Welcome to Excel VBA Programming

VBA stands for Visual Basic for Applications (the application being of course Excel) and is the technology and tools used to program and automate Microsoft Excel.

It's not only used just within the framework of Microsoft Excel but other applications too including Microsoft Access, Microsoft Word, Microsoft Outlook to name but a few.

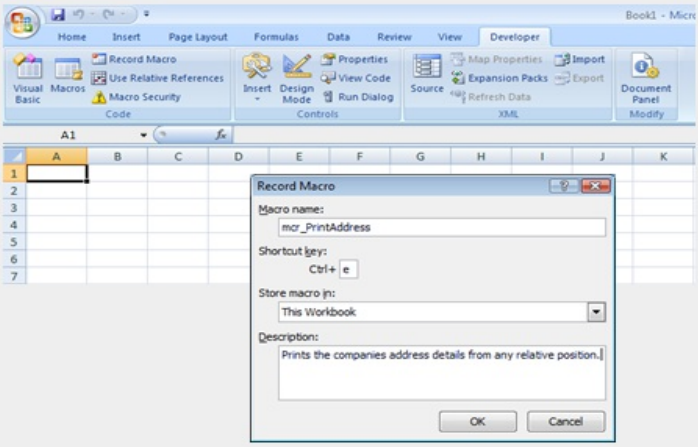
It has the power to communicate with other applications beyond the Microsoft range and even the Microsoft Windows operating system across other platforms.

So, learning the principles of VBA using Excel as the tool environment will stand you in good stead for the other applications should you wish to program and code them in the future.

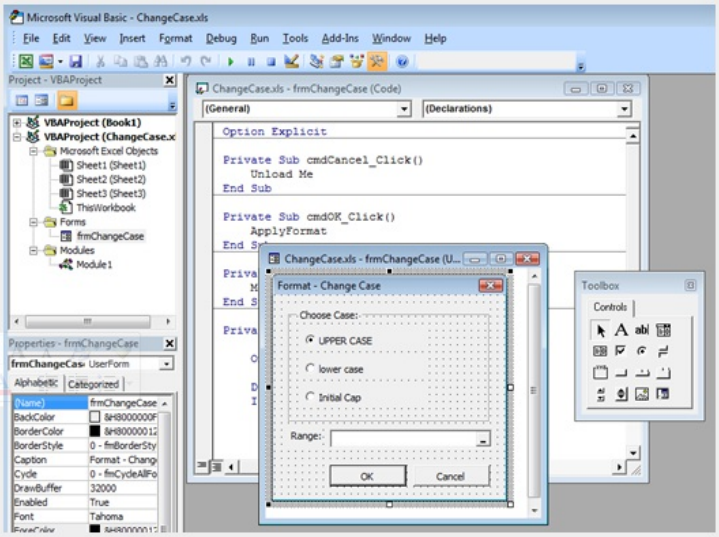
The only difference between other applications when wanting to use VBA will simply be learning to load and work with different libraries (which I intend to teach you in due course throughout this free online reference guide).

This site is used in conjunction with my classroom instructor lead teaching (for my students attending an Excel VBA course) and is intended as a reference guide only. But if you have attended or taught Excel VBA yourself, this will help you too.

You will start by learning to record, edit and manage macros in Excel capturing the VBA code automatically giving you the confidence and basic knowledge to the VBA code language itself.



At some point you will want to get down to learning about the power of VBA using Excel as the environment tool to test the code. This will introduce you to the programming conventions, concepts and techniques that simply go beyond the scope of the Excel Macro Recorder tool.



There is a lot more VBA code that can not be recorded which include logic testing, iteration, interactive macros, creating functions and assigning variables.

I will gently ease you into learning VBA code smoothing out the steep learning curve as much as

Note: Excel Version illustrated throughout this website is based on 2007 and therefore some of the commands may vary on previous versions.

possible.

I hope you find this resource helpful, Thank you!
Ben Beitler

Want to teach yourself Access? Free online guide at About Access Databases

Looking at the code

VBA code is stored in a module which is part of the Excel workbook but is viewed via the Visual Basic Editor (VBE) interface.

1. Click the **Macro** icon from the **Developer** tab.
2. Select the macro you wish to view.
3. Click on the **Edit** button.

Now look at the differences between the absolute and the relative macros.

Absolute Macro

```
Sub Absolute()
'
' Absolute Macro
' Macro to place company name and address into cells A1:A5
'
' Keyboard Shortcut: Ctrl+e
'
    Range("A1").Select
    ActiveCell.FormulaR1C1 = "ABC Ltd"
    Range("A2").Select
    ActiveCell.FormulaR1C1 = "ABC House"
    Range("A3").Select
    ActiveCell.FormulaR1C1 = "50 Fleet Street"
    Range("A4").Select
    ActiveCell.FormulaR1C1 = "London"
    Range("A5").Select
    ActiveCell.FormulaR1C1 = "EC4A 5TE"
    Range("A1").Select
End Sub
```

`Range("A1").Select`

In plain English, this means, "click on the cell A1".

`ActiveCell.FormulaR1C1 = "ABC Ltd"`

In plain English, this means "enter the text ABC Ltd into the active cell".

