

Table of Contents

Chapter 1.....	1
Introduction	2
The Macro Recorder.....	2
The Visual Basic Editor	4
The VBAProject Window	5
The Properties Window (Sheet1)	6
The Properties Window (Module1)	7
The Code Window	8
Modifying Macros	9
Creating Macros with VBA for Excel	12
Testing Macros	14
Chapter 2.....	16
Introduction.....	17
Variable Data Types.....	17
Why we use variables.....	19
Declaring Variables.....	20
Excel VBA Variables Lifetime & Scope.....	22
Procedure-Level Variables.....	23
Module-Level Variables.....	23
Project-Level, Workbook Level, or Public Module-Level.....	23
Modules and Procedures.....	23
Modules and Procedures and Their Scope.....	23
Calling Sub Procedures and Function Procedures.....	24
Passing Argument by Reference or by Value	25
Chapter 3.....	27
Message Box.....	28
InputBox.....	28
Application.InputBox.....	30
Chapter 4.....	33
Why Loops?.....	34
Loop Structures.....	34
For ... Next	34
For ... Next Loop With Step.....	35
Do While ... Loop.....	36
Do Until ... Loop	36
Do ... Loop While	36
Do ... Loop Until.....	37
Conditional Statements.....	37
If..Then...End If.....	37
If..Then...End If (multiple tiers).....	37
If..Then...And...End If.....	38
If..Then...Or...End If.....	38
If..Then...Else...End If.....	38
If..Then...ElseIf...End If.....	38

Select Case.....	39
Chapter 5.....	40
Introduction.....	41
Collections	42
Referencing Items in A Collection.....	42
Workbook and Worksheet Object.....	42
Range Object and Cells Property.....	44
Referencing Ranges in Other Sheets.....	47
Working with cells	48
Offset property.....	48
Using the CurrentRegion Property to Quickly Select a Data Range.....	49
Using the SpecialCells Method to Select Specific Cells.....	50
Methods and Properties.....	51
Chapter 6.....	53
Introduction	54
The On Error Statement.....	54
Debug.Print.....	56
Break Points.....	57
Stepping Through Code.....	57
Locals Window.....	58
Watch Window.....	58
Chapter 7.....	60
Beginning VBA: Events.....	61
Where To Put Event Code.....	61
Events.....	61
Workbook_Open.....	61
Worksheet_Change.....	61
Worksheet_SelectionChange.....	61
Automatically Run Excel Macros Upon Opening Excel Workbook/Files. Auto Run Excel VBA	
Macro Code.....	62
Auto Run Excel Macros Upon Open.	62
Workbook_Open Event.....	62
Auto Run Excel Macros Upon Data Entry.	64
Worksheet_Change Event.....	64
Monitor More Than 1 Cell (Target).....	65
Example	66
Chapter 8.....	67
Creating an UserForm.....	68
The Course Booking Form.....	68
Building the Form.....	70
Adding the Code: 1 Initialising the Form.....	70
Adding the Code: 2 Making the Buttons Work.....	72
Adding the Code 3: Manipulating the Form.....	75
Opening the Form.....	76

Macros

Introduction

Macros are a set of actions that u can record. Macros can speed up any common editing sequence you may execute in an Excel spreadsheet. To create macros you should know VBA. VBA is "Visual Basic for Application". It is a programming language that allows users to program macros to accomplish complex tasks within an application. With VBA for Excel (Visual Basic for Application) you can develop small procedures (macros) that will make your professional life easier and allow you to do more in less time. But VBA is also a very powerful programming language with which you can develop within Excel real programs that will accomplish in a few minutes very complex tasks. With VBA for Excel you can develop a program that does EXACTLY what you need and nothing else.

Anybody can develop simple macros (VBA procedures) and with time and interest you can get to a point where you can develop very complex procedures to accomplish very sophisticated tasks

The best way to learn macros is to record it and then view the code.

The Macro Recorder

With the Excel macro recorder you cannot develop a macro that will damage Excel or your computer. The bolder you are in your trials the more you will learn.

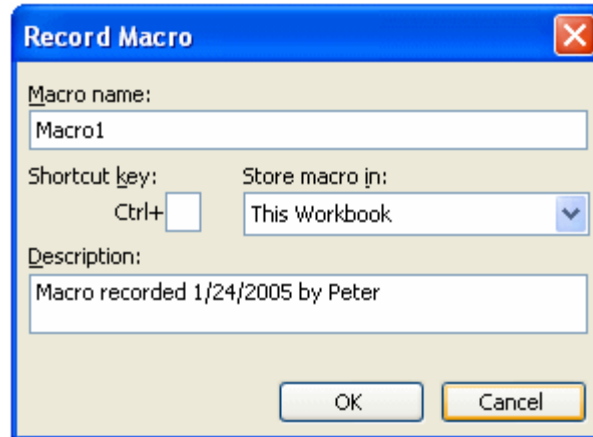
To record macro follow the instructions step by step.

First make sure that the macros are nor deactivated in Excel. Go to the menu bar "Tools/Macro/Security". Check "Medium". Close Excel without saving the active workbook and re-open it. From now on every time you open a spreadsheet that contains macros you will be asked to "Enable" or "Disable" the macros. If you choose "Disable" you can open the spreadsheet but the macros won't work. If you don't know who is sending the spreadsheet select "Disable".

1- Open Excel.

2- On the menu bar go to "Tools/Macro/Record New macro"

In the following window you see that a macro called "Macro1" will be created. You can change the name if you want. In "Store macro in:" select "This Workbook". When you click on "OK" Excel will record a macro named as you specified and save it in the workbook that you are working in. The macro can run in any workbook but only if this workbook is open. The macro will be saved with this workbook when you save and close it. It also means that Macro1 will die when you delete the workbook. When you go to the Visual Basic Editor you will also see that "Macro1" is stored in "Module1".



3- Click on "OK". A small icon might appear in the middle of the screen, forget about it.

4- Let's now record a macro with the Excel macro recorder:

Select cell A1 and enter 34

Select cell A2 and enter 55

Select cell A3 and enter the formula $=A1 + A2$

Select cell A2 and change the color of the font to red

Select cell A1 and change the background color to blue

Select cell A3 and change the size of the font to 24

5- Go to the menu bar "Tools/Macro/Stop Recording"

The macro has been recorded by the Excel macro recorder now let's test it

6- Go to Sheet2

7- Go to the menu bar "Tools/Macro/Macros..." select "Macro1" in the list and click on "Run". The macro is executed.

8- Save your workbook as "VBATest1" .


You have recorded your first macro with the Excel macro recorder and used it, Now you will be able to look at the macro that you have recorded and then create your own macros.

The Visual Basic Editor

You develop VBA procedures (macros) in the Visual Basic Editor (VBE). It is a marvelous user friendly development environment. The VBA procedures developed in the VBE become part of the workbook in which they are developed and when the workbook is saved the VBA components (macros, modules, userforms) are saved at the same time.

The VBE is integrated into Excel and you can open it from the Excel Menu Bar "Tools/Macro/Visual Basic Editor" or click on the icon on the Visual Basic ToolBar

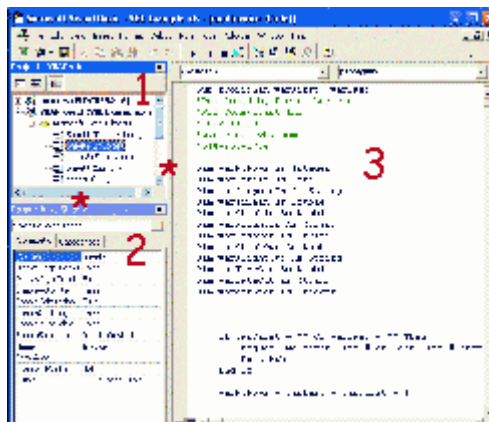


in Excel. From the VBE you can go to Excel by clicking on the Excel button at the top/left of your screen . So if the VBA toolbar is visible in Excel

(View/Toolbar/VBA) you can navigate from VBE to Excel using the two buttons 



Now open the workbook that you have created "VBATest1.xls" and open the VBE.



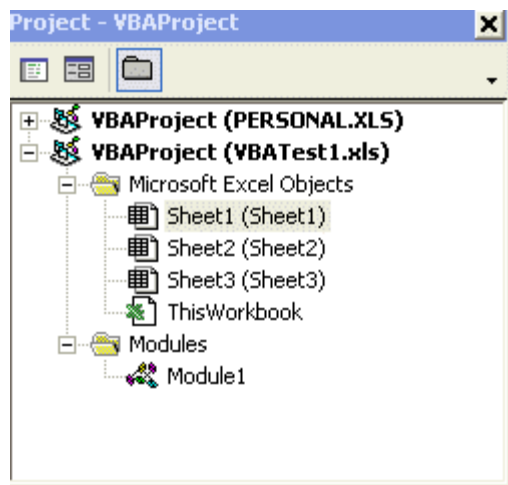
The Visual Basic Editor

When you work with the VBE, there is always 3 windows that are open. The VBAProject window (1), the Property window (2) and the Code window (3).

On the menu bar of the VBE choose "View" and select "Project Explorer" then go back to the menu bar and select "Properties Window". You don't call the Procedure window from the menu bar, it is there and contains the VBA code for the element that you have chosen in the "VBAProject" window by double clicking on it (sheet, module, form).

You can resize the windows by placing the cursor over the borders (*) and drag them right, left, up or down.

The VBAProject Window

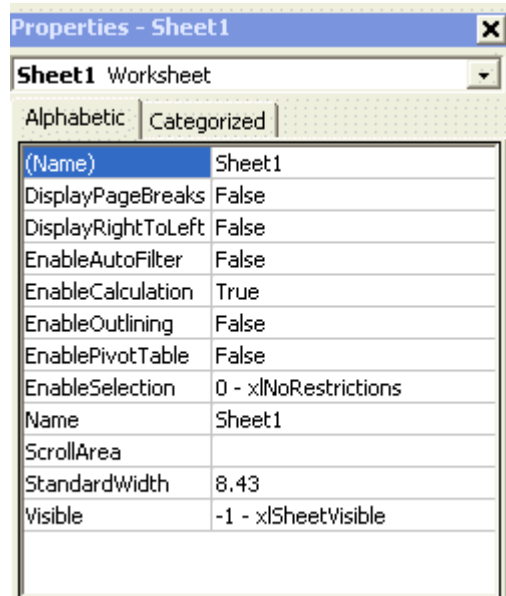


The VBAProject window shows you all the workbooks that are open. You can add forms and modules to any workbook by right clicking anywhere within one of the workbook and select "Insert" in the contextual menu.

On your screen you might not see the "PERSONAL.XLS" workbook. IF you click on the + sign besides "VBAProject(VBATest1.xls)" you see that your "VBATest1.xls" has three sheets and one module plus the "ThisWorkbook" object. If you double click on any of the elements the code window will show you the code developed within this element. The properties window will show you the properties of the element that you have selected in the project window. click on Sheet1.

If you have more than one workbook open you can copy a module from one workbook to the other by simply left clicking, holding and dragging.

The Properties Window (Sheet1)



The Properties window shows you the properties of the element that is selected in the VBAProject Window. Double click on "Sheet1" in the project window and the properties windows on your screen will look as the one on the left.

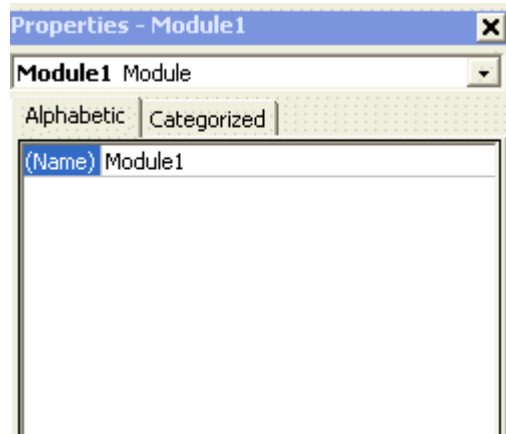
You can modify any of the properties from there. The list of properties changes as you select different types of components (thisWorkbook, worksheets, modules, forms or controls).

Note that there are two "Name" properties with and without parenthesis. The name property "(Name)" on the first line is the programmatical name of the sheet. The other name property "Name" on the 9th line is the caption the name of the sheet that the user sees and that you use in your Excel formula. You can either change it here or in Excel.

Do the following changes and see what happens in the VBAProject window. Change the name on the 9th line to "Info" and go to Excel to see that the caption of the sheet has changed. Change it back to "Sheet1". Now change the name on the first line to "shInfo" and see in Excel that the caption has not changed. Change it back to Sheet1.

