A light hearted look at some of the serious issues encountered when programming or project managing an Excel project for other people.'*

So far we have assumed that you will be undertaking the design and programming yourself in your own department and to your own specifications. In this chapter we will take a slightly less serious look at some of the very serious factors associated with designing systems for other team members, programming solutions for other people based on their requirements or managing programmers who are working to your specifications.

Your new Excel VBA skills will soon have you in great demand from your workmates and fellow professionals. If experience is anything to go by, then calls of "oh, that's useful, can I pinch that for my department?" or "...can you just send me over that workbook?" will soon be ringing in your ears. You are then just one step away from being volunteered into designing, writing or even project managing numerous workbooks, tools or even whole VBA systems.

Depending on your position, point of view, and general sense of karma you may see this as a great opportunity to help your colleagues, a way of getting into your bosses good books or a complete pain in the backside!

For that 1% of you tending towards the latter of those three thoughts ...here's a series of polite excuses that should avoid the need for you to undertake any work yourself that, um, a friend of a friend, says he might have heard sometime and may have achieved the desired result ... possibly.

"That's a really good idea ..." (smile and start on a positive note, but now gradually realise the vastness of their request, and slowly change your face to one of concern. Gently but audibly suck in a little air through your teeth) "but we really need to look at all the ..." (pick one or more from the following) "compatibility issues, maintenance difficulties, legacy problems, departmental anomalies, version control concerns, security implications, on-line help requirements, technical changes ... first". Use a different reason to delay the decision each time the issue comes up and start again at the beginning of the list if necessary ... don't worry they won't understand or even remember any of these new 'techie' phrases you've started using. Finally, for the most determined troublemakers, or if all else fails ...

*"Installing a similar tool in your department is a brilliant idea ... I know my boss was looking to save money and cross-charge out some of our department's time."*

... will ensure the topic is never brought up again in polite conversation!

Okay, so your boss has promised you a big pay rise and use of the swivel chair in team meetings from now on, and you've decided to take on Bob's project. Here are some considerations...

### You don't do Bob's job?

There is one difference that is more important than any other when designing, managing or writing code for other people, and that is that only *they* know exactly what they are thinking. You are probably in Bob's department, or at least have a fair idea of his work which is a big advantage.

But having an understanding of Bob's priorities and concerns is massively important to designing any Excel project, and can lead it down a completely different route. Bob's priorities could be as diverse as accuracy, timeline, presentation, ease of use, calculation speed, intuitiveness or perhaps the cost of the solution, but don't assume ... ask him.

#### Pro-Tip

NEVER assume you know Bob's priorities ... sit down and ask him directly... "What are your priorities for this project?"

Before you follow the nine point plan, there also other consideration to give when designing for other people or departments ...

### Understand the problem

Sit down with Bob and discuss exactly what the project is designed to achieve and what ideas he already has for the tool and how it will work. Don't be afraid to challenge the project limitations or decrease the scope of the system if you feel any of his ideas are unrealistic or unworthy. Don't be afraid to keep going back to Bob to clarify points or discuss alternatives. This is all time well spent.

#### Pro-Tip

If you don't understand a point fully, go back and clarify it again and again and again if necessary. Nobody likes to feel stupid, but having to re-design your system because you misunderstood something makes you look REALLY daft!

### Set realistic expectations

- You can't guarantee the work will be bug free.
- Your solution probably will not work in 5 years time unless it has been maintained.
- You may be off sick or be called upon to cover a colleague which extends the project timeline.
- There may be an emergency in your own job that requires immediate attention and which takes precedence over Bob's project.
- There may be unexpected issues along the way that extend the project's timeline or may even mean any deadline could be missed.

You know these things... you're a programmer and project manager, but Bob may not. Setting realistic expectation and explaining potential problems and their consequences BEFORE you undertake or at least start the project is prudent and usually very much appreciated.

### **Design for your audience**

You probably won't be using the tool but who will? Are they technical or not? Should you conceal the workings of your tool and protect the data or let the user make changes?

Always consider the end user(s) whilst designing your solution. If it's not Bob, don't be afraid to schedule 10 minutes with the actual users to get their input ... you'll be surprised how good they are!