Relative Macro

```
Sub Relative()
'
' Relative Macro
' Macro to place company name and address into the active cell
'
' Keyboard Shortcut: Ctrl+h
'
    ActiveCell.Select
    ActiveCell.FormulaR1C1 = "ABC Ltd"
    ActiveCell.Offset(1, 0).Range("A1").Select
    ActiveCell.FormulaR1C1 = "ABC House"
    ActiveCell.Offset(1, 0).Range("A1").Select
    ActiveCell.FormulaR1C1 = "50 Fleet Street"
    ActiveCell.Offset(1, 0).Range("A1").Select
    ActiveCell.FormulaR1C1 = "London"
    ActiveCell.Offset(1, 0).Range("A1").Select
    ActiveCell.FormulaR1C1 = "EC4A 5TE"
    ActiveCell.Offset(-4, 0).Range("A1").Select
End Sub
```

`Activecell.Select`

In plain English, this means, "click on the active cell".

`ActiveCell.FormulaR1C1 = "ABC Ltd"`

In plain English, this means "enter the text ABC Ltd into the active cell".

Tip: Alt + F11 function keys switches between Excel and VBE window.

VBA Keywords: ActiveCell, Range, Selection, OffSet.

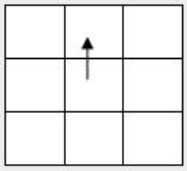
Note: For further details and other members, see Range/Selection objects.

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

In plain English, this means, "select the cell one row down, but stay in the same column".

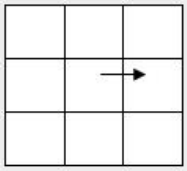
OffSet (Keyword)

Offset is a command used in relative macros, which allows you to select a particular cell in relation to the active cell.



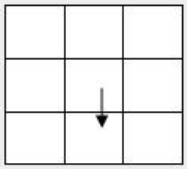
`Offset(-1,0)`

Move one row up, but stay in the same column.



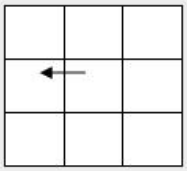
`Offset(0,1)`

Stay in the same row, but move one column to the right.



`Offset(1,0)`

Move one row down, but stay in the same column.



`Offset(0,-1)`

Stay in the same row, but move one column to the left.

Unnecessary Code

As previously mentioned, recording macros does create a lot of unnecessary code.

In this example, the **`Range("A1")`** that appears in rows 3, 5, 7, 9 and 11 of the relative macro is unnecessary and can be removed.

Next Topic: Ways of running a macro

Ways of running macros

Most users will automatically run a macro from either button on the **Quick Access toolbar** (known as **Toolbars** on previous versions) or via the conventional **Macro** dialog box.

The following is a list of ways to run a macro:

1. The **Macro** dialog box.
   Version 2003 (or earlier) - Click on the **Tools** menu, select **Macro** and choose **Macros**.
   Version 2007 (or later) - From the **Developer** tab, click the **Macro** icon.

   Or the shortcut key to all versions is **Alt + F8**.

   Select the Macro you wish to run and click on the **Run** button.

2. Using a shortcut key as assigned, i.e. **Ctrl + e**.

3. From a **Button** on the worksheet.

4. From an icon **Button** on the Quick Access toolbar.
   Previous versions uses Toolbars.

5. From the **Ribbon Bar** (though requires some XML knowledge).
   Previous versions uses a menu item from the menu bar.

6. From another type of object, e.g. **Chart** or **Graphic** image.

7. From a **Control** drawn on the worksheet, e.g. **Combo Box**.

8. A worksheet or workbook event, e.g. when a workbook is opened
   This is maintained in the Visual Basic Editor (VBE) interface.

The last item is a great way to get Excel to run your code without any user intervention as it's uses Excel's own processes to trigger the macro.

Most users will not be aware that Excel constantly listens for events to happen but do not see any physical results until they learn to manipulate the event handlers provided.

There are many events from a **Control** (i.e. Button) to opening (Open) and closing (Close) workbooks.

Think about how **Data Validation** and **Conditional Formatting** work in Excel worksheets. They respond to when a user has clicked the **Enter** key (Worksheet_Change) to trigger the two utilities. - *More on this later...*

Next Topic: Where macros are stored

Tip: Alt + F8 function keys displays the **Macro** dialog box to run loaded macros.

VBA Keywords: CommandBars, Worksheet_Change (event), Workbook_Open (event), Workbook_Close (event).

Where macros are stored

There are three locations to choose from which affect the scope and availability of a macro:

1. This Workbook
2. New Workbook
3. Personal Workbook

1. This workbook will store macros in the current workbook, which recorded the macro and is said to be a *local* macro.

That will mean, every time users want to run the macro, they will first have to load the file and then execute the macro.

2. New Workbook will store the macro to an unsaved new file and is generally used for distributing to other users which they would need to load and run manually. Treat this as the same scope for that of **This Workbook** and is deemed as a *local* macro too.

3. Personal Workbook is a specially reserved named file which is generated (*first time around*) automatically to store the recorded macros.

The name given to this file (*which is still treated like any other Excel file*) is **Personal.xls/xlsm/xlsb**.

This file is hidden by default as it is not intended to be used as a normal spreadsheet.