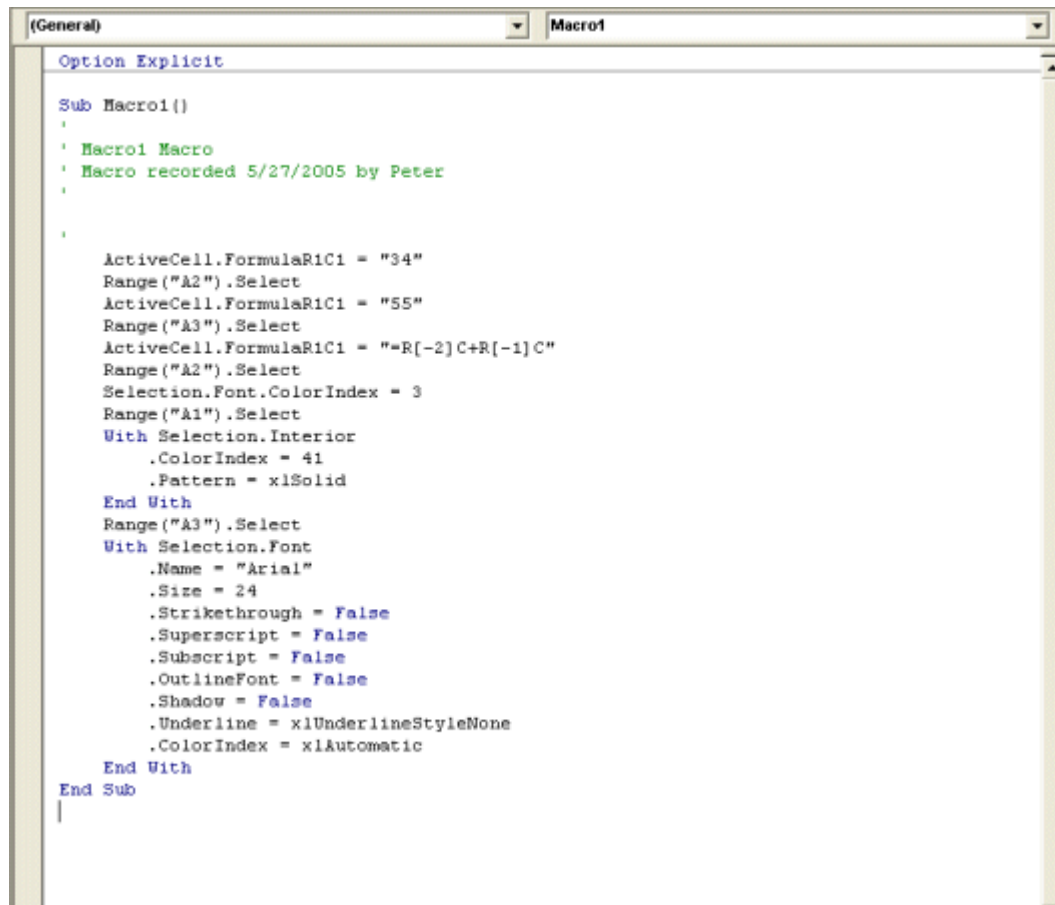
On line one give a significant name to your sheets starting with the prefix "sh" and a first capital letter for any significant word comprised in the name like shInfo, shRawData, shWhatever. You will see later how important becomes this prefix and capital letter habit.

The Properties Window (Module1)



Double click on Module1 in the VBAProject window and your property window will look like this one. There is only one property, the name. Please name your modules. It is so much friendlier to see modImport, modDatabase, modExportData, modWhatever. Give a significant name to your module starting with the prefix "mod " and a first capital letter for any significant word comprised in the name. You will see later how important becomes this prefix and capital letter habit.

The Code Window



In the VBAProject window double click on Module1 and the Code window above will appear. In this code window you can see the procedure that you have recorded called "Macro1".

Below "End Sub" enter "Sub Macro2()" and click enter. VBE will add a line "End Sub". Now go to the list box (top/right of the Code window) and click on the arrow. You see that you can navigate from one procedure to the other by selecting them in this list. Erase what you have just added (Macro2) and see that it has also disappeared from the list box.

Very rarely you develop procedures that are triggered by an event related to the worksheet or the workbook. But let's take a quick look at the topic.

In the VBAProject window double click on Sheet1. The code window is now empty because you have not developed procedures that are triggered by an event related to the sheet (activate, deactivate, etc...). In the top/left list box (General) select "Worksheet". Automatically VBE creates a procedure called "Worksheet_SelectionChange(ByVal Target As Range)". Now if you go to the top/right list box you can see all the events relating to the sheet. Select "Activate". VBE creates a procedure called "Sub Worksheet_Activate()". Here you could

develop the procedure so that something happens when you activate the sheet. Erase everything in the code window.

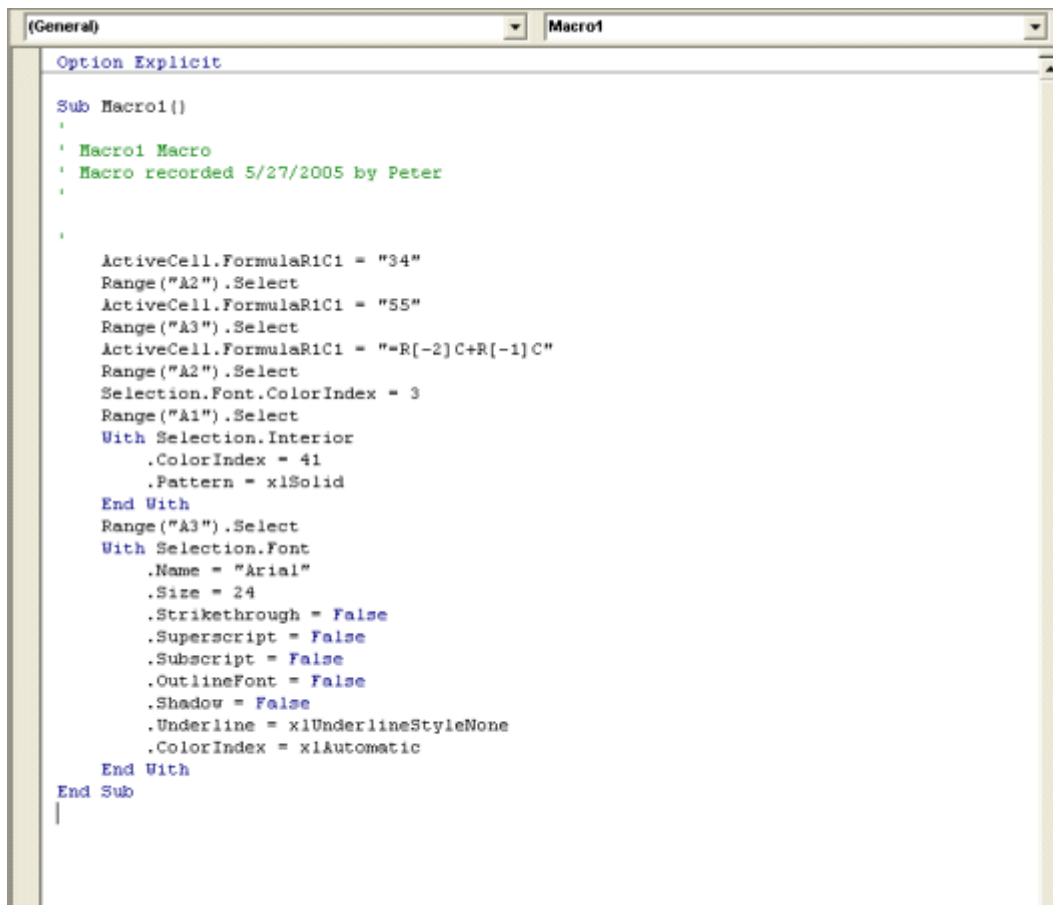
In the VBAProject window double click on "ThisWorkbook". The code window is now empty because you have not developed procedures that are triggered by an event related to the workbook (Open, BeforeClose, etc...). In the top/left list box (General) select "Workbook". Automatically VBE creates a procedure called "Workbook_Open()". Now if you go to the top/right list box you can see all the events relating to the workbook. Within the "Workbook_Open()" you could develop a procedure so that something happens when you open the workbook. Erase everything in the code window.

Modifying Macros

Modifying a Macro

Modify the VBA procedure that you have created with the recorder earlier. Open Excel, open the workbook "VBATest1.xls" and go to the Visual Basic Editor. Double click on Module1 in the VBA project window and the following code will appear in the Code window.

Click in the code window and print the module for future reference
"File/Print/Current Module"



Follow the instructions step by step.

The Macro Recorder (MR) has a very special way to write code. The procedure above works so we could leave it alone but just to make it clearer let's make it simpler. You can replace each line of code yourself or copy/paste the entire new procedure. When we started the Macro Recorder cell A1 was selected and you entered 34 in it. The MR writes

ActiveCell.FormulaR1C1="34"

You will never use the FormulaR1C1 property. Secondly we will add a line of code to tell Excel to start this procedure in cell A1 otherwise, 34 will be entered in the cell that is selected when we start the procedure Cell A1 is the ActiveCell and you can replace this line simply by:

Range("A1").Select
ActiveCell.Value=34

Line 2 and 3 of the procedure read like this:

Range("A2").Select
ActiveCell.FormulaR1C1 = "55"

Again we will avoid the FormulaR1C1 thing and we won't select the cell we will just

give it a value so replace lines 2 and 3 by this:

```
Range("A2").Value=55
```

Line 4 and 5 of the procedure read like this:

```
Range("A3").Select
```

```
ActiveCell.FormulaR1C1 = "=R[-2]C+R[-1]C"
```

Again we will avoid the FormulaR1C1 but we will not use Value because we want to enter a formula in this cell. Again we wont select the cell we will just enter a formula so replace lines 4 and 5 by this:

```
Range("A3").Formula="=A1+A2"
```

Line 6 and 7 of the procedure read like this:

```
Range("A2").Select
```

```
Selection.Font.ColorIndex = 3
```

Here we wont select the cell we will just specify that we want to change the color of the font so replace lines 6 and 7 by this:

```
Range("A2").Font.ColorIndex = 3
```

Three is red. Use the MR in these situation because you don't have to remember all the color codes.

Line 8 to 12 of the procedure read like this:

```
Range("A1").Select
```

```
With Selection.Interior
```

```
    .ColorIndex = 41
```

```
    .Pattern = xlSolid
```

```
End With
```

The MR uses a lot of "With..End With". The code developed by the MR would read in plain English: Select cell A1 (cell A1 becoming the Selection) then for the Selection.Interior make the ColorIndex be 41 and the Pattern be xlSolid. Here is a simpler version so replace lines 8 to 12 by this:

```
Range("A1").Interior.ColorIndex = 41
```

```
Range("A1").Interior.Pattern = xlSolid
```

Line 13 to 25 of the procedure read like this:

```
Range("A3").Select
```

```
With Selection.Font
```

```
    .Name = "Arial"
```

```
    .Size = 24
```

```
    .Strikethrough = False
```

```
    .Superscript = False
```

```
    .Subscript = False
```

```
    .OutlineFont = False
```

```
    .Shadow = False
```

```
    .Underline = xlUnderlineStyleNone
```

.ColorIndex = xlAutomatic
End With

Again MR uses "With...End With" and also modifies ALL the properties of the font each time. We just want the font's size to be changed to 24 so replace lines 13 to 25 by this:

Range("A3").Font.Size= 24

The VBA procedure or macro called "Macro1" now looks like this:

Sub Macro1()

' Macro1 Macro

' Macro recorded 5/27/2005 by Peter'

ActiveCell.Value = 34
Range("A2").Value = 55
Range("A3").Formula = "=A1+A2"
Range("A2").Font.ColorIndex = 3
Range("A1").Interior.ColorIndex = 41
Range("A1").Interior.Pattern = xlSolid
Range("A3").Font.Size = 24

End Sub

Excel Color Constants:

Selection.Font.ColorIndex = 1' Black
Selection.Font.ColorIndex = 2' White
Selection.Font.ColorIndex = 3' Red
Selection.Font.ColorIndex = 4' Bright Green
Selection.Font.ColorIndex = 5' Blue
Selection.Font.ColorIndex = 6' Yellow
Selection.Font.ColorIndex = 7' Pink
Selection.Font.ColorIndex = 8' Bright Blue
Selection.Font.ColorIndex = 9' Brown
Selection.Font.ColorIndex = 11' Dark Blue
Selection.Font.ColorIndex = 13' Teal

Creating Macros with VBA for Excel

Let's now create a brand new macro. Open Excel, open the workbook "VBATest1.xls" that you have created previously. Add a sheet to your workbook and rename it "Test2" (right click on the tab, chose "Rename" and type the name

in). In cell A1 to A10 enter 10 first names and in cells B1 to B10 enter 10 last names.

Now let's create a new module. Go to the VBE and right click in the VBA project Window within the project "VBATest1" and select "Insert/Module". You now have a new module. In the properties window double click on the "(Name)" property box and enter "modTest". We now have a new module with a name.

In the VBAProject window double click on the sheet named "Test2". In the properties window double click on the "(Name)" property box and enter "shTest2". You now have a sheet with a caption reading "Test2" and with the name "shTest2".

In the VBAProject window double click on "modTest2" and let's go to the code window to develop a macro.

Write "Sub proTest2 ()" and click "Enter". VBE adds a line "End Sub". We will write our code between these two lines so make some room for it by setting the cursor at the end of "End Sub" and clicking "Enter" a few times.

Just below "Sub proTest2()" write your name preceded by an apostrophe (') ' Peter Clark and click "Enter". Notice that the font in the line you have just written is green. Because of the apostrophe VBA will consider the text as a remark (REM) and will not bother with it. You can add any number of REM lines anywhere to make your code understandable. Click "Enter" again to insert an empty line. Do not hesitate to insert empty lines anywhere to make your code more easily readable.

Here are the 6 lines of code that you will be writing (or copy/paste them from this webpage):

```
shTest2.Select  
Range("C1").Select  
  
Do Until Selection.Offset(0, -2).Value = ""  
    Selection.Value = Selection.Offset(0, -2).Value & " " &  
Selection.Offset(0, -1)  
    Selection.Offset(1, 0).Select  
    Loop  
  
Range("A1").Select
```

The first 2 lines tell VBA where to start the procedure. If you don't mention it VBA will start the procedure from whatever cell is selected in your workbook.

Line 5 to 7 is the actual task that you want VBA to perform. You are saying: do something until the value of the cells 2 columns left of the selected cell is empty (double double quotes). When a cell is selected after the line where is the last first name the task will end. If there is a first name you do else you loop (last line of the statement).

What is the task: make the value of the selected cell equal to the value of the cell 2 columns left and (&) a space (space between two double quotes) (" ") and the value of the cell 1 column left of the selected cell.

Then move down one cell.

When the task has been accomplished go to cell A1.

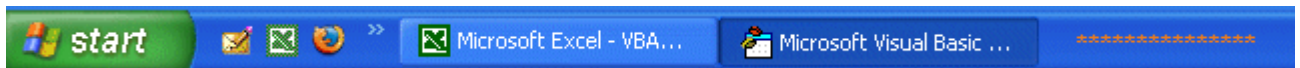
Now go back to Excel go to "Tools/Macro/Macros" select the macro "proTest2" and click "Run". Erase column C and do it again. Add first names and last names to your list and do it again.

You have created your first VBA procedure.

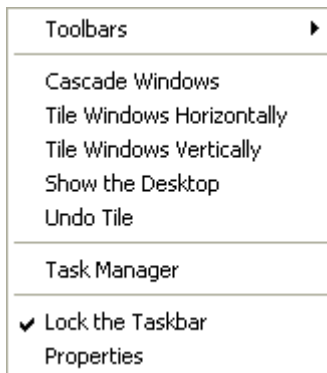
Testing Macros

Here is a very interesting way to test a macro step by step while seeing it at work in the Excel workbook.