#### **Pro-Tip**

Discussing the project with the end-user and taking their feedback on board at the design stage helps with their buy-in of the project - A key factor in any project's success.

It has the added advantage that the end users are happier to help with the testing of *their* project.

### **Explain your proposed solution**

Once you have a clear idea in your head what is required and how it is going to be achieved, sit down with Bob and discuss your proposed solution. Sometimes it is good to have two slightly differing proposals and discuss the pros and cons of each. Be sure to agree the life expectancy and any limitations the project may have, such as a constant data structure or fixed product list, and emphasise the difficulties of changing the solution once development is underway.

### **Get help testing your solution**

Make it clear to Bob you will not simply be handing over a finished project. It is categorically his responsibility to accept the solution, and to that end it is his responsibility to test that the product meets the criteria specified and his expectations. Arrange specific times when Bob and/or the users can see how things are going and comment on progress. With a small project this would probably be right at the end, but in larger projects organise one or two feedback and test sessions at appropriate points. It is perfectly acceptable to tell users at these sessions that certain aspects are not finished and just to comment/test on particular parts of the project.

### **Deadlines (and REAL deadlines)**

If the project needs to be completed by a specific deadline discuss this in detail with Bob. Ascertain whether this is a 'comfort' deadline (one arbitrarily invented by Bob) or a genuine deadline (for example a salary payment), and talk about the consequences of not meeting the deadline. If the system is critical discuss a 'Plan B' and under what circumstances it would be invoked.

## Get your project signed off

Once your project is finished, tested, Bob is happy, and all the users have been instructed on its use, it is important for YOU that you formally hand over the project and all its associated responsibilities. This will save you the hassle of help and support calls but most importantly requests to change the tool which will all be requested as if you are a 24hr support line with nothing to do except a responsibility to sort their problems out immediately. With small projects or where you are Bob's friend or close colleague nothing more formal than something like "right Bob, that's it from me ... if you have any problems or issues jot them down and we'll discuss it at our weekly team meeting." will probably suffice, however, if your time has been budgeted or cross charged (or if you're expecting Bob to come back with lots of new requests disguised as bugs) you may want to get a formal email or written sign off.

### Other issues:

#### Screen size:

Make sure the user(s) are all working with the same screen resolution. If necessary develop your solution in the lowest screen resolution viewed by the users.

#### Excel Version:

Check you and all expected users are all using the same version of Excel. If versions differ ideally develop using the earliest version but remember to test your solution in all versions.

#### Macro Security:

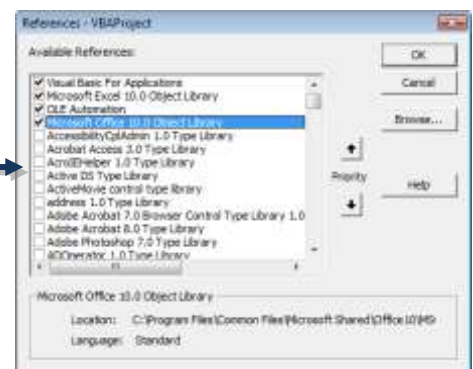
Check all users know about Macro security, and are able to set their computers to the correct security level (usually medium).

#### Missing References: (Can't find project or library)

Occasionally your code will fail unexpectedly on other people's computers. The failure will usually happen straight away and the message will pertain to Excel not being able to find the project or library. Follow these steps to fix the problem.

From the VBE (Tool / Macros / Visual Basic Editor) select References from the Tools menu. A userform similar to that shown below should be visible, containing several 'ticked' items.

If any items are ticked are stated as 'UNAVAILABLE' then de-click that item and save / re-open the workbook.



Note that these problems are most likely to occur with different Excel versions or machine builds to the one on which your system was developed. For the sake of stability always try to use the same system build as the expected end users.

## **The many faces of Bob**

There are many, many different Bobs for whom you may design a small worksheet or even program a large system. Here are some of the more common Bobs with their little foibles. Most of these Bobs aren't bad Bobs at all, and with a little love, can become your best buddy Bobs.