The location of the file is important and must reside in the ***XLSTART*** folder of where the user's profile or Excel application is installed.

This special path responds to the loading event of Excel and loads any file stored in this folder. Therefore, macros which are stored in the file in the path ***XLSTART*** will open too.

Macros that require a *global* use are stored in this type of file (*i.e. User Defined functions*).

The full path and file for the default installation of Excel would look something like the following:

Excel 2007 (Windows Vista)

C:\Users\Ben\AppData\Roaming\Microsoft\Excel\XLSTART\PERSONAL.XLSB

Excel 2003 (Windows XP)

C:\Documents & Settings\Ben\Application Data\Microsoft\Excel\XLSTART\PERSONAL.XLS

Next Topic: Reasons to write macros

Note: The full path to locate folders to the PERSONAL file may be hidden and will require some modification in Windows Explorer to view them.

Also check with your IT administrator if this has been restricted.

Reasons to write macros

Many experienced users (and developers) who have discovered macros and VBA tend to lean towards using code to automate Excel as much as possible.

However, there should be clear reasons as to why you would use a macro in the first place.

The following points will give poise for thought before utilising any macro within Excel:

1. To automate a repetitive operation.

If there is a pattern to your work which can be considered long and repetitive, using a macro will speed up how you process your tasks in Excel.

Consistency is key. If your requirement deviates from the standard procedures then don't expect the macro to run smoothly which is why you need to edit a macro by adding code that will ask questions of the routines trying to complete.

2. To automate a tricky task.

If your task is quite a lengthy procedure which may lead to user error then this would be another good reason to employ a macro.

In most cases, interactive macros will be key here (which of course can not be recorded) helping the user flow through multiple decision processes.

3. Help user access large blocks of data.

The amount of data that can be stored in Excel varies between versions. For instance in Excel 2003 you have 65,536 rows compared to Excel 2007 which contains 1,048,576. *Remember, this is just one worksheet!*

Managing large data sets can be clumsy and time consuming when carried out by the user manually and a macro can be as short as simply capturing the range should you wish to format, edit or print information as a simple task.

4. Perform math's not supported by menu commands or functions.

Though Excel provides a wealth of calculating functions for your convenience, it will be fair to say that not every mathematical process will have been provided for.

General users may not have the required knowledge to write complex formulae especially if this is used on a regular basis.

Creating your own functions therefore (User Defined Functions) is a macro which can not be recorded at all but provides a wrapper for general users to treat as a regular Excel function.

5. Environmental macros

What I call *environmental* macros are macros which are very short but simulate Excel commands that I wish to customise normally by attaching a keyboard shortcut to it.

Some Excel commands do not have keyboard shortcuts and each individual user will have their own working habits which they will typically custom build Excel accordingly.

It can be as simple as clearing all attributes (contents, formats and comments) to a range of cells not just deleting the contents only (DEL key).

6. Protect data from user errors.

Instead of allowing users to gain direct access to your data, protecting it via a macro will give you better control in how users can manage your Excel processes.

Viewing the data maybe required in most tasks and allowing users to protect and unprotect ranges, worksheets and workbooks (with or without passwords) to edit and format information can be controlled in decision making macros.

User form can also provide a level of protection and require macro VBA code too.

Do not *'re-invent the wheel'* in other words learn as much as possible about the general features of Excel to rule out if you really need to have a macro at all.

You may find a feature in Excel can do all your tasks in one simple step and you would have wasted time creating a macro in the first place.

Next Topic: writing macros

Note: If you intend to write macros for external organisations, be careful to check with the recipient that they can use macros at all as some firms disable macros altogether. Additionally, security can be a problem too - seek you IT administrator.

Writing macros

Start by writing a task list of the step you wish to capture and use this as your checklist to help cover all actions required and in the correct order.

Define your task

1. Define the task you wish to program.
2. The overall task must be broken down into smaller tasks.
3. The program consists of a set of instructions or code, which the computer will follow.
4. The order in which you place these statements is very important.

Layout of procedures

```
Declaration Area

Procedure Starts Name()
      1st line of Statements 'Comments
      2nd line of Statements 'Comments
      ..........
End of Procedure
```

The blue text represents the procedure starting and ending signatures

The green text represents the narrative/comments for documentation purposes which are excluded from the procedure.

All procedures must have starting signature and ending signature.

Pseudo code

Write the program out in plain English to explain what is going to happen.

```
sub formatting()
bold
italic
underline
end sub
```

Calling a procedure (formatting)

```
sub start()
select cells A2:A10
formatting
select cells B1:G1
formatting
end sub
```

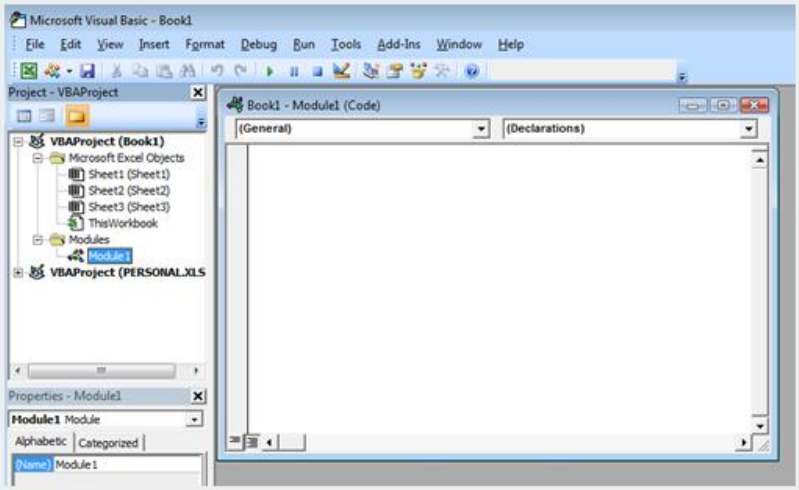
Writing macros from scratch

The following macro will select Sheet 1 and type January into cell A1 and 100 into cell A2.