Close every program on your computer. Open Excel and the workbook "VBATest1.xls" in which we have created a new macro called "proTest2". Open the VBE. On the Window status bar at the bottom of your screen you can see that Excel is open and the VBE also. Right click on the status bar in the empty space where have added the orange stars:



The following menu appears:



Click on "Tile Windows Vertically"

The Excel workbook occupies half of the screen and the VBE occupies the other half. Make the Code window wider in the VBE so that you can see all the code. Make sure to remove values and formats in cell C1 to C10 of the workbook.

In the Excel workbook select cell D3 of the sheet "Test2", and then select any sheet other than "Test2". In VBE, click anywhere within the procedure "proTest2". Click on the F8 key at the top of your keyboard and see that the first line of code become yellow. Click again on F8 and the sheet "Test2" is selected, click again and cell C1 is selected click a few more times until no line in the code is yellow. You have tested your code step by step.

Variables and Data Types

Introduction

A Variable is used to store temporary information that is used for execution within the Procedure, Module or Workbook. Before we go into some detail of Variables, there are a few important rules that you must know about.

- 1) A Variable name must start with a letter and not a number. Numbers can be included within the name, but not as the first character.
- 2) A Variable name can be no longer than 250 characters.
- 3) A Variable name cannot be the same as any one of Excel's key words. You cannot name a Variable with such names as Sheet, Worksheet etc.
- 4) All Variables must consist of one continuous string of characters only. You can separate words by either capitalising the first letter of each word, or by using the underscore characters if you prefer.

You can name variables with any valid name you wish. For Example you could name a variable "David" and then declare it as any one of the data types shown below. However, it is **good practice to formalize some sort of naming convention**. This way when reading back your code you can tell at a glance what data type the variable is. An example of this could be! If you were to declare a variable as a Boolean you may use: **bIsOpen**. You might then use this Boolean variable to check if a Workbook is open or not. The "b" stands for Boolean and the "IsOpen" will remind you that you are checking if something is open.

You may see code that uses letters only as variables, **this is bad programming and should be avoided**. Trying to read code that has loads of single letters only can (and usually does) cause grief.

Variable Data Types

Byte data type

A data type used to hold positive integer numbers ranging from 0 to 255. Byte variables are stored as single, unsigned 8-bit (1-byte) numbers.

Boolean data type

A data type with only two possible values, True (-1) or False (0). Boolean variables are stored as 16-bit (2-byte) numbers.

Integer data type

A data type that holds integer variables stored as 2-byte whole numbers in the range -32,768 to 32,767. The Integer data type is also used to represent enumerated values. The percent sign (%) type-declaration character represents an Integer in Visual Basic.

Long data type

A 4-byte integer ranging in value from -2,147,483,648 to 2,147,483,647. The ampersand (&) type-declaration character represents a Long in Visual Basic.

Currency data type

A data type with a range of -922,337,203,685,477.5808 to 922,337,203,685,477.5807. Use this data type for calculations involving money and for fixed-point calculations where accuracy is particularly important. The at sign (@) type-declaration character represents Currency in Visual Basic.

Single data type

A data type that stores single-precision floating-point variables as 32-bit (2-byte) floating-point numbers, ranging in value from -3.402823E38 to -1.401298E-45 for negative values, and 1.401298E-45 to 3.402823E38 for positive values. The exclamation point (!) type-declaration character represents a Single in Visual Basic.

Double data type

A data type that holds double-precision floating-point numbers as 64-bit numbers in the range -1.79769313486232E308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values. The number sign (#) type-declaration character represents the Double in Visual Basic.

Date data type

A data type used to store dates and times as a real number. Date variables are stored as 64-bit (8-byte) numbers. The value to the left of the decimal represents a date, and the value to the right of the decimal represents a time.

String data type

A data type consisting of a sequence of contiguous characters that represent the characters themselves rather than their numeric values. A String can include letters, numbers, spaces, and punctuation. The String data type can store fixed-length strings ranging in length from 0 to approximately 63K characters and dynamic strings ranging

in length from 0 to approximately 2 billion characters. The dollar sign (\$) type-declaration character represents a String in Visual Basic.

Object data type

A data type that represents any Object reference. Object variables are stored as 32-bit (4-byte) addresses that refer to objects. Variant data type A special data type that can contain numeric, string, or date data as well as the special values Empty and Null. The Variant data type has a numeric storage size of 16 bytes and can contain data up to the range of a Decimal, or a character storage size of 22 bytes (plus string length), and can store any character text. The VarType function defines how the data in a Variant is treated. All variables become Variant data types if not explicitly declared as some other data type.

Why we use variables

Excel will still allow us to run our code without using variables, it is not a must! But having said this it is very bad programming to not use variables. You could quite easily just assign a value, string or whatever each time you need it, but it would mean:

- 1) Your code would become hard to follow (even for yourself)
- 2) Excel would constantly need to look for the value elsewhere.
- 3) Editing your code would become awkward.

Let's use an example to highlight the above

Sub NoVariable()

```
Range("A1").Value = Range("B2").Value  
Range("A2").Value = Range("B2").Value * 2  
Range("A3").Value = Range("B2").Value * 4  
Range("B2").Value = Range("B2").Value * 5
```

End Sub

In the above code, Excel would need to retrieve the value from cell **B2 five times**. It would also mean if we had many other procedures using the same value i.e **B2**, it would need to retrieve it's value even more times.

There is a lot of editing to be done if we were to change from wanting **B2** value to say, **B5** value. It is messy code.

Let's now use a variable to store the value of cell **B2!**

Sub WithVariable()

```
Dim iMyValue as Integer
    iMyValue = Range("B2").Value
    Range("A1").Value = iMyValue
    Range("A2").Value = iMyValue * 2
    Range("A3").Value = iMyValue * 4
    Range("B2").Value = iMyValue * 5
```

End Sub

In the above code Excel only needs to retrieve the value of cell **B2 once**.

To edit our code we only need to change it in one place.

It is easier to read.

You might be thinking that there is no big difference in the above 2 examples, and to a point you would be correct. But what you must realize is, most VBA projects will consist of hundreds (if not thousands) of lines of code. They would also contain a lot more than one procedure. If you had 2 average size VBA projects, one using variables and one without, the one using variables would run far more efficiently!

Declaring Variables

To declare a variable we use the word "Dim" (short for Dimension) followed by our chosen variable name then the word "As" followed by the variable type. So a variable dimmed as a String could look like:

Dim sMyWord As String

You will notice that as soon as we type the word As, Excel will display a drop-down list of all variables.

The default value for any Numeric type Variable is zero.

The default value for any String type variable is "" (empty text).

The default value for an Object type Variable is Nothing. While the default value for an Object type Variable is Nothing, Excel will still reserve space in memory for it.

To assign a value to a Numeric or String type Variable, you simply use your Variable name, followed by the equals sign (=) and then the String or Numeric type. eg:

Sub ParseValue()

```
Dim sMyWord as String
Dim iMyNumber as Integer
    sMyWord = Range("A1").Text
    iMyNumber = Range("A1").Value
```

End Sub

To assign an Object to an Object type variable you must use the key word "Set".
eg:

Sub SetObject()

```
Dim rMyCell as Range
    Set rMyCell = Range("A1")
```

End Sub

In the example immediately above, we have set the Object variable to the range **A1**. So when we have finished using the Object Variable "rMyCell" it is a good idea to Set it back to it's default value of Nothing. eg:

Sub SetObjectBack()

```
Dim rMyCell as Range
    Set rMyCell = Range("A1")
```

```
    Set rMyCell = Nothing
```

End Sub

This will mean Excel will not be reserving unnecessary memory.

In the first example above (Sub ParseValue()) we used 2 lines to declare our 2 variables ie

Dim sMyWord as String**Dim iMyNumber as Integer**

We can, if we wish just use:

Dim sMyWord as String, iMyNumber as Integer

There is no big advantage to this, but you may find it easier.

Not Declaring Variables

There is a difference between using variables and correctly declaring them. You can if you wish not declare a variable and still use it to store a Value or Object. Unfortunately this comes at a price though! If you are using variables which have not been dimensioned Excel (by default) will store them as the Variant data type. This means that Excel will need to decide each time it (the variable) is assigned a value what data type it should be. The price for this is slower running of code! My advise is do it right and form the good habit early!

There is also another advantage to correctly declaring variables and that is Excel will constantly check to ensure you have spelt the variable name correctly. It does this by capitalizing the all lower case letters that are capitalized at the point it was dimensioned. Let's assume you use:

Dim iMyNumber As Integer

At the top of your procedure. You then intend to use this variable in other parts of the procedure. Each time you type **imynumber** and then push the Space bar or Enter Excel will capitalize it for you i.e **imynumber** will become **iMyNumber**. This is a very simple and easy way to ensure you have used the correct spelling.

While we are on this subject, it is very good practice to type all code in lower case, because not only will Excel do this for variables but also for all Keywords!

There may be times when you will actually need to use a Variant data type as you cannot be certain what will be parsed to it, say from a cell. It might be text, it maybe a very low or high number etc. In these circumstances you can use:

Dim vUnKnown As Variant

Or, simply:

Dim vUnKnown

Both are quite valid! The reason we do not have to explicitly declare a Variant is because the default for a variable is a Variant.

Excel VBA Variables Lifetime & Scope

In Excel, when coding in VBA, we can use what are know as variables to store information. These variables (as the name suggests) can be varied and changed to store different data information. As soon as a variable loses scope it loses its stored value.

Excel VBA Variables Levels

There are 3 levels at which we can dimension (Dim) variables. These are;

- 1) Procedure-Level
- 2) Module-Level
- 3) Project-Level, Workbook Level, or Public Module-Level

Each of these levels differ in scope and lifetime. This is discussed below

Procedure-Level Variables

These are probably the best known and widely used variables. They are dimensioned (Dim) inside the Procedure itself. See Example below;

```
Sub MyMacro ()  
Dim IRows as Long  
    'Code Here  
End Sub
```

All variables dimensioned at this level are only available to the Procedure that they are within. As soon as the Procedure finishes, the variable is destroyed.

Module-Level Variables

These are variables that are dimensioned (Dim) outside the Procedure itself at the very top of any Private or Public Module. See Example below;

```
Dim IRows as Long
```

```
Sub MyMacro ()  
    'Code Here  
End Sub
```

All variables dimensioned at this level are available to all Procedures that they are within the Module it is dimensioned in. It's value is retained unless the variable is referenced outside its scope, the Workbook closes or the **End Statement** is used.

Project-Level, Workbook Level, or Public Module-Level

These variables are dimensioned at the top of any standard public module, like shown below;

```
Public IRows as Long
```

All variables dimensioned at this level are available to all Procedures in all Modules. It's value is retained unless the Workbook closes or the **End Statement** is used.

Modules and Procedures

Modules and Procedures and Their Scope

A **module** is a container for procedures as shown in our prior examples. A **procedure** is a unit of code enclosed either between the **Sub** and **End Sub** statement or between the **Function** and **End Function** statements.

The following sub procedure (or sub routine) print the current date and time on cell C1:

```
Sub ShowTime()  
    Range("C1") = Now()  
End Sub
```

The following function sum up two numbers:

```
Function sumNo(x, y)  
    sumNo = x + y  
End Function
```

Procedures in Visual Basic can have either private or public scope. A procedure with private scope is only accessible to the other procedures in the same module; a procedure with public scope is accessible to all procedures in every module in the workbook in which the procedure is declared, and in all workbooks that contain a reference to that workbook. By default, procedures has public scope.

Here are examples of defining the scope for procedure.

```
Public Sub ShowTime()  
    Range("C1") = Now()  
End Sub
```

```
Private Sub ShowTime()  
    Range("C1") = Now()  
End Sub
```

Calling Sub Procedures and Function Procedures

There are two ways to call a sub procedure. The following example shows how a sub procedure can be called by other sub procedures.

```
Sub z(a)  
    MsgBox a  
End Sub
```

```
Sub x()  
    Call z("ABC")  
End Sub
```

```
Sub y()  
    z "ABC"  
End Sub
```

Sub z procedure takes an argument (a) and display the argument value ("ABC") in a message box. Running either Sub x or Sub y will yield the same result.

The following example calls a function procedure from a sub procedure.

```
Sub ShowSum()  
    msgbox sumNo(3,5)  
End Sub
```

```
Function sumNo(x, y)  
    sumNo = x + y  
End Function
```

The ShowSum sub procedure calls the sumNo function and returns an "8" in a message box.

If there are procedures with duplicate names in different modules, you must need to include a module qualifier before the procedure name when calling the procedure.

For example:

Module1.ShowSum

Passing Argument by Reference or by Value

If you pass an argument **by reference** when calling a procedure, the procedure access to the actual variable in memory. As a result, the variable's value can be changed by the procedure. Passing by reference is the default in VBA. If you do not explicitly specify to pass an argument **by value**, VBA will pass it by reference. The following two statements yield the same outcome.

```
Sub AddNo(ByRef x as integer)  
Sub AddNo(x as integer)
```

Here is an example to show the by reference behavior. The sub procedure, TestPassing1 calls AddNo1 by reference and display "60" (50 + 10) on the message box.

```
Sub TestPassing1()  
    Dim y As Integer  
    y = 50  
    AddNo1 y  
    MsgBox y  
End Sub
```

```
Sub AddNo1(ByRef x As Integer)  
    x = x + 10  
End Sub
```

The following example shows the by value behavior. The sub procedure, TestPassing 2 calls AddNo2 by value and display "50" on the message box.

```
Sub TestPassing2()  
    Dim y As Integer  
    y = 50  
    AddNo2 y  
    MsgBox y  
End Sub
```

```
Sub AddNo2(ByVal x As Integer)  
    x = x + 10  
End Sub
```

Message Box and Input Box

Message Box

The message box is the primary tool to interact with the user. You can use it to inform or alert him. Few types of message boxes are: the basic MsgBox (vbOKOnly), the 3 button one (vbYesNoCancel) and the specialized one (vbDefaultButton..).