### **The 'not-so-bad' Bobs**

#### **The Bob that knows exactly what he wants ...NOT!**

Otherwise known as fag-packet-sketch Bob - Bob will invariably describe his requirement like this ..."I want something to produce the monthly report" or "I want something to calculate gross margins" ... at which time, in his ideal world you will take over the project and deliver it to him, completed, just before lunch! This Bob is not really a bad Bob. He probably has little or no experience with system or workbook design and certainly doesn't know about accurately specifying requirements as a pre-requisite to any project. A little education on the reasons why you need an accurate description of what is required and how he expects it to work in exchange for you designing his system is usually enough to send this Bob scurrying to his scribble pad. If he doesn't, see "The Bob that is only interested when you have the solution" .... a far more problematical Bob.

#### **The Bob that has told you everything you need to know ...NOT!**

Bob knows what he wants ...sort of! He just isn't willing or able to communicate it to you. He either sees this as a 'test' for you to come up with exactly the same solution as him, or a way of extracting ideas from you on other ways in which the project may be accomplished, or really does think you CAN read his mind.

If you think Bob isn't communicating enough of his thoughts to you, a meeting is usually enough to settle things down. Start by asking him exactly what he thinks the program should do and how he thinks it should be designed or featured. Gently question his motives or ideas with enquiries such as "And that is definitely the best way to calculate that, yes?" or "Have you had the chance to consider what will happen when ...." It is important you do NOT challenge or belittle Bob. The idea is to get a broad idea of how much thought he has given to the process, what his skill levels are in understanding the requirements and more about his hidden' expectations ...those project expectations that he has in his head that he isn't readily willing to share. If the meeting goes well you should be able discuss a solution upon which you both agree that will meet all of Bob's un-communicated requirements. With practice you'll be able to impose any changes or limitations and get Bob to think that they were his idea!

#### **The Bob that keeps changing his mind**

Bob will change his mind. This is a fact of life and a reality that we need to deal with if we are ever to write a system for any sort of Bob. So once we've accepted that Bob is going to change his mind what we really need to ask is how can we deal with it and how much is change is acceptable. The "No, no, no" attitude adopted by many professionals is not conducive to good relations. Instead inform Bob of the consequences of the changes honestly and thoroughly (don't be afraid to tell him you need to think about it first). Once Bob understands that the change will add half a day, half a week or half a million pounds to the project, it then becomes his decision .... this approach soon sorts out the 'must have' changes from the 'nice to have' changes.

### **The Bob that keeps finding bugs in your solution (changing the requirements)**

Bob accepted your solution and it was signed off right? So he has to take some responsibility if he later finds bugs in your solution. In reality it is IMPOSSIBLE to create a bug free solution, but a far more common issue is the change / new requirement disguised as a bug. More than likely Bob has changed something along the way (for example a different data structure or added a new product) then when your solution doesn't work as expected tells you that there is a bug. Ascertain the cause of the problem which (if you have properly tested your solution) will either be a new variation of something that could have been expected ( point out that maybe you AND Bob should have anticipated this, then fix the problem) or it is a change disguised as a bug. If the latter discuss with Bob that this is effectively a new requirement, and then it is up to you how you react to HIS problem.

### **The Bob with unrealistic expectations**

Setting realistic expectations is probably the most important aspect of your communication to Bob that you will need to impress upon him, all throughout, but particularly at the start of the project. There are many expectations in any project. The two primary and obvious expectations are time and features, but you should invest time discussing the more 'hidden' expectations that he may have, such as how easy it will to change a calculation or add a new feature. Many professional analysts and programmers advocate telling their customers that everything is complex, difficult and will take a long time (especially when it comes to changes) however being honest, conservative, open, thorough and realistic right the way through the project is usually a good option. Inform Bob about the potential consequences of any design change and make sure he understands this fact. Discuss the consequences of non-delivery or late delivery before you start, and always try to build in a contingency or alternative plan.

Never, EVER be tempted to agree to Bob's deadline simply because that is the deadline that Bob requires.

### **The Bob that JUST wants to .....**

When you are talking to Bob, 'JUST' is always a bad word. It might be ...

I JUST want to add ....

I JUST want to change ....

I JUST want it to work!

Bob uses the word JUST because Bob thinks the problem is simple. Maybe Bob is right and his (invariably) new requirement IS simple, but Bob doesn't have the knowledge or experience to know that. If Bob starts using this word often it is definitely time to explain to him that there is no such word as 'JUST' in computing. Everything needs to be considered with due regard to the implications or impact on all the other parts of the system. Tell him the every time he uses that word in your presence he'll have to put a fiver in your 'Just box' ... that should do the trick!