1. Create a new blank workbook.
2. Click on the **Developer** tab, click **Visual Basic** icon.
3. In the VBE window, click on the **Insert** menu and select **Module**.

Tip: Alt + F11 function keys switches between Excel and VBE window.

VBA Keywords: ActiveCell, Range, Worksheets, Select.



Pseudo Code

```
Sub january()
Select Sheet1
Select the cell A1
Type January
Select the cell A2
Type 100
End Sub
```

VBA Code

```
Sub january()
Worksheets("Sheet1").Select
Range("A1").Select
ActiveCell.Value = "January"
Range("A2").Select
ActiveCell.Value = 100
End Sub
```

4. Type the VBA code into the module and test.

Note that the text 'January' is entered with speech (double-quote) marks, as it is a piece of text, where as the number 100 is entered without.

Next Topic: Procedure types

Want to teach yourself Access? Free online guide at About Access Databases

Different Types of Procedures

VBA Keywords: MsgBox, Call, Exit Sub, Exit Function.

There are three types of procedures:

1. **Sub** - Standard sub routine
2. **Function** - a routine that returns an answer
3. **Property** - reserved for *Class Modules*

The third item is not discussed in this topic as it is deemed advanced VBA.

Sub Procedure

This is the most commonly used procedure that a recorded and edited macro typically uses.

It executes code line by line in order, carrying out a series of actions and/or calculations.

The signature for this type of procedure is:

```
Sub NameOfProcedure([Arguments])
      1st line of executed code 'Comments
      2nd line of executed code 'Comments
      ..........
End Sub
```

The '*Arguments*' element is optional which can be **explicit** or **implicit**. This allows values and /or references to be passed into the calling procedure and handled as a variable.

When recording a macro, no arguments are used and the parenthesis for the named procedure remains empty.

If you create a procedure intended as a macro in Excel, users must not specify any arguments.

Sub procedures can be recursive meaning that branching to another procedure is permitted which then returns back to the main calling procedure.

Calling another procedure can include the **Call** statement followed by the name of the procedure with optional arguments. If arguments are used, users must use parenthesis around the argument list.

Example of the CALL statement

```
'Procedure to be called with a single
argument explicitly 'declared as a string
Sub MyMessage(strText As String)
      MsgBox strText
End Sub
```

(Click here for an understanding of the **MsgBox** statement)

Correct

```
'Test the calling procedure
Sub TestMessage()
      Call MyMessage("It worked!")
End Sub
```

Incorrect - must use the parenthesis

```
'Test the calling procedure
Sub TestMessage()
      Call MyMessage "Did it work?"
End Sub
```

Correct (alternative) - No Call keyword used & no parenthesis therefore required.

```
'Test the calling procedure
Sub TestMessage()
      MyMessage "It worked!"
```

```
End Sub
```

A procedure can be prematurely terminated, placed before the '***End Sub***' statement by using the '***Exit Sub***' statement.

```
'This procedure will terminate after part A and never run part B.
Sub TerminateNow()
      Code part A here...
     Exit Sub
      Code part B here....
End Sub
```

Function Procedure

The main difference between a **Sub** and **Function** procedure is that a **Function** procedure carries out a procedure and will return an answer whereas a **Sub** procedure carries out the procedure without an answer.

A simple analogy of a **Function** procedure compared to that of a **Sub** procedure could be illustrated using two example features of Excel:

- *File*, *Save* is an action and does not return the answer – *Sub Procedure*.
- The *Sum* function calculates the range(s) and returns the answer – *Function Procedure*.

The signature for this type of procedure is:

```
Function NameOfProdedure([Arguments]) [As Type]
     Code is executed here
     NameOfProcedure = Answer of the above code executed
End Function
```

The **Arguments** element is optional which can be **explicit** or **implicit**. This allows values and /or references to be passed into the calling procedure and handled as a variable.

The optional **Type** attribute can be used to make the function explicit. Without a type declared, the function is implicit (*As Variant*).

The last line before the **End Function** signature uses the name of the procedure to return the expression (*or answer*) of the function.

Users cannot define a function inside another function, sub or even property procedures.

This type of procedure can be called in a module by a **Sub** procedure or executed as a user defined function on a worksheet in Excel.

A procedure can be prematurely terminated, placed before the ***End Function*** statement by using the **Exit Function** statement. This acts and responds in the same way as described in the previous section (*Sub Procedures*).