To generate the message box above, the code is:
`MsgBox "Your message here"`

When VBA procedures take time to run, offer a message box to user at the end of the execution.

`MsgBox "The procedure has been executed."`

You want to send the following message to your user Go to cell "A1". with double quotes around the A1. Double the quotes within the main quotes.

`MsgBox "Go to cell ""A1""."`

You can insert MsgBox within your code and run it up to the MsgBox to test it. The syntax is then:

`MsgBox "TestOK"`

InputBox

Collect User Data/Input

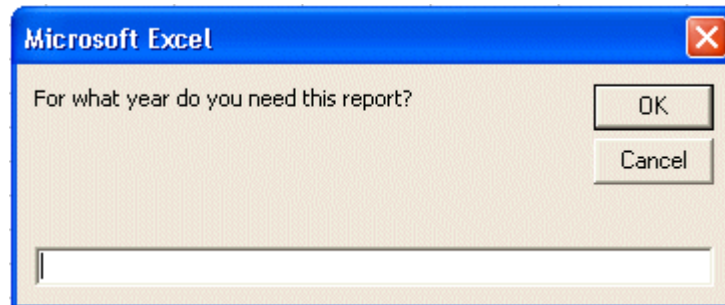
There are many times in Excel VBA that we are required to gather information from a user. To actually allow the user to enter some text, number or even a formula we can use the InputBox Function. The syntax for the InputBox Function is;

`InputBox(prompt[, title] [, default] [, xpos] [, ypos] [, helpfile, context])`

It is rare that you will need to use [, xpos] [, ypos] or [, helpfile, context]. See Excel help for details on these. It should also be noted that, the

InputBox Function returns a String only when used in this way. (more on another way soon).

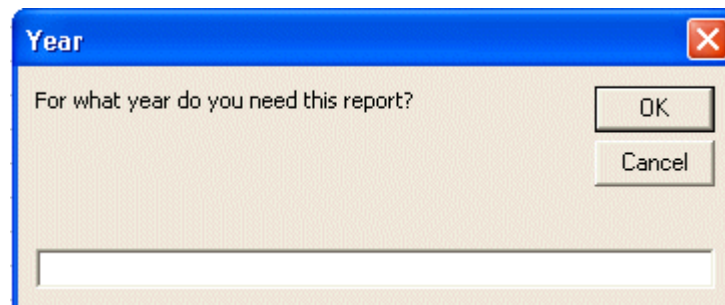
You will use the input box when you want to require a SINGLE value from the user.



The input box above is the basic input box and is generated by the following code:

```
Sub proInput()  
Dim varInput As Integer  
  
    varInput = InputBox("For what year do you need this report?")  
    Sheets("Intro").Range("A1").value = varInput  
  
End Sub
```

A variable is created "varInput" to receive the value submitted by the user. Here the value is entered in a cell but it could be use in the rest of a more complex VBA procedure. Note here that the variable is declared as "Integer" so if the user submits a text "string" an error is generated. Shown below is how to handle such errors so that the user is not confronted with the VBA error dialog box.



You can also use your own title for the input box ("Year" in the example above). To do so, you must use the extended code that follows:

```
InputBox("For what year do you need this report?","Year")
```

Here is the code to handle errors (cancel or invalid response form the user):

```
Sub proInput()  
Dim varInput As Integer
```

```

On Error GoTo addError
varInput = InputBox("For what year do you need this report?", "Year")

addError:

    MsgBox "No value submitted or value not an integer, procedure aborted."
    Exit Sub

End Sub

```

Another example

Ok, lets assume we need to gather the name of the user and do some stuff depending on that name. The macro below will achieve this.

Sub GetUserName()

```
Dim strName As String
```

```

strName = InputBox(Prompt:="You name please.", _
    Title:="ENTER YOUR NAME", Default:="Your Name here")

```

```

If strName = "Your Name here" Or _
    strName = vbNullString Then
    Exit Sub

```

```
Else
```

```
    Select Case strName
```

```
        Case "Bob"
```

```
            'Do Bobs stuff
```

```
        Case "Bill"
```

```
            'Do Bills stuff
```

```
        Case "Mary"
```

```
            'Do Marys stuff
```

```
        Case Else
```

```
            'Do other stuff
```

```
    End Select
```

```
End If
```

End Sub

Note the use of the **Select Case Statement** to determine the name the user supplies.

Application.InputBox

When we precede the InputBox Function with "Application" we get an InputBox Method that will allow us to specify the type of info that we can collect. Its Syntax is;

InputBox(Prompt, Title, Default, Left, Top, HelpFile, HelpContextId, Type)

As you can see, the Prompt, Title and Default are the same as in the InputBox Function. However, it is the last argument "**Type**" that allows us to specify the type of data we are going to collect. These are as shown below;

Type:=0 A formula
Type:=1 A number
Type:=2 Text (a string)
Type:=4 A logical value (True or False)
Type:=8 A cell reference, as a Range object

We have already covered a String being returned so lets look, what most believe, to be the most useful. That is, Type 8 & 1. The code below shows how we can allow the user to specify a Range Object.

Sub RangeDataType()

Dim rRange As Range

On Error Resume Next

Application.DisplayAlerts = False

Set rRange = Application.InputBox(Prompt:= _
"Please select a range with your Mouse to be bolded.", _
Title:="SPECIFY RANGE", Type:=8)

On Error GoTo 0

Application.DisplayAlerts = True

If rRange Is Nothing Then

Exit Sub

Else

rRange.Font.Bold = True

End If

End Sub

Note the use of both, **On Error Resume Next** and **Application.DisplayAlerts = False**. These stop Excel from trying to handle any bad input from the user, or if they Cancel. Take the lines out, run the code and click Cancel, or specify a non valid range and Excel will bug out in the case of Cancel.

Let's now look at how we can collect a numeric value from a user.

Sub NumericDataType()

Dim INum As Long

```

On Error Resume Next
    Application.DisplayAlerts = False
    INum = Application.InputBox _
        (Prompt:="Please enter you age.", _
        Title:="HOW OLD ARE YOU", Type:=1)
On Error GoTo 0
Application.DisplayAlerts = True

    If INum = 0 Then
        Exit Sub
    Else
        MsgBox "You are " & INum & " years old."
    End If

```

End Sub

Again, we take over the possibility of the user electing to Cancel, or entering a non-numeric value. If they enter anything that is not numeric and click OK, they are taken back to the InputBox Method with their entry highlighted.

Unlike the **InputBox Function**, we can combine different Types for the **InputBox Method** and take action based on their data type. See example.

Sub Numeric_RangeDataType()

Dim vData

```

On Error Resume Next
    Application.DisplayAlerts = False

    vData = Application.InputBox _
        (Prompt:="Please select a single cell housing the number, " _
        & "or enter the number directly.", _
        Title:="HOW OLD ARE YOU", Type:=1 + 8)

```

```

On Error GoTo 0

```

```

Application.DisplayAlerts = True

```

```

    If IsNumeric(vData) And vData <> 0 Then
        MsgBox "You are " & vData & " years old."
    Else
        Exit Sub
    End If

```

End Sub

Loops and Conditional Statements

Why Loops?

The purpose of a loop is to get Excel to repeat a piece of code a certain number of times. How many times the code gets repeated can be specified as a fixed number (e.g. do this 10 times), or as a variable (e.g. do this for as many times as there are rows of data).

Loops can be constructed many different ways to suit different circumstances. Often the same result can be obtained in different ways to suit your personal preferences.

Loop Structures

For ... Next

Use **For ... Next** loop if the number of loops is already defined and known. A **For ... Next** loop uses a counter variable that increases or decreases in value during each iteration of the loop. This loop structure is being used the most for our examples on this site.

Here is an example of the **For ... Next** loop:

```
For i = 1 to 10  
    Cells(i, 1) = i  
Next i
```

	A	
1	1	
2	2	
3	3	
4	4	
5	5	
6	6	
7	7	
8	8	
9	9	
10	10	
11		
12		

In this example, *i* is the counter variable from 1 to 10. The looping process will send value to the first column of the active sheet and print *i* (which is 1 to 10) to row 1 to 10 of that column.

Note that the counter variable, by default, increases by an increment of 1.

For ... Next Loop With Step

You can use the **Step** Keyword to specify a different increment for the counter variable.

For example:

```
For i = 1 to 10 Step 2  
    Cells(i, 1) = i  
Next i
```

This looping process will print values with an increment of 2 on row 1, 3, 5, 7 and 9 on column one.

	A
1	1
2	
3	3
4	
5	5
6	
7	7
8	
9	9
10	

You can also have decrement in the loop by assign a negative value after the **Step** keyword.

For example:

```
For i = 10 to 1 Step -2  
    Cells(i, 1) = i  
Next i
```

This looping process will print values with an increment of -2 starts from 10 on row 10, 8, 6, 4 and 2 on column one.

	A
1	
2	2
3	
4	4
5	
6	6
7	
8	8
9	
10	10

Do While ... Loop

You can use the **Do While ... Loop** to test a condition at the start of the loop. It will run the loop as long as the condition is true and stops when the condition becomes false. For Example:

```
i = 1
Do While i =< 10
    Cells(i, 1) = i
    i = i + 1
Loop
```

This looping process yields the same result as in the **For ... Next** structures example.

One thing to be caution is that sometimes the loop might be a infinite loop. And it happens when the condition never becomes false. In such case, you can stop the loop by press **[ESC]** or **[CTRL] + [BREAK]**.

Do Until ... Loop

You can test the condition at the beginning of the loop and then run the loop until the test condition becomes true.

Example:

```
i = 1
Do Until i = 11
    Cells(i, 1) = i
    i = i + 1
Loop
```

This looping process yields the same result as in the **For ... Next** structures example.

Do ... Loop While

When you want to make sure that the loop will run at least once, you can put the test at the end of loop. The loop will stop when the condition becomes false. (compare this loop structure to the Do ... While Loop.)

For Example:

```
i = 1
Do
    Cells(i, 1) = i
    i = i + 1
Loop While i < 11
```

This looping process yields the same result as in the **For ... Next** structures example.

Do ... Loop Until

This loop structure, like the **Do ... Loop While**, makes sure that the loop will run at least once, you can put the test at the end of loop. The loop will stop when the condition becomes true. (compare this loop structure to the **Do ... Until** Loop.)

For Example:

```
i = 1
Do
    Cells(i, 1) = i
    i = i + 1
Loop Until i = 11
```

This looping process yields the same result as in the **For ... Next** structures example.

Conditional Statements

The statements that I use in my VBA Excel Macros more often are: If..Then..End If, Do...Loop, For...Next and Select Case

If..Then...End If

When there is only one condition and one action, you will use the simple statement:

```
If Selection.Value > 10 Then
    Selection.Offset(1,0).Value = 100
End If
```

In plain English: if the value of the selected cell is greater than 10 then the value of the cell below is 100 if not do nothing.

Note: Tests on strings are case sensitive so when you test a string of characters and you don't know if the user will use upper case or lower case letters, use the function LCase function within your test and write the strings in your code in lower case letters:

```
If LCase(Selection.Value).Value= "yes" then...
```

With this approach, your test will be valid whatever case your client uses (Yes, YES or any other combination of cases).

If..Then...End If (multiple tiers)

When there are only two conditions that you want to check sequentially, you will use the statement:

```
If Selection.Value > 10 Then
    If Selection.Value = 12 Then
```

```
Selection.Offset(1,0).Value = 100
End If
End If
```

In plain English: first check if the value of the selected cell is greater than 10. If it is not do nothing. If it is check if the value of the selected cell is equal to 12. if so set the value of the cell below at 100 else do nothing.

If..Then...And...End If

When there are two inclusive conditions, you will use the statement:
If Selection.Value >= 10 And Selection.Offset(0,1).Value < 20 Then
Selection.Offset(1,0).Value = 100
End If

In plain English: if the value of the selected cell is greater or equal to 10 and smaller than 20 the value of the cell below is 100 otherwise do nothing.

If..Then...Or...End If

When there are two exclusive conditions and one action, you will use the statement:
If Selection.Value = 10 Or Selection.Offset(0,1).Value = 20 Then
Selection.Offset(1,0).Value = 100
End If

In plain English: if the value of the selected cell is equal to 10 or equal to 20 then the value of the cell below is 100 otherwise do nothing.

If..Then...Else...End If

When there is only one condition but two actions, you will use the statement:
If Selection.Value > 10 Then
Selection.Offset(1,0).Value = 100
Else
Selection.Offset(1,0).Value = 50
End If

In plain English: if the value of the selected cell is greater than 10 then the value of the cell below is 100 else the value of the cell below is 50.

If..Then..Elseif...End If

When there are more than one condition linking each to a different action you will use the statement:
If Selection.Value = 1 Then
Selection.Offset(1, 0).Value = 10


```

Elseif Selection.Value = 2 Then
    Selection.Offset(1, 0).Value = 20
Elseif Selection.Value = 3 Then
    Selection.Offset(1, 0).Value = 30
Elseif Selection.Value = 4 Then
    Selection.Offset(1, 0).Value = 40
Elseif Selection.Value = 5 Then
    Selection.Offset(1, 0).Value = 50
End If

```

In plain English: If the value of the selected cell is 1 then the value of the cell below is 10 but if the value of the selected cell is 2 then the value of the cell below is 20 but if the value of the selected cell is 3 then the value of the cell below is 30 but if the value of the selected cell is 4 then the value of the cell below is 40 but if the value of the selected cell is 5 then the value of the cell below is 50 but then if the value of the selected cell is not 1, 2, 3, 4 or 5 do nothing.

You can use IF..Elseif when you have 2 or 3 conditions but if you have more you might want to use the much more interesting Select..Case statement.

Select Case

Select Case statement is an alternative to the Elseif statement. This method is more efficient and readable in coding the the **If ... Then ... Elseif** statment.