**The Bob that thinks your solution will work forever**

By now you should know the importance of predicting (or in this case agreeing) the life expectancy of your solution in order to help you make value judgements about how the solution is best designed, what changes you need to take into account and what value for time/money your solution represents. It is critical that you discuss with Bob all the critical assumptions you will be making in your code BEFORE you start work on them (If you are unfortunate enough to work in a blame culture environment we suggest you do this in writing) Examples of critical assumptions might be data structure, maximum number of data items, Excel version used or the fact that will only be used by one person at a time .... and get Bob to agree with these assumptions. If any of these mini-agreements are broken it is important that Bob knows the solution may or may not work. And if it doesn't there is no telling how long or even IF it can be fixed. Bad luck Bob.

**The Bob that thinks you'll always be there to help him**

The reality is that you won't be, even if you wanted to. That is why it is important to hand over your solution to Bob at the end of the project and it has been fully tested or first used. Give him as much time and advice as is required or is possible, but it is always wise to *categorically* finish and hand over your project. It may be necessary to write documentation to hand over with the project so at least part of the knowledge can be passed on at any time in the future. If necessary inform Bob that whilst you may be able to assist if he has any problems in the future there is absolutely no guarantee of this and it would only ever be at your or your department's convenience and that your number has been changed to a premium rate line at £8.50 minute to stop nonsense time wasting technical support callers.

**And finally, two genuinely troublesome Bobs ....****The Bob that is only interested in getting the solution.**

We'll keep this one simple. You will need Bob's time throughout the process. For discussion, agreement, advice, or at the very least just the plain courtesy of communication. If Bob has no interest in how you are doing or won't make time to discuss the project, the project is heading for trouble. No exceptions.

**The Bob that would do it himself, but just hasn't got the time.**

Beware the Bob that would do it himself, but either hasn't got the time or better still tells you that it's more efficient if you do it because you've already done something similar. The truth is that Bob couldn't do it himself. In reality Bob might be able to have a go, but the result would be so ponderous, unprofessional and unmanageable it would be dumped as soon as his manager's back was turned. Tell Bob in no uncertain terms that if he 'can' do it himself, it will definitely be quicker for him and better for all concerned if he does exactly that. (And that is a TRUE statement) - If he hasn't got time to undertake the project, he won't have time to consider it, plan it, discuss it, specify it, answer questions upon it, or implement it... and whatever you design for him, you can bet as soon as there's any sort of problem, Bob would tell you how he would have done it differently to you ... and HIS solution would have worked absolutely perfectly!

## Here endeth the workshop...

As a last thought, good programmers often need to think outside the box to solve complex problems, and the very best still manage to keep their solutions simple.

We leave you with words from Noah to his animals, as the waters receded;



### **"Go forth and multiply."**


And the ark emptied, except for two snakes that stayed behind. When Noah asked them why they had disobeyed him, they replied, "We can't multiply. We're adders."

Noah, immediately cut away part of the foredeck of his wooden vessel and built a small table with the lumber therefrom.

And they saw that it was good.

And Noah picked up the snakes and placed them upon it.

And Noah and the adders and the rest of the animals were overjoyed as all the creatures knew that even adders can multiply on a log table.



## Addendum: Project

### The Havemore Fruit Company Project

Your employer, The Havemore Fruit Company, buys and sells fruit. Eight varieties of apple, pear and strawberry are sourced from four different suppliers on a daily best price basis. The price for each fruit is set weekly for its customers who telephone their orders in to the company's sales staff and often place multiple orders daily.

An Excel data workbook that details all the company sales for the previous week is emailed to you from the sales department every Monday. The workbook always has the same structure and column headers and has a consistent file naming convention. (yymmdd).

You boss has tasked you to automate the weekly report and analysis that you have already created and which you currently update manually each week. Two weekly data files are available for testing WeeklyData040202.xls and WeeklyData040209.xls

You have agreed the following assumptions and limitations with your boss ..

- You will always get 5 days of sales, starting on a Monday
- The list of products, advisors, customers & suppliers will not change
- The varieties may change