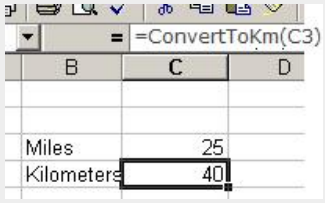
An example of a Function procedure:

```
'This function calculates the distance of miles into kilometres.
Function ConvertToKm(dblMiles As Double) As Double
      ConvertToKm = dblMiles * 1.6
End Function
```

A **Sub** procedure that uses of the above function:

```
'Using the above function that must use parenthesis.
Sub CarDistance
      MsgBox ConvertToKm(25)
End Sub
```

In Excel, this function can also be used (known as a **User Defined Function - UDF**)

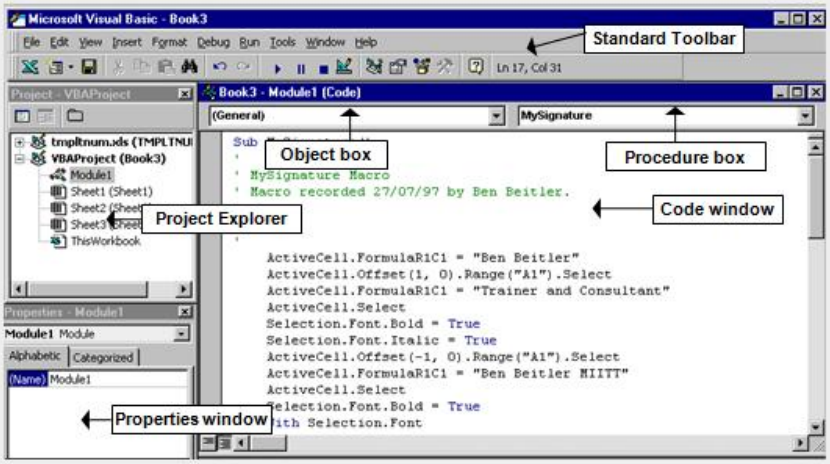


Click on this link for more information on user defined functions.

Next Topic: Visual Basic editor (VBE)

Visual Basic Editor

All macros can be edited and created from the **V**isual **B**asic **E**ditor (VBE) application as mentioned earlier.



Standard Toolbar

Contains all the basic buttons to this window like save, switching to Excel and hide/show other windows. There are other Toolbars available; Edit, Debug, User Form and Toolbox.

Object Box

This displays the name of the selected object chosen from the drop down box.

Procedure Box

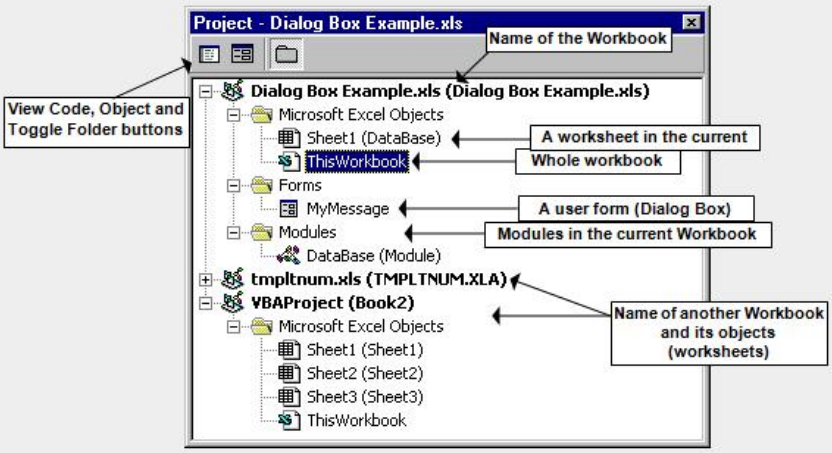
This displays the name of the procedure or event of the object (*i.e. worksheet*).

Code Window (Module)

This is where you maintain the VBA code. One or more *sub* and *function* procedures are stored in this view and users manage macros across one or more **modules**.

Project Explorer

All the code associated with a workbook is stored in the '*Project*' window. This is automatically saved with the Workbook.



Like a workbook, the '*Project Explorer*' contains all associated objects, which include worksheets, user forms and modules.

Macros are stored in either the sheet object or module object. Consider using the module object to

Tip: Use **F5** function key to run a macro from the VB Editor.

store macros for general use in that workbook rather than a specific macro for a specific sheet.

By double clicking on an object or clicking the view code button at the top left corner of the '*Project*' window, displays the objects code (*macros associated*).

Properties Window

Properties are characteristics of the selected object.
This window allows you change these characteristics to a worksheet, workbook and user form.

This above window is task sensitive and therefore changes as you click from one control to another.

Edit Toolbar

Select **View**, **Toolbars** and select the Toolbar required.

Edit Toolbar

1. **Lists Properties/Methods** box in a code window. This is task sensitive as it shows properties and methods to active keywords.
2. **List Constants**.
3. **Quick Info** displays a label for the active keyword or variable.
4. **Parameter Info** displays the syntax label of known keywords.
5. **Complete word** displays a scroll list box of keywords and completes the beginning of known types keywords.
6. **Indent** tabs once to the right.
7. **Outdent** tabs once to the left.
8. **Toggle Breakpoint** allows marking a line of code at which point a macro will stop.
9. **Comment Block** 'rem' the line (*put an apostrophe at the beginning of the line*).
10. **Uncomment Block** removes the 'rem' line.
11. **Toggle Bookmark** marks with a blue marker a piece of code so that scrolling between code lines is quick and simple.
12. **Next Bookmark** moves to the next bookmark.
13. **Previous Bookmark** moves to the previous bookmark.
14. **Clear All Bookmarks** clears all bookmarks.