Example:

```

Select Case Grade
    Case Is >= 90
        LetterGrade = "A"
    Case Is >= 80
        LetterGrade = "B"
    Case Is >= 70
        LetterGrade = "C"
    Case Is >= 60
        LetterGrade = "D"
    Case Else
        LetterGrade = "Sorry"
End Select

```

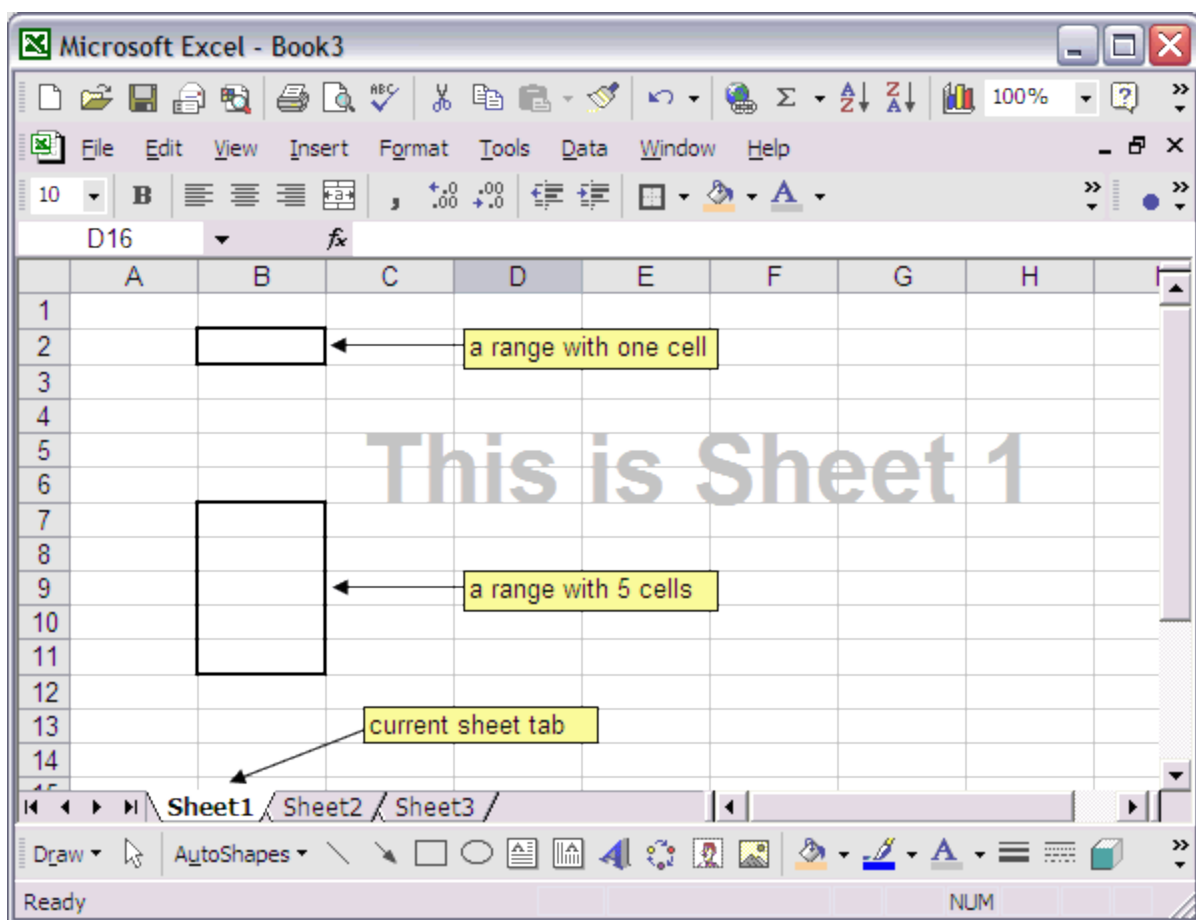
Objects and Collections

Introduction

Objects are the fundamental building blocks of Visual Basic. An **object** is a special type of variable that contains both data and codes. A **collection** is a group of objects of the same class. The most used Excel objects in VBA programming are Workbook, Worksheet, Sheet, and Range.

Workbooks is a collection of all Workbook objects. Worksheets is a collection of Worksheet objects. The Workbook object represents a workbook, the Worksheet object represents a worksheet, the Sheet object represents a worksheet or chartsheet, and the Range object represents a range of cells.

The following figure shows all the objects mentioned. The workbook (Excel file) is currently Book3.xls. The current worksheet is Sheet1 as the Sheet Tab indicated. Two ranges are selected, range B2 and B7:B11.



Collections

Many types of objects in Excel come in groups. There may be multiple workbooks open in Excel at a given time. Each workbook may have multiple worksheets. A worksheet can have multiple graphs. These multiple objects are gathered together in what's called a collection. The collection itself is an object, with its own set of properties and methods.

For example, the Application object has a property called Workbooks, which represents a Workbook Collection object. It includes a single Workbook object for each workbook currently open in Excel. Similarly, the Workbook object has a property called Worksheets which is a Worksheet Collection object, including one Worksheet object for each worksheet in the workbook.

Referencing Items in A Collection

The simplest way to reference an item in a collection is by number. All objects in a collection have a numeric index, starting with 1. To reference an element in a collection by number, simply put the number in parentheses after the collection object. Read below.

Workbook and Worksheet Object

A **workbook** is the same as an Excel file. The Workbook collection contains all the workbooks that are currently opened. Inside of a workbook contains at least one **worksheet**. In VBA, a worksheet can be referenced as followed:

worksheet in the collection Worksheets("Sheet1")

Worksheets("Sheet1") is the worksheet that named "Sheet1."

Another way to refer to a worksheet is to use n

The above refers to the first.

Number index like the following:

Worksheets(1)

* Note that Worksheets(1) is not necessary the same sheet as Worksheets("Sheet1").

Sheets is a collection of worksheets and chart sheets (if present). A sheet can be indexed just like a worksheet. Sheets(1) is the first sheet in the workbook.

To refer sheets (or other objects) with the same name, you have to qualify the object. For example:

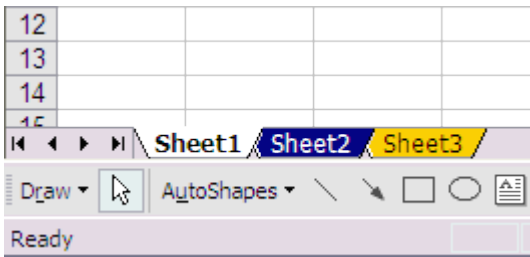
Workbooks("Book1").Worksheets("Sheet1")

Workbooks("Book2").Worksheets("Sheet1")

The sheet tab on the bottom the spreadsheet (worksheet) shows which sheet is active. As the figure below shows, the active sheet is "Sheet1" (show in bold font

and white background).

If the object is not qualified, the active or the current object (for example workbook or worksheet) is used.



* You can change the color of the sheet tabs by right click the tab, choose Tab Color, then select the color for the tab.

The sub routine below shows the name of each sheet in the current opened workbook. You can use **For Each...Next loop** to loop through the Worksheets collection.

```
Sub ShowWorkSheets()  
    Dim mySheet As Worksheet  
  
    For Each mySheet In Worksheets  
        MsgBox mySheet.Name  
    Next mySheet  
  
End Sub  
  
Sub SeeSheetNames()  
    Dim iSheetCount As Integer  
    Dim iSheet As Integer  
    iSheetCount = ActiveWorkbook.Worksheets.Count  
    For iSheet = 1 To iSheetCount  
        Worksheets(iSheet).Activate  
        MsgBox Worksheets(iSheet).Name  
    Next iSheet  
  
End Sub
```

Range Object and Cells Property

Range represents a cell, a row, a column, a selection of cells containing one or more contiguous blocks of cells, or a 3-D range. We will show you some examples on how Range object can be used.

The following example places text "AB" in range A1:B5, on Sheet2.

```
Worksheets("Sheet2").Range("A1:B5") = "AB"
```

	A	B
1	AB	AB
2	AB	AB
3	AB	AB
4	AB	AB
5	AB	AB
6		

Note that, `Worksheets.Range("A1", "B5") = "AB"` will yield the same result as the above example.

The following place "AAA" on cell A1, A3, and A5 on Sheet2.

```
Worksheets("Sheet2").Range("A1, A3, A5") = "AAA"
```

	A
1	AAA
2	
3	AAA
4	
5	AAA

Range object has a **Cells** property. This property is used in every VBA projects on this website (**very important**). The Cells property takes one or two indexes as its parameters.

For example,

`Cells(index)` or `Cells(row, column)`

where *row* is the row index and *column* is the column index.

The following three statements are interchangeable:

ActiveSheet.Range.Cells(1,1)
 Range.Cells(1,1)
 Cells(1,1)

The following returns the same outcome:

Range("A1") = 123 and Cells(1,1) = 123

The following puts "XYZ" on Cells(1,12) or Range("L1") assume cell A1 is the current cell:

Cells(12) = "XYZ"

The following puts "XYZ" on cell C3:

Range("B1:F5").cells(12) = "ZYZ"

	A	B	C	D	E	F	G
1		1	2	3	4	5	
2		6	7	8	9	10	
3		11	XYZ 12	13	14	15	
4		16	17	18	19	20	
5		21	22	23	24	25	
6							

The small gray number on each of the cells is just for reference purpose only. They are used to show how the cells are indexed within the range.

Here is a sub routine that prints the corresponding row and column index from A1 to E5.

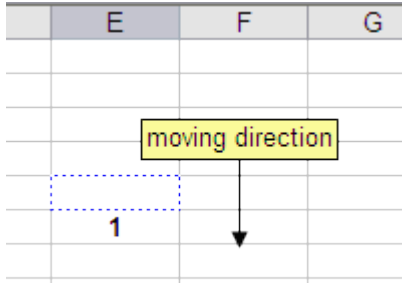
```
Sub CellsExample()
  For i = 1 To 5
    For j = 1 To 5
      Cells(i, j) = "Row " & i & " Col " & j
    Next j
  Next i
End Sub
```

	A	B	C	D	E	
1	Row 1 Col 1	Row 1 Col 2	Row 1 Col 3	Row 1 Col 4	Row 1 Col 5	
2	Row 2 Col 1	Row 2 Col 2	Row 2 Col 3	Row 2 Col 4	Row 2 Col 5	
3	Row 3 Col 1	Row 3 Col 2	Row 3 Col 3	Row 3 Col 4	Row 3 Col 5	
4	Row 4 Col 1	Row 4 Col 2	Row 4 Col 3	Row 4 Col 4	Row 4 Col 5	
5	Row 5 Col 1	Row 5 Col 2	Row 5 Col 3	Row 5 Col 4	Row 5 Col 5	
6						

Range object has an **Offset** property that can be very handy when one wants to move the active cell around. The following examples demonstrate how the Offset property can be implemented (assume the current cell before the move is E5):

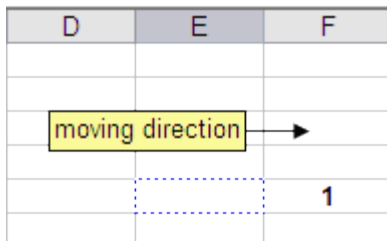
`ActiveCell.Offset(1,0) = 1`

Place a "1" one row under E5 (on E6)



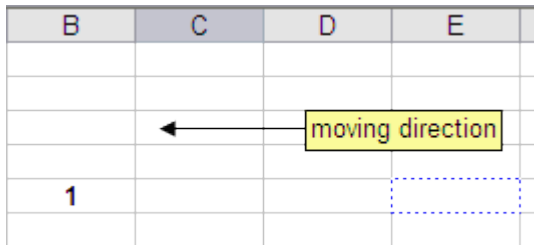
`ActiveCell.Offset(0,1) = 1`
(on F5)

Place a "1" one column to the right of E5



`ActiveCell.Offset(0,-3) = 1`
(on B5)

Place a "1" three columns to the left of E5



Referencing Ranges in Other Sheets

Switching between sheets by activating the needed sheet can drastically slow down your code. Instead, you can refer to a sheet that is not active by referencing the Worksheet object first:

```
Worksheets("Sheet1").Range("A1")
```

This line of code references Sheet1 of the active workbook even if Sheet2 is the active sheet.

If you need to reference a range in another workbook, then include the Workbook object, the Worksheet object, and then the Range object:

```
Workbooks("InvoiceData.xls").Worksheets("Sheet1").Range("A1")
```

Be careful if you use the Range property as an argument within another Range property. You must identify the range fully each time. Let's say that Sheet1 is your active sheet and you need to total data on Sheet2:

```
WorksheetFunction.Sum(Worksheets("Sheet2").Range(Range("A1"),  
Range("A7")))
```

This line does not work. Why? Because Range(Range("A1"), Range("A7")) refers to an extra range at the beginning of the code line. Excel does not assume that you want to carry the Worksheet object reference over to the other Range objects. So what do you do? Well, you could write this:

```
WorksheetFunction.Sum(Worksheets("Sheet2").Range(Worksheets("Sheet2").  
Range("A1"), Worksheets("Sheet2").Range("A7")))
```

But this is not only a long line of code, it is difficult to read! Thankfully, there is a simpler way, With...End With:

```
With Worksheets("Sheet2")  
WorksheetFunction.Sum(.Range(.Range("A1"), .Range("A7")))  
End With
```

Notice now that there is a .Range in your code, but without the preceding object reference. That's because With Worksheets("Sheet2") implies that the object of the Range is the Worksheet

Working with cells

Use the "Range" object when you work with cells

To select all the cells

`Cells.Select`

Selecting a single cell

`Range("A1").Select`

Selecting contiguous cells

`Range("A1:G8").Select`

Selecting non-contiguous cells

`Range("A1,B6,D9").Select`

`Range("A1,B6:B10,D9").Select`

To select rows or columns

`Rows("1").Select`

`Columns("A").Select`

or

`ActiveCell.EntireRow.Select`

`ActiveCell.EntireColumn.Select`

To select many contiguous rows or columns

`Columns("A:C").Select`

`Rows("1:5").Select`

To select many non-contiguous rows and columns

NOTE: Use the "Range" object and not Columns or Rows

`Range("A:A, C:C, E:F").Select`

`Range("1:1,5:6,9:9").Select`

Offset property

To move to the first cell in a row:

`ActiveCell.Offset(0, -ActiveCell.Column + 1).Select`

To move to the first cell in the column:

`ActiveCell.Offset(-ActiveCell.Row + 1,0).Select`

And to move to the top/left cell of the sheet.....