There other toolbars that you may need to review and can be found via the **View** menu.

Next Topic: Rules & conventions

Rules & Conventions

VBA Keywords: If...Then, MsgBox, vbNewLine.

It is not mandatory to follow Microsoft's rules and conventions regarding name spaces and prefixes. Users could always introduce their own standards, rules and conventions, which will help other users who may need to maintain processes within the organisation.

The following is a guideline to perhaps how authors and users alike could manage the code.

Naming macros, procedures and variables should be meaningful to the process to help clarify the task in hand.

Do not name a macro or procedure '*MyProcedure1'* or '*Macro1'* but keep it user friendly to help described the process.

Users can use more than one word provided there are no spaces or invalid characters used (*operators*). When using more than 'one-worded' procedures, consider initially capping each word to help see the name of the procedure clearly.

For example, **Sub *openallorders*()** would be better shown as **Sub *OpenAllOrders*()**.

Variables such as **X = 10** would be more helpful if **X** was named to be more meaningful to the intended process i.e. the number of years and could therefore be shown as **NumberOfYears = 10** or **NoYears = 10**.

> Variables and naming conventions are covered elsewhere in this guide – see *Variables and Constants*.

Do not use keywords when naming procedures or variables, as this will cause potential conflicts and errors.

Indentation

Code should be clearly positioned in a module. Use the tab key to indent logical blocks of code. Users can use as many indentations to emphasis new blocks of code (*as nested*) if required to show where a block starts and ends. This will help when browsing for long portions of code.

```
Sub MarcoName()
    First line is indented (1 tab) after the signature.

    New block starts here (1 tab).
        Code for the block is entered here (2 tabs).
    Block ends here (1 tab).

    New block starts here (1 tab).
        Code for the block is entered here (2 tabs).
        Second block of code starts here (2 tabs).
            Code for second block is entered here (3 tabs).
        Second Block ends here (2 tabs).
    Block ends here (1 tab).
End Sub
```

Comments

Commenting your code is important to the author and other users who may need to maintain code fragments. By default, commented lines are coloured green when text is typed following an apostrophe (') or the keyword '**Rem**' (*remark*).

As part of the opening signature (*either before or after the signature*), a brief description of the procedure along with a date and name of the author should be documented.

For example:

```
Sub ProcessInvoice()
'*************************************************************
' This procedure will validate all entries to the new invoice.
' It will calculate sub total and tax values and post it to the
' data store. ' It will print and close the invoice.
' Author: Ben Beitler
' Date Created: 12/04/2010
' Date Modified: 20/04/2010
'*************************************************************
      executed code is entered here
End Sub
```

Comments can appear anyway in the module provided it is remarked correctly as this type of text is

ignored during code execution. Comments should also be added to unusual or difficult lines of code (*known as inline comments*) to help explain the nature of the action.

For example:

```
Sub SomeProcedure()
' Comments here.

      [executed code is entered here...]
      ' This block is to validate if the field had been
      ' completed using my own custom function
      ' ValidateEntry(field).
      If mField = ValidateEntry(txtDate) Then
           [executed code is continued here...]
      intCounter = 1 'Set the flag back to 1 in order to 'restart counter.

      [executed code is entered here...]
End Sub
```

Line Breaks

Generally code should not be written beyond the screen/page width as it becomes cumbersome to work with, as users would have to scroll left and right unnecessarily.

Consider introducing a line break for single line code that extends beyond the page width by using the characters '**spacebar**' and a '**underscore**' (_).

For example:

```
Sub MessageTestLineBreak()
      MsgBox "This is the first Line." & vbNewLine & _
             "This is the second Line.", _
     vbInformation, "Message Box Test"
End Sub
```

Microsoft produced various documents on this subject. For a full list, check out

http://msdn.microsoft.com/library and search for '*Code Conventions*"

More information about conventions regarding variables are covered later in this manual – see Variables & Constants.

Next Topic: Excel objects

Excel Objects

VBA Keywords: Application, ActiveSheet, ActiveWorkbook, ActivePrinter, ActiveCell, ActiveChart, ActiveWindow, CreateObject, Workbooks, Worksheets, Name, MsgBox, SaveAs, Count and Add.

There are many categories (classes) of Excel objects that can be controlled in VBA. In fact, nearly all objects can be controlled in VBA that users manipulate in the Excel interface. VBA can also control more than the Excel interface provides which is one of the key reasons why '*power users*' use VBA!

The Object hierarchy provides the levels of various key objects ranging from the cell ranges (the lowest level) through to the application itself (the highest level).

This section focuses on the **Application**, **WorkBook(s)**, **Worksheet(s)** and **ActiveSheet/Workbook** objects (see Range & Selection objects for more extended information).