`Range("A1").Select`

To move 13 rows down and 14 column to the right of a cell you can use either:

`ActiveCell.Offset(13, 14).Select`

`Selection.Offset(13, 14).Select`

`Range("G8").Offset(13, 14).Select`

To move 13 rows up and 14 column to the left of a cell you can use either:

`ActiveCell.Offset(-13, -14).Select`

```
Selection.Offset(-13, -14).Select  
Range("G8").Offset(-13, -14).Select
```

But most of the time you don't know the number of rows and columns that you have in a dataset so the most important property of the Range object, the ActiveCell and the Selection is the CurrentRegion. It includes all the cells until there is an empty column and an empty row. For example if there are data from cell A1 to cell K5600 the line Range("A1").CurrentRegion.Select will select all the cells from cell A1 to cell K5600. You can then count the rows and the columns of the current region and easily move from the first to the last column or row. To move to the end of the dataset use:

```
Range("A1").Offset(varNbRows, varNbColumns).Select  
and to position to add a new record  
Range("A1").Offset(varNbRows, 0).Select
```

Most book on VBA for Excel will suggest the following lines of code to go to the last row or the last column:

```
Range("A1", Range("A1").End(xlDown)).Select  
Range(ActiveCell, ActiveCell.End(xlDown)).Select  
Range("A1", Range("A1").End(xlToRight)).Select  
Range(ActiveCell, ActiveCell.End(xlToLeft)).Select
```

but if you use these lines of code and there data only in cell A1 you end up in cell A65000 or IV1 and you have a problem. That is why its best to always select the current region of a dataset and use the offset property as in the example above.

We have use the Offset property to move up and down or left and right a certain number of cells and select the cell that we then reach. To select all the cells from a cell or the activecell to 10 cells to the right

```
Range("A1", Range("A1").Offset(0, 10)).Select  
Range(ActiveCell, ActiveCell.Offset(0, 10)).Select
```

To select a range from a cell or the activecell to 10 cells to the left

```
Range("A1", Range("A1").Offset(0, -10)).Select  
Range(ActiveCell, ActiveCell.Offset(0, -10)).Select
```

To select a range from a cell or the activecell to 10 cells below

```
Range("a1", Range("a1").Offset(10, 0)).Select  
Range(ActiveCell, ActiveCell.Offset(10, 0)).Select
```

Using the CurrentRegion Property to Quickly Select a Data Range

CurrentRegion returns a range object representing a set of contiguous data. As long as the data is surrounded by one empty row and one empty column, you can select the table with CurrentRegion:

RangeObject.CurrentRegion

The following line would select contiguous range of cells around cell A1:

Range("A1").CurrentRegion.Select

Using the SpecialCells Method to Select Specific Cells

Even Excel power users may never have encountered the Go To Special dialog box. If you press the F5 key in an Excel worksheet, you get the normal Go To dialog box. In the lower-left corner of this dialog is a button called Special. Click that button to get to the super-powerful Go To Special dialog

In the Excel interface, the Go To Special dialog enables you to select only cells with formulas, or only blank cells, or only the visible cells. Selecting visible cells only is excellent for grabbing the visible results of AutoFiltered data.

To simulate the Go To Special dialog in VBA, use the SpecialCells method. This enables you to act on cells that meet a certain criteria:

RangeObject.SpecialCells(Type, Value)

This method has two parameters: Type and Value. Type is one of the xlCellType constants:

- xlCellTypeAllFormatConditions
- xlCellTypeAllValidation
- xlCellTypeBlanks
- xlCellTypeComments
- xlCellTypeConstants
- xlCellTypeFormulas
- xlCellTypeLastCell
- xlCellTypeSameFormatConditions
- xlCellTypeSameValidation
- xlCellTypeVisible

Value is optional and can be one of the following:

- xlErrors
- xlLogical
- xlNumbers
- xlTextValues.

The following returns all the ranges that have conditional formatting set up. It puts a border around each contiguous section it finds:

```
Set rngCond = ActiveSheet.Cells.SpecialCells(xlCellTypeAllFormatConditions)
If Not rngCond Is Nothing Then
    rngCond.BorderAround xlContinuous
End If
```

Methods and Properties

Each object contains its own methods and properties.

A **Property** represents a built-in or user-defined characteristic of the object. A **method** is an action that you perform with an object. Below are examples of a method and a property for the Workbook Object:

Workbooks.Close

Close method close the active workbook

Workbooks.Count

Count property returns the number of workbooks that are currently opened

Some objects have default properties. For example, Range's default property is Value.

The following yields the same outcome.

```
Range("A1") = 1    and    Range("A1").Value = 1
```

Here are examples on how to set and to get a Range property value:

The following sets the value of range A1 or Cells(1,1) as "2005". It actually prints "2005" on A1.

```
Range("A1").Value = 2005
```

The following gets the value from range A1 or Cells(1,1).

```
X = Range("A1").Value
```

Method can be used with or without argument(s). The following two examples demonstrate this behavior.

Methods That Take No Arguments:

```
Worksheets("Sheet").Column("A:B").AutoFit
```

Methods That Take Arguments:

```
Worksheets("Sheet1").Range("A1:A10").Sort _  
Worksheets("Sheet1").Range("A1")
```

Worksheets("Sheet1").Range("A1") is the **Key** (or column) to sort by.

Assigning Object Variables and Using Named Argument

Sometime a method takes more than one argument. For example, the Open method for the Workbook object, takes 12 arguments. To open a workbook with password protection, you would need to write the following code:

```
Workbooks.Open "Book1.xls", , , , "pswd"
```

Since this method takes so many arguments, it is easy to misplace the password argument. To overcome this potential problem, one can use named arguments like the following example:

```
Workbook.Open fileName:="Book1.xls", password:="pswd"
```

You can also assign an object to an object variable using the **Set** Statement.

For example:

```
Dim myRange as Range  
Set myRange = Range("A1:A10")
```

Error Handling

Introduction

Error handling refers to the programming practice of anticipating and coding for error conditions that may arise when your program runs. Errors in general come in three flavors: compiler errors such as undeclared variables that prevent your code from compiling; user data entry error such as a user entering a negative value where only a positive number is acceptable; and run time errors, that occur when VBA cannot correctly execute a program statement. We will concern ourselves here only with run time errors. Typical run time errors include attempting to access a non-existent worksheet or workbook, or attempting to divide by zero. The example code in this article will use the division by zero error (Error 11) when we want to deliberately raise an error.

Your application should make as many checks as possible during initialization to ensure that run time errors do not occur later. In Excel, this includes ensuring that required workbooks and worksheets are present and that required names are defined. The more checking you do before the real work of your application begins, the more stable your application will be. It is far better to detect potential error situations when your application starts up before data is change than to wait until later to encounter an error situation.

If you have no error handling code and a run time error occurs, VBA will display its standard run time error dialog box. While this may be acceptable, even desirable, in a development environment, it is not acceptable to the end user in a production environment. The goal of well designed error handling code is to anticipate potential errors, and correct them at run time or to terminate code execution in a controlled, graceful method. Your goal should be to prevent unhandled errors from arising.

A note on terminology: Throughout this article, the term *procedure* should be taken to mean a Sub, Function, or Property procedure, and the term *exit statement* should be taken to mean Exit Sub, Exit Function, or Exit Property. The term *end statement* should be taken to mean End Sub , End Function, End Property, or just End.

The On Error Statement

The heart of error handling in VBA is the On Error statement. This statement instructs VBA what to do when an run time error is encountered. The On Error statement takes three forms.

On Error Goto 0
On Error Resume Next
On Error Goto <label>:

The first form, On Error Goto 0, is the default mode in VBA. This indicates that when a run time error occurs VBA should display its standard run time error message box, allowing you to enter the code in debug mode or to terminate the VBA program. When On Error Goto 0 is in effect, it is the same as having no enabled error handler. Any error will cause VBA to display its standard error message box.

The second form, On Error Resume Next, is the most commonly used and misused form. It instructs to VBA to essentially ignore the error and resume execution on the next line of code. It is very important to remember that On Error Resume Next does **not** in any way "fix" the error. It simply instructs VBA to continue as if no error occurred. However, the error may have side effects, such as uninitialized variables or objects set to Nothing. It is the responsibility of your code to test for an error condition and take appropriate action. You do this by testing the value of Err.Number and if it is not zero execute appropriate code. For example,

```
On Error Resume Next
N = 1 / 0 ' cause an error
If Err.Number <> 0 Then
    N = 1
End If
```

This code attempts to assign the value 1 / 0 to the variable N. This is an illegal operations, so VBA will raise an error 11 -- Division By Zero -- and because we have On Error Resume Next in effect, code continues to the If statement. This statement tests the value of Err.Number and assigns some other number to N.

The third form On Error of is On Error Goto <label>:which tells VBA to transfer execution to the line following the specified line label. Whenever an error occurs, code execution immediately goes to the line following the line label. None of the code between the error and the label is executed, including any loop control statements.

```
On Error Goto ErrHandler:
N = 1 / 0 ' cause an error
'
' more code
'

Exit Sub
ErrHandler:
' error handling code
Resume Next
End Sub
```

Enabled And Active Error Handlers

An error handler is said to be **enabled** when an On Error statement is executed. Only one error handler is enabled at any given time, and VBA will behave according to the enabled error handler. An **active** error handler is the code that executes when an error occurs and execution is transferred to another location via a On Error Goto <label>: statement.

Error Handling Blocks And On Error Goto

An error handling block, also called an error handler, is a section of code to which execution is transferred via a On Error Goto <label>: statement. This code should be designed either to fix the problem and resume execution in the main code block or to terminate execution of the procedure. You can't use the On Error Goto <label>: statement merely skip over lines. For example, the following code will not work properly:

```
On Error GoTo Err1:
Debug.Print 1 / 0
' more code
Err1:
On Error GoTo Err2:
Debug.Print 1 / 0
' more code
Immediate Window
```

The Immediate Window is a window in the VBE in which you can enter commands and view and change the contents of variables while you code is in Break mode or when no macro code is executing. (Break mode is the state of VBA when code execution is paused at a break point (see Breakpoints, below). To display the Immediate Window, press CTRL+G or choose it from the View menu.

In the Immediate Window, you can display the value of a variable by using the ? command. Simply type ? followed by the variable name and press Enter. VBA will display the contents of the variable in the Immediate Window. For example,

```
?ActiveCell.Address
$A$10
```

You can also execute VBA commands in the Immediate Window by omitting the question mark and entering the command followed by the Enter key:

Debug.Print

You can use the Debug.Print statement anywhere in your code to display messages or variable values in the Immediate Window. These statements don't require any confirmation or acknowledgement from the user so they won't affect the operation of your code. For example, you can send a message to the Immediate Window when a particular section of code is executed.

```
'  
' some code  
'
```

```
Debug.Print "Starting Code Section 1"
```

The liberal use of Debug.Print statements makes it easy to track the execution of your code. Debug.Print statements have no effect on the execution of your code and so it is safe to leave them in code projects that are distributed to end users. Debug.Print statements send messages to the Immediate Window, so you should have this window open in order to see the messages.

Unfortunately, there is no way to programmatically clear the Immediate Window. This is a shortcoming that has frustrated many programmers.

```
Range("A1").Value = 1234
```

Break Points

A break point is a setting on a line of code that tells VBA to pause execution immediately before that line of code is executed. Code execution is placed in what is called *break mode*. When VBA is in break mode, you can enter commands in to the Immediate Window to display or change the values of variables.

To put a break point on a line of code, place the cursor on that line and press F9 or choose "Toggle Breakpoint" from the Debug menu. To remove a break point, place the cursor on the line with the break point and press F9 or choose "Toggle Breakpoint" from the Debug menu. When a line contains a break point, it is displayed with a brick colored background. Immediately before this line of code is executed, it will appear with a yellow background. Remember, when a break point is encountered, code execution is paused but that line of code has not yet executed. You cannot place break points on blank lines, comment lines, or variable declaration lines (lines with Dim statements).

After a break point is encountered, you can resume normal code execution by pressing F5 or choosing "Continue" from the Run menu, or stepping through the code line by line (see below). Note that break points are not saved in the workbook file. If you close the file, all break points are removed. Breakpoints are preserved as long as the file is open.

Stepping Through Code

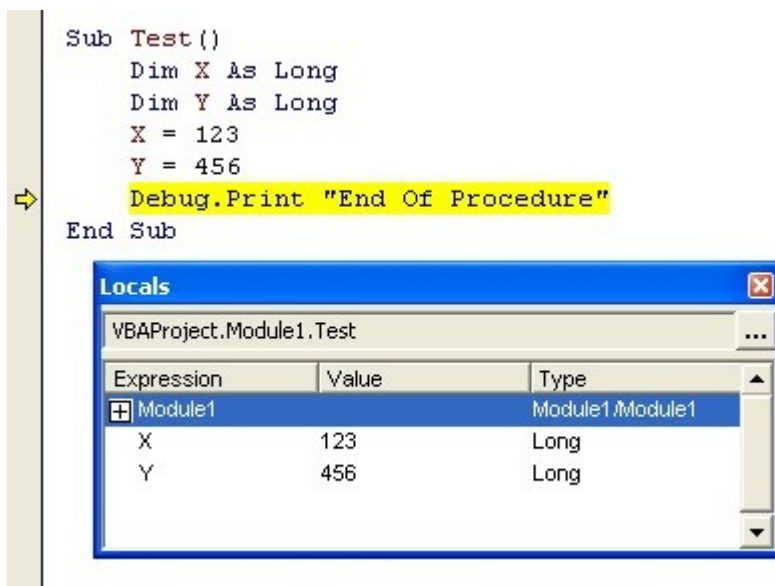
Normally, your code runs unattended. It executes until its logical end. However, when you are testing code, it is often useful to step through the code line by line, watching each line of code take effect. This makes it easy to determine exactly what line is causing incorrect behavior. You can step through code line by line by pressing the F8 key to start the procedure in which the cursor is, or when VBA is paused at a break point. Pressing F8 causes VBA to execute each line one at a time, highlighting the next line of code in yellow. Note, the highlighted line is the line of code that will execute when you press F8. It has not yet been executed.

If your procedure calls another procedure, pressing F8 will cause VBA to step inside that procedure and execute it line by line. You can use SHIFT+F8 to "Step Over" the procedure call. This means that the entire called procedure is executed as one line of code. This can make debugging simpler if you are confident that the problem does not lie within a called procedure.

When you are in a called procedure, you can use CTRL+SHIFT+F8 to "Step Out" of the current procedure. This causes VBA to execute until the end of the procedure is reached (an End Sub or Exit Sub statement) and then stop at the line of code immediately following the line which called the procedure.

Locals Window

The Locals Window displays all the variables in a procedure (as well as global variables declared at the project or module level) and their values. This makes it easy to see exactly what the value of each variable is, and where it changes, as you step through the code. You can display the Locals Window by choosing it from the View menu. The Locals Window does not allow you to change the values of variables. It simply displays their names and values. The Locals Window is shown below. Note that the variables X and Y in procedure Test are displayed in the window. The line highlighted in yellow is the current line of execution -- it is the next line of code that VBA will execute.

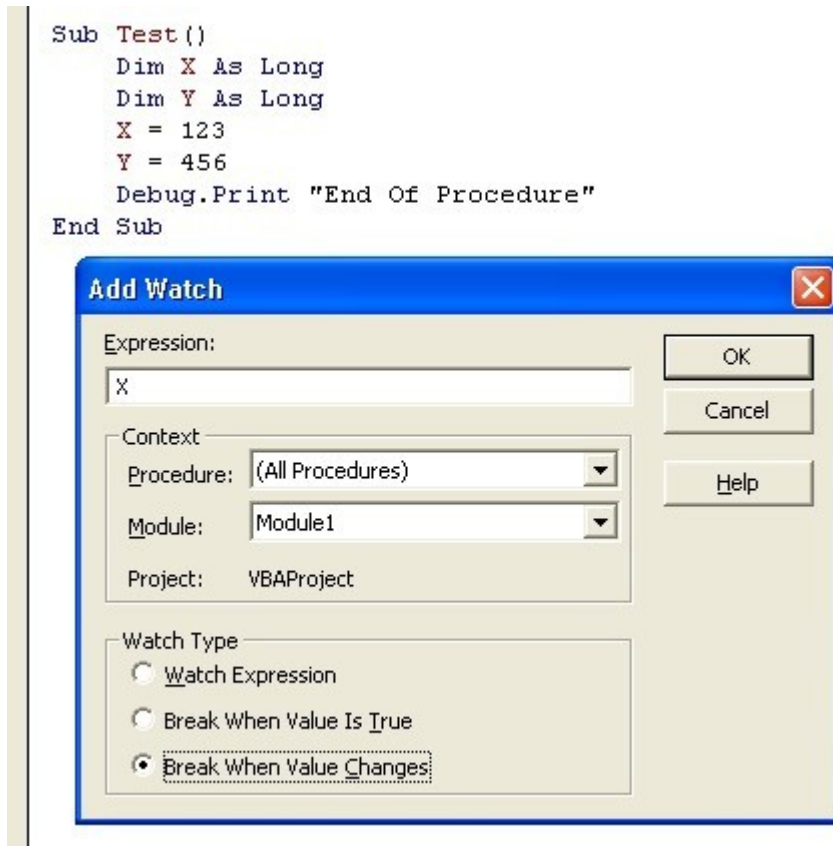


Watch Window

The Watch Window allows you to "watch" a specific variable or expression and cause code execution to pause and enter break mode when the value of that variable or expression is True (non-zero) or whenever that variable is changed.

(Note, this is not to be confused with the Watch object and the Watches collection).

To display the Watch Window, choose it from the View menu. To create a new watch, choose Add Watch from the Debug menu. This will display the Add Watch window, as shown below.



There are three types of watches, shown in the Watch Type group box. "Watch Expression" causes that watch to work much like the Locals Window display. It simply displays the value of a variable or expression as the code is executed. "Break When Value Is True" causes VBA to enter break mode when the watch variable or expression is True (not equal to zero). "Break When Value Changes" causes VBA to enter break mode when the value of the variable or expression changes value.

You can have many watches active in your project, and all watches are displayed in the Watch Window. This makes it simple to determine when a variable changes value.

Events

Beginning VBA: Events

Every time you do something in Excel, an event happens. Selecting cells, changing cells, clicking a commandbar button: all these are events. Excel exposes some, but not all, events in VBA and you, as a programmer, can write code to react to these events.

Where To Put Event Code

Event code goes in class modules. For purposes of this post, we'll be dealing with those special class modules named ThisWorkbook, Sheet1, etc. When you open the ThisWorkbook code window in the VBE, there are two dropdown boxes at the top of the window. The left contains a list of objects associated with that code window and the right contains a list of the events that are available to you.

Let's look at some **commonly used events**:

Events

Workbook_Open

This event fires whenever the workbook containing the event is opened. It's often used to do set up work for your application, such as hiding sheets or creating custom commandbars. There are no arguments to Workbook_Open.

Worksheet_Change

This event fires whenever a cell's value changes on the worksheet. You can write code to test the value entered in a cell, for example, replacing the built-in Data Validation feature. It's also commonly used to apply conditional formatting to cells where you need more than three conditions. The Target argument is a reference to the cell(s) that were changed. Remember that this event doesn't fire as a result of a formula recalculating, it requires input into a cell.

Worksheet_SelectionChange

This event fires when the user selects a cell. Its argument is the same as the Change event, that is, a reference to the cell(s) that was(were) selected. For both the Change and SelectionChange events, you can limit the events to a particular range. These examples limit the event so that the code runs when one cell was selected, cell J10 was selected, and any cell in column A was selected:

Recursive Event Calls

Ooh, that sounds complicated. What this means is that your code calls events too. If you change a cell via code, the Worksheet_Change event will fire and that code will run. It's not a bad thing, unless you don't want it to happen.

The Application object has a property called EnableEvents which you can use to prevent events from firing while your code is running. If, for instance, you used the Worksheet_Change event to test a cell and, if the cell's value is invalid, to change the cell to something else, the event would fire again from within itself. When a procedure (sub or function) calls itself, it's called recursion. You could potentially call the Change event over and over until you run out of memory. Not good. You can wrap your code like this to make sure it doesn't happen.

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

```
    Application.EnableEvents = False
```

```
    If Target.Value <> 5 Then
```

```
        Target.Value = "Invalid"
```

```
    End If
```

```
    Application.EnableEvents = True
```

```
End Sub
```

Changing EnableEvents to False (then back to True at the end) prevents the Change event from being called from itself (when Target.Value is set to Invalid).

Automatically Run Excel Macros Upon Opening Excel Workbook/Files. Auto Run Excel VBA Macro Code

Auto Run Excel Macros Upon Open.

Excel has the ability to automatically run/execute any Excel macro code of our choosing when opening an Excel Workbook. This of course is providing that the user has chosen to **enable macros!** In the past (pre Excel 97) one would name a macro **Auto_Open** and Excel would automatically run/execute any code within this procedure. *This method, by the way, is still workable in Excel versions since 97 for backward compatibility.*

Workbook_Open Event

In Excel 97 Microsoft introduced what are called **Events**, one of these is the **Workbook_Open Event**. However, unlike the old Auto_Open macro, the Workbook_Open event is a procedure of the Workbook Object and as such, the Workbook_Open procedure **MUST** reside in the private module of the Workbook Object (***ThisWorkbook***). The Auto_Open macro on the other hand, must reside in a standard public module.

Accessing the Private Module of the Workbook Object

(ThisWorkbook)

There are at least 2 ways to gain access to the private module of the Workbook Object (ThisWorkbook)

1) While in Excel proper, right click on the Excel icon, top left next to **File** and choose **View Code**.

2) While in the VBE double click the Module called ***ThisWorkbook***, seen in the Project Explorer (**View>Project Explorer (Ctrl+R)**)

Once here, you can select "**Workbook**" from the Object drop down list, located in the top left of the module pane. After you have selected "**Workbook**" Excel will default to;

Private Sub Workbook_Open()

Note, in Excel 97 accessing ThisWorkbook would automatically add the Workbook_Open event.

How to use the Workbook_Open Event

There are basically 2 ways we can have Excel VBA macro code run/execute via this Workbook_Open event. See the 2 simple examples below;

Private Sub Workbook_Open()

MsgBox "Hi, thanks for opening me", vbInformation

End Sub

Private Sub Workbook_Open()

Run "MyMacro"

End Sub

In the second case, "**MyMacro**" **must** be the name of the procedure in any standard public module (**Insert>Module**).

If you wish to run/execute code that **also resides in the same private module** you cannot use the **Run statement**, only the procedure name. E.g.

Private Sub Workbook_Open()

MyMacro

End Sub

Private Sub MyMacro()

MsgBox "Hi, thanks for opening me", vbInformation

End Sub

Automatically Run Excel Macros When a Cell Changes/Enter Data. Worksheet Change Event

Auto Run Excel Macros Upon Data Entry.

Excel has the ability to automatically run/execute any Excel macro code of our choosing when data in any cell on a Worksheet changes. We can do this via the **Worksheet_Change** event. We can even specify which cell/cells must be changed before running our code.

Worksheet_Change Event

In Excel 97 Microsoft introduced what are called **Events**, one of these is the **Worksheet_Change Event**. The Worksheet_Change event is a procedure of the Worksheet Object and as such, the Worksheet_Change procedure **MUST** reside in the private module of the Worksheet Object.

Accessing the Private Module of the Worksheet Object

There are at least 2 ways to gain access to the private module of the Worksheet Object.

1) While in Excel proper, right click on the Worksheet name tab and choose **View Code**.

2) While in the VBE double click the Module called **Sheet* (Sheet*)**, seen in the Project Explorer (**View>Project Explorer (Ctrl+R)**) under Microsoft Excel Objects.

Once here, you can select "**Worksheet**" from the Object drop down list, located in the top left of the module pane. After you have selected "**Worksheet**" Excel will default to;

Private Sub Worksheet_Change(ByVal Target As Range)

*Note, in Excel 97 accessing any the Module called **Sheet* (Sheet*)** would automatically add the Workbook_Change event.*

How to use the Worksheet_Change Event

You will note in the: **Private Sub Worksheet_Change(ByVal Target As Range)** there is a predefined variable called **Target**. This represents the Range Object of the cell that **was changed** after pushing Enter. *It is **not** the cell you land in **after** pushing **Enter**!* It is the variable called **Target** that we can use to determine exactly which cell has changed. See the simple example below

Private Sub Worksheet_Change(ByVal Target As Range)

MsgBox "You just changed " & Target.Address

End Sub

Note how we use the **Address Property** of the Range Object (Target) to identify the cell.

Let's now see an example of how we can use the **Address Property** of the Range Object (Target) to only run code when a specific cell is changed. In the case below, we are going to multiply the new entry in **A1** by 2, but **only if a number** has been entered. We have also told Excel to do nothing (exit sub) if more than one cell has been changed, or the data in A1 has been deleted.

Private Sub Worksheet_Change(ByVal Target As Range)

```
'Do nothing if more than one cell is changed or content deleted  
If Target.Cells.Count > 1 Or IsEmpty(Target) Then Exit Sub
```

```
    If Target.Address = "$A$1" Then  
        'Ensure target is a number before multiplying by 2  
        If IsNumeric(Target) Then  
            'Stop any possible runtime errors and halting code  
            On Error Resume Next  
            'Turn off ALL events so the Target * 2 does not _  
            'put the code into a loop.  
            Application.EnableEvents = False  
            Target = Target * 2  
            'Turn events back on  
            Application.EnableEvents = True  
            'Allow run time errors again  
            On Error GoTo 0  
        End If  
    End If
```

End Sub

In the above code we have made use of **Application.EnableEvents** to ensure the multiplying of A1 by 2 does **NOT** cause the change event to run again and **get itself in a loop**.

If **want** the change event to run when/if more than one cell is changed, we would remove the **Target.Cells.Count > 1** and replace all occurrences of **Target** (but **not** ByVal Target as Range) to **Target(1,1)**. This then forces Excel to only consider the **active cell** should more than one cell be changed. If you are not aware, there can only ever be 1 active cell in any selection and that is always the cell you start your selection from.

Monitor More Than 1 Cell (Target)

It is often needed that we run code when any cell in a range of cells changes. For example, let's say each time any cell in the range **A1:A10** changes we need to

multiple by 2. In this case we would use an **If Not** Statement, with the **Intersect Method** to determine if the Target **is** within the range A1:A10. That is;

If Not Intersect(Target, Range("A1:A10")) Is Nothing Then

Example

Mask Excel Time Entries

Unlike Access, Excel does not allow for us to mask time and/or date entries. However, we can use the **Worksheet_Change Event** to achieve masked time entries. For the code to work times should be entered as 4 digits, e.g. 2244, 0130, 1325 will change to 22:44, 01:30 and 13:25 respectively. Note how the time mask is only applicable to the range A1:A100, but can be any range. You can change the display format (.NumberFormat = "[h]:mm") of the code to suit your needs.

To insert the code, right click on the Sheet name tab, select **View Code** and type the code below;

Private Sub Worksheet_Change(ByVal Target As Range)

Dim vVal

If Target.Cells.Count > 1 Then Exit Sub

If Intersect(Target, Range("A1:A100")) Is Nothing Then Exit Sub

With Target

vVal = Format(.Value, "0000")

If IsNumeric(vVal) And Len(vVal) = 4 Then

Application.EnableEvents = False

.Value = Left(vVal, 2) & ":" & Right(vVal, 2)

.NumberFormat = "[h]:mm"

End If

End With

Application.EnableEvents = True

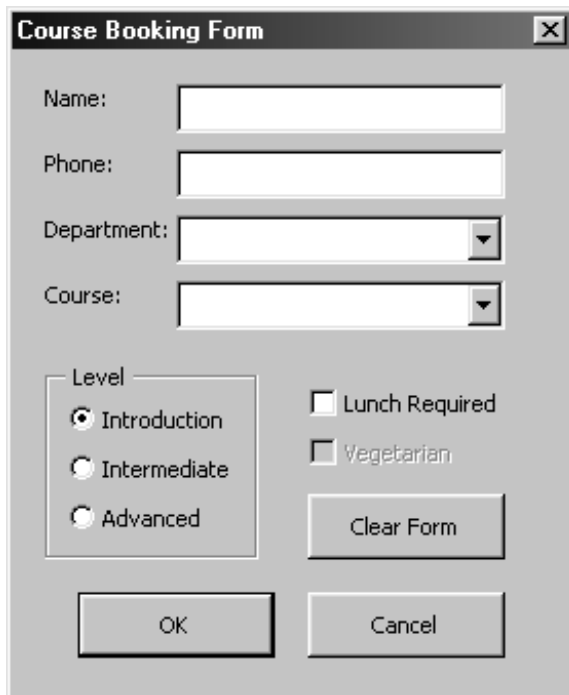
End Sub

UserForms

A UserForm [object](#) is a window or dialog box that makes up part of an application's user interface.