Application object

The word **Application** refers to the host (in this case Excel) and is deemed the top level object.
(Note: VBA can communicate beyond Excel and technically this is not the top level as you have the ability to code to Microsoft Office (Word, PowerPoint etc) and to other applications including the operating system).

Use this object as the entry point (the gateway) to the Excel object model and is implicit which means that you can omit this keyword in your code as it's the default. The following two VBA commands do the same thing:

```
Application.ActiveSheet.Name = "January"

ActiveSheet.Name = "January"
```

The first example included the **Application** object keyword (as explicit) and the second one excluded (as implicit) it but produced the same result.

You only need to use this keyword if you are coding with other applications (that is not Excel) or wish to communicate to Excel from another application's environment (i.e. Microsoft Word). You will need to learn about object variables and set application objects to Excel.

The following code snippet creates an Excel object from outside of Excel (which uses VBA too) and opens a workbook called "Sales.xlsx":

```
Sub OpenExcelWorkbook()
      Dim xl As Object
      Set xl = CreateObject("Excel.Sheet")
      xl.Application.WorkBooks.Open("Sales.xlsx")
      'executed code continues...

End Sub
```

ActiveWorkbook and Workbooks objects

This object appears below the **Application** object along with other key objects including **Chart** and **Pivot Table** and control the tasks for any workbook from creating, opening, printing to saving and closing documents.

The singular keyword **Workbook** refers to the current or a single file you wish to control compared with the plural keyword **Workbooks** which is the collection of one or more documents you wish to control

Use the **Workbook** object referred in code as **ActiveWorkbook** to open, save, print, close and manipulate the documents attributes as required.

```
Sub WorkBookNameExample()
      MsgBox "Current workbook is " & ActiveWorkbook.Name
End Sub
```

Save a copy of the current workbook:

```
Sub SaveAsWorkBookExample1()
      ActiveWorkbook.SaveAs "VBA Workbook.xlsx"
End Sub
```

The above can also be expressed as follows:

```
Sub SaveAsWorkBookExample2()
      Workbooks(1).SaveAs "VBA Workbook.xlsx"
End Sub
```

Using the **Workbooks** keyword which is a collection of current workbooks, you can provide an index number (starting at 1 for the first document and incrementing by 1 for each open document) to execute code using the same identifiers as **ActiveWorkbook** object.

How many workbooks are currently open?

```
Sub WorkBookCount()
      MsgBox "There are currently " & Workbooks.Count & _
                                        " workbook(s) open"
End Sub
```

The **Workbooks** object doesn't have any parenthesis and an index number reference when dealing with a collection of many documents.

(Note: the above will also count all open and hidden documents).

ActiveSheet and Worksheets objects

Most of the time, you will work with this object along the range object as the normal practice is worksheet management in a workbook when working with the Excel interface.

Again, the singular **Worksheet** object referred as **ActiveWorkSheet** controls the current or single worksheet objects including its name. The plural keyword **Worksheets** refers to one or more worksheets in a workbook which allows you to manipulate a collection of worksheets in one go.

Name a worksheet:

```
Sub RenameWorksheetExample1()
      ActiveWorkSheet.Name = "January"
End Sub
```

or use

```
Sub RenameWorksheetExample2()
      WorkSheets(1).Name = "January"
End Sub
```

assuming the first worksheet is to be renamed.

Insert a new worksheet and place it at the end of the current worksheets:

```
Sub InsertWorksheet1()
      Worksheets.Add After:=Worksheets(Worksheets.Count)
End Sub
```

or it can shortened using the **Sheets** keyword instead:

```
Sub InsertWorksheet2()
      Sheets.Add After:=Sheets(Sheets.Count)
End Sub
```

(Note: Have you noticed when adding a new worksheet via Excel interface how it always inserts it to the left of the active sheet!).

'Active' objects

Within the **Application** object you have other properties which act as shortcuts (*Globals*) to the main objects directly below it. These include **ActiveCell**, **ActiveChart**, **ActivePrinter**, **ActiveSheet**, **ActiveWindow** and **ActiveWorkbook**.

You use the above keywords as a direct implicit reference to the singular active object in the same way (as in the above already illustrated).

Remember, you can only have one active object when working in the Excel interface and therefore the VBA code is emulating the way users are conditioned to work. Even when a range of cells is selected (Selection object) only on cell is active (the white cell).

```
Sub PrinterName()
      MsgBox "Printer currently set is " & ActivePrinter
End Sub
```

Next Topic: Range & Selection objects

Want to teach yourself Access? Free online guide at About Access Databases

Range & Selection Objects

Range is one of the most widely used objects in Excel VBA, as it allows the manipulation of a row, column, cell or a range of cells in a spreadsheet.

When recording absolute macros, a selection of methods and properties use this object:

```
Range("A1").Select
Range("A1").FormulaR1C1 = 10
```

A generic global object known as **Selection** can be used to determine the current selection of a single or range cells.