Creating an UserForm

The Course Booking Form

The screenshot shows a UserForm titled "Course Booking Form" with a standard Windows-style title bar (minimize, maximize, close buttons). The form contains several controls: four text boxes labeled "Name:", "Phone:", "Department:", and "Course:"; a group box labeled "Level" containing three radio buttons for "Introduction", "Intermediate", and "Advanced"; two check boxes labeled "Lunch Required" and "Vegetarian"; and three command buttons labeled "OK", "Cancel", and "Clear Form". The "Introduction" radio button is selected, and the "Vegetarian" check box is disabled (greyed out).

The **Course Booking Form** is a simple form illustrating the principles of UserForm design and the associated VBA coding.

It uses a selection of controls including text boxes, combo boxes, option buttons grouped in a frame, check boxes and command buttons.

When the user clicks the OK button their input is entered into the next available row on the worksheet.

Description of the Form:

There are two simple text boxes (Name: and Phone:) into which the user can type free text, and two combo boxes (Department and Course) that let the user to pick an item from the list.

There are three option buttons (Introduction, Intermediate and Advanced) grouped in a frame (Level) so that the user can choose only one of the options.

There are two check boxes (Lunch Required and Vegetarian) that, because they are not grouped in a frame, can both be chosen if required. However, if the person making the booking does not want lunch we do not need to know whether or not they are vegetarian. So, the Vegetarian check box is greyed-out until required. There are three command buttons (OK, Cancel and Clear Form) each of which performs a pre-defined function when clicked.

The Control Properties Settings:

Control	Type	Property	Setting
UserForm	UserForm	Name	frmCourseBooking
		Caption	Course Booking Form
Name	Text Box	Name	txtName
Phone	Text Box	Name	txtPhone
Department	Combo Box	Name	cboDepartment
Course	Combo Box	Name	cboCourse
Level	Frame	Name	fraLevel
		Caption	Level
Introduction	Option Button	Name	optIntroduction
Intermediate	Option Button	Name	optIntermediate
Advanced	Option Button	Name	optAdvanced
Lunch Required	Check Box	Name	chkLunch
Vegetarian	Check Box	Name	chkVegetarian
		Enabled	False
OK	Command Button	Name	cmdOk
		Caption	OK
		Default	True
Cancel	Command Button	Name	cmdCancel
		Caption	Cancel
		Cancel	True
Clear Form	Command Button	Name	cmdClearForm

Building the Form

If you want to build the form yourself, simply copy the layout shown in the illustration above. Follow the steps below:

1. Open the workbook that you want the form to belong in (UserForms like macros have to be attached to a workbook) and switch to the Visual Basic Editor.
2. In the Visual Basic Editor click the **Insert UserForm** button (or go to **Insert > UserForm**).
3. If the toolbox does not appear by itself (first click the form to make sure it isn't hiding) click the **Toolbox** button (or go to **View > Toolbox**).
4. To place a control on the form click the appropriate button on the toolbox then click the form. Controls can be moved by dragging them by their edges, or resized by dragging the buttons around their perimeter.
5. To edit the properties of a control, make sure the chosen control is selected then make the appropriate changes in the **Properties** window. If you can't see the properties window go to **View > Properties Window**.
6. To remove a control from the form, select it and click the **Delete** key on your keyboard.

A UserForm will not actually do anything until the code that drives the form and its various controls is created. The next step is to write the code that drives the form itself.

Adding the Code: 1 Initialising the Form

Most forms need some kind of setting up when they open. This may be setting default values, making sure fields are empty, or building the lists of combo boxes. This process is called Initialising the Form and it is taken care of by a macro called UserForm_Initialize. Here's how to build the code to initialise the Course Booking Form:

1. To view the form's code window go to **View > Code** or click **F7**.
2. When the code window first opens it contains an empty **UserForm_Click()** procedure. Use the drop-down lists at the top of the code window to choose **UserForm** and **Initialize**. This will create the procedure you need. You can now delete the UserForm_Click() procedure.
3. Enter the following code into the procedure:

```
Private Sub UserForm_Initialize()  
    txtName.Value = ""  
    txtPhone.Value = ""  
    With cboDepartment  
        .AddItem "Sales"
```



```

        .AddItem "Marketing"
        .AddItem "Administration"
        .AddItem "Design"
        .AddItem "Advertising"
        .AddItem "Dispatch"
        .AddItem "Transportation"
    End With
    cboDepartment.Value = ""
    With cboCourse
        .AddItem "Access"
        .AddItem "Excel"
        .AddItem "PowerPoint"
        .AddItem "Word"
        .AddItem "FrontPage"
    End With
    cboCourse.Value = ""
    optIntroduction = True
    chkLunch = False
    chkVegetarian = False
    txtName.SetFocus
End Sub

```

How the Initialise Code Works:

The purpose of the UserForm_Initialize() procedure is to prepare the form for use, setting the default values for the various controls and creating the lists that the combo boxes will show.

These lines set the contents of the two text boxes to empty:

```

txtName.Value = ""
txtPhone.Value = ""

```

Next come the instructions for the combo boxes. First of all the contents of the list are specified, then the initial value of the combo box is set to empty.

```

With cboDepartment
    .AddItem "Sales"
    .AddItem "Marketing"
    (as many as necessary...)
End With
cboDepartment.Value = ""

```

If required an initial choice can be made from the option group, in this case:

```

optIntroduction = True

```

Both check boxes are set to False (i.e. no tick). Set to True if you want the check box to appear already ticked:

```

chkLunch = False

```

```
chkVegetarian = False
```

Finally, The focus is taken to the first text box. This places the users cursor in the text box so that they do not need to click the box before they start to type:

```
txtName.SetFocus
```

Adding the Code: 2 Making the Buttons Work

There are three command buttons on the form and each must be powered by its own procedure. Starting with the simple ones...

Coding the Cancel Button:

Earlier, we used the Properties Window to set the **Cancel** property of the Cancel button to *True*. When you set the Cancel property of a command button to True, this has the effect of "clicking" that button when the user presses the **Esc** key on their keyboard. But this alone will not cause anything to happen to the form. You need to create the code for the click event of the button that will, in this case, close the form. Here's how:

1. With the UserForm open for editing in the Visual Basic Editor, double-click the Cancel button. The form's code window opens with the **cmdCancel_Click()** procedure ready for editing.
2. The code for closing a form is very simple. Add a line of code to the procedure so it looks like this:

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

Coding the Clear Form Button:

Add a button to clear the form in case the user wanted to change their mind and reset everything, and to make it easier if they had several bookings to make at one time. All it has to do is run the Initialise procedure again. A macro can be told to run another macro (or series of macros if necessary) by using the **Call** keyword:

1. Double-click the Clear Form button. The form's code window opens with the **cmdClearForm_Click()** procedure ready for editing.
2. Add a line of code to the procedure so it looks like this:

```
Private Sub cmdClearForm_Click()  
    Call UserForm_Initialize  
End Sub
```

Coding the OK Button:

This is the piece of code that has to do the job of transferring the user's choices and text input on to the worksheet. When we set the Cancel button's Cancel property to True we also set the OK button's **Default** property to *True*. This has the effect of clicking the OK button when the user presses the **Enter** (or **Return**) key on their keyboard (provided they have not used their **Tab** key to tab to another button). Here's the code to make the button work:

1. Double-click the OK button. The form's code window opens with the **cmdOK_Click()** procedure ready for editing.
2. Edit the procedure to add the following code:

```
Private Sub cmdOK_Click()  
    ActiveWorkbook.Sheets("Course Bookings").Activate  
    Range("A1").Select  
    Do  
        If IsEmpty(ActiveCell) = False Then  
            ActiveCell.Offset(1, 0).Select  
        End If  
        Loop Until IsEmpty(ActiveCell) = True  
        ActiveCell.Value = txtName.Value  
        ActiveCell.Offset(0, 1) = txtPhone.Value  
        ActiveCell.Offset(0, 2) = cboDepartment.Value  
        ActiveCell.Offset(0, 3) = cboCourse.Value  
        If optIntroduction = True Then  
            ActiveCell.Offset(0, 4).Value = "Intro"  
        ElseIf optIntermediate = True Then  
            ActiveCell.Offset(0, 4).Value = "Intermed"  
        Else  
            ActiveCell.Offset(0, 4).Value = "Adv"  
        End If  
        If chkLunch = True Then  
            ActiveCell.Offset(0, 5).Value = "Yes"  
        Else  
            ActiveCell.Offset(0, 5).Value = "No"  
        End If  
        If chkVegetarian = True Then  
            ActiveCell.Offset(0, 6).Value = "Yes"  
        Else  
            If chkLunch = False Then  
                ActiveCell.Offset(0, 6).Value = ""  
            Else  
                ActiveCell.Offset(0, 6).Value = "No"  
            End If  
        End If  
        Range("A1").Select  
    End Sub
```

How the CmdOK_Click code works:

The first two lines make sure that the correct workbook is active and moves the selection to cell A1:

```
ActiveWorkbook.Sheets("Course Bookings").Activate  
Range("A1").Select
```

The next few lines moves the selection down the worksheet until it finds an empty cell:

```
Do  
If IsEmpty(ActiveCell) = False Then  
    ActiveCell.Offset(1, 0).Select  
End If  
Loop Until IsEmpty(ActiveCell) = True
```

The next four lines start to write the contents of the form on to the worksheet, using the active cell (which is in column A) as a reference and moving *along the row* a cell at a time:

```
ActiveCell.Value = txtName.Value  
ActiveCell.Offset(0, 1) = txtPhone.Value  
ActiveCell.Offset(0, 2) = cboDepartment.Value  
ActiveCell.Offset(0, 3) = cboCourse.Value
```

Now we come to the option buttons. These have been placed in a frame on the form so the user can choose only one. An IF statement is used to instruct Excel what to do for each option:

```
If optIntroduction = True Then  
    ActiveCell.Offset(0, 4).Value = "Intro"  
Elseif optIntermediate = True Then  
    ActiveCell.Offset(0, 4).Value = "Intermed"  
Else  
    ActiveCell.Offset(0, 4).Value = "Adv"  
End If
```

VBA IF statements are much easier to manage than Excel's IF function. You can have as many options as you want, just insert an additional **Elseif** for each one. If there were only two options, you wouldn't need the **Elseif**, just the **If** and **Else** would suffice (don't forget - they all need an **End If**).

There is another IF statement for each check box. For the Lunch Required check box, a tick in the box means "Yes" the person requires lunch, and no tick means "No" they don't.

```
If chkLunch = True Then  
    ActiveCell.Offset(0, 5).Value = "Yes"  
Else  
    ActiveCell.Offset(0, 5).Value = "No"  
End If
```

We could use a similar IF statement for the Vegetarian check box, but if the person does not require lunch it is irrelevant whether or not they are vegetarian. In any case, it would be wrong to assume that they were not vegetarian simply because

they did not require lunch. The IF statement therefore contains a second, nested if statement:

```
If chkVegetarian = True Then
    ActiveCell.Offset(0, 6).Value = "Yes"
Else
    If chkLunch = False Then
        ActiveCell.Offset(0, 6).Value = ""
    Else
        ActiveCell.Offset(0, 6).Value = "No"
    End If
End If
```

A tick in the box means "Yes" the person is vegetarian. If there is no tick in the box, the nested IF statement looks at the Lunch Required check box. If the Lunch Required check box has a tick in it then no tick in the Vegetarian check box means that the person is not vegetarian so it inserts "No" into the cell. However, if the Lunch Required check box does not have a tick in it, then we do not know whether or not the person is vegetarian (it doesn't matter anyway) so the cell is left blank ("").

Finally the selection is taken back to the beginning of the worksheet, ready for the next entry:

```
Range("A1").Select
```

Adding the Code 3: Manipulating the Form

Finally, an example of how the controls on a form can be manipulated whilst it is in use. When the control properties were set, the **Enabled** property of the Vegetarian check box was set to *False*. When a control is not enabled the *user* cannot enter a value into it, although it can hold a value that was there already, and VBA can add, remove or change the value.

We don't need to know whether or not the person is vegetarian (even if they are!) if they aren't ordering lunch. So, the Vegetarian check box remains disabled unless a tick is placed in the Lunch Required check box. Then the user is free to tick the Vegetarian check box if they want to. If they tick it we will know that they have answered "Yes" and if they don't we know they have answered "No".

We can toggle the **Enabled** property from *False* to *True* by having a procedure that runs automatically whenever the value of the Lunch Required check box changes. Fortunately, more controls have a *Change* procedure and the one we use here is **chkLunch_Change()**. We'll use this to enable the Vegetarian check box when the Lunch Required check box is ticked, and disable it when the Lunch Required check box is not ticked.

There's just one more thing we need to do. Supposing someone ticked the Lunch Required check box, and also ticked the Vegetarian check box. Then they changed their mind and removed the tick from the Lunch Required check box. The Vegetarian check box would be disabled but the tick that was put in earlier would remain.

An extra line of code can make sure the tick is removed when the box is disabled. Here's the whole thing:

```
Private Sub chkLunch_Change()  
    If chkLunch = True Then  
        chkVegetarian.Enabled = True  
    Else  
        chkVegetarian.Enabled = False  
        chkVegetarian = False  
    End If  
End Sub
```

Opening the Form

The form is now ready for use so it needs to be opened with a simple macro. That can be attached to a custom toolbar button, a command button drawn on the worksheet, or any graphic (right click the graphic and choose **Assign Macro**). If necessary, create a new module for the workbook and add this procedure:

```
Sub OpenCourseBookingForm()  
    frmCourseBooking.Show  
End Sub
```