When recording relative macros, a selection of methods and properties use this object:

```
Selection.Clear
Selection.Font.Bold = True
```

There are many properties and methods that are shared between **Range** and **Selection** objects and below are some illustrations (my choice of commonly used identifiers):

ADDRESS Property

Returns or sets the reference of a selection.

```
Sub AddressExample()
    MsgBox Selection.Address '$A$1 (default) - absolute
    MsgBox Selection.Address(False, True) '$A1 - column absolute
    MsgBox Selection.Address(True, False) 'A$1 - row absolute
    MsgBox Selection.Address(False, False) 'A1 - relative
End Sub
```

AREAS Property

Use this property to detect how many ranges (*non-adjacent*) are selected.

```
'Selects three non-adjacent ranges
Sub AreaExample()
    Range("A1:B2", E4, G10:J25").Select
    MsgBox Selection.Area.Count 'Number '3' - ranges returned
End Sub
```

The **Count** method returns the number selected as the **Areas** is a property only.

```
'Check for multiple range selection
Sub AreaExample2()
   If Selection.Areas.Count > 1 Then
       MsgBox "Cannot continue, only one range must be selected."
       Exit Sub
    End If
    [Code continues here...]
End Sub
```

Use the **Areas** property to check the state of a spreadsheet. If the system detects multiple ranges, a prompt will appear.

CELLS Property

This property can be used as an alternative to the absolute range property and is generally more flexible to work with, as variables are easier to pass into it.

There are two optional arguments:

```
Cells([row] [,column])
```

Leaving the arguments empty (*no brackets*), it will detect the current selection as the active range.

Adding an argument to either row or column with a number will refer to the co-ordination of the

Tip: You can refer to **Range("A1")** using the convention **[A1]** which may be easier to write.

VBA Keywords: Range, Select, Clear, Font, Bold, Address, Selection, MsgBox, Area, Count, Cells, InputBox, CurrentRegion, OffSet, Resize, Columns, Rows, Column, Row, Dim, Activate, ClearFormats, ClearComments, ClearContents, ClearNotes, ClearOutline, Cut, Copy, PasteSpecial, Insert, Delete, EntireRow, Set, Borders, Interior, Do...Loop, For...Next and If...Then

number passed.

Adding both arguments will explicitly locate the single cell's co-ordinate.

```
'Examples of the Cells property
Sub CellsExample()
    Cells.Clear 'clears active selection
    Cells(1).Value = "This is A1 - row 1"
    Cells(, 1).Value = "This is A1 - col 1"
    Cells(1, 1).Value = "This is A1 - explicit"
    Cells(3, 3).Value = "This is C3"
    Cells(5, 3).Font.Bold = True
End Sub
```

Variables can be passed into the **Cells** property and then nested into the **Range** object as in the following example:

```
'Two InputBoxes for rows and columns
Sub CellsExample2()
    On Error GoTo handler
    Dim intRows As Integer
    Dim intCols As Integer
    intRows = CInt(InputBox("How many rows to populate?"))
    intCols = CInt(InputBox("How many columns to populate?"))
    'starts at cell A1 to the number of rows and columns passed
    Range(Cells(1, 1), Cells(intRows, intCols)).Value = "X"
    Exit Sub
handler:
    'Error code is handled here...
End Sub
```

By wrapping a range property around two cell properties, the flexibility of passing variables becomes apparent.

```
Range(Cells(1, 1), Cells(intRows, intCols))
```

> Error handlers and InputBox functions are covered later in this guide.

Column(s) and Row(s) Properties

Four properties that return the column or row number for the focused range.

The singular (***Column*** *or* ***Row***) returns the active cell's co-ordinate and the plural (***Columns*** *or* ***Rows***) can be used to count the current selections configuration.

```
Sub ColRowExample()
    MsgBox "Row " & ActiveCell.Row & _
        " : Column " & ActiveCell.Column
End Sub
```

```
Sub ColsRowsCountExample()
    MsgBox Selection.Rows.Count & " rows by " _
        & Selection.Columns.Count & " columns selected"
End Sub
```

CURRENTREGION Property

Selects from the active cell's position all cells that are adjacent (*known as a region*) until a blank row and blank column breaks the region.

Use this statement to select a region.

```
Selection.CurrentRegion.Select
```

> Make sure you have some data to work with.

To select a region of data and exclude the top row for a data list:

	A	B	C
1	Customer ID	Company Name	Contact Name
2	ALWAO	Always Open Quick Mart	Melissa Adams
3	ANDRC	Andre's Continental Food Market	Heeneth Ghandi
4	ANTHB	Anthony's Beer and Ale	Mary Throneberry
5	AROUT	Around the Horn	Thomas Hardy
6	BABUJ	Babu Ji's Exports	G.K.Chattergee
7	BERGS	Bergstad's Scandinavian Grocery	Tammy Wong

Run this piece of code:

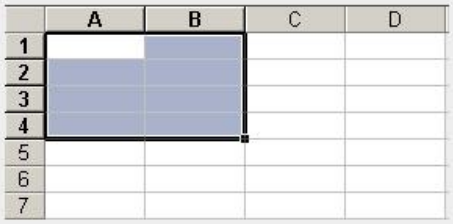
```
Sub RegionSelection()
    ActiveCell.CurrentRegion.Offset(1, 0).Resize( _
        ActiveCell.CurrentRegion.Rows.Count - 1, _
            ActiveCell.CurrentRegion.Columns.Count).Select
End Sub
```

Make sure the active cell is in the region of data you wish to capture before running the above procedure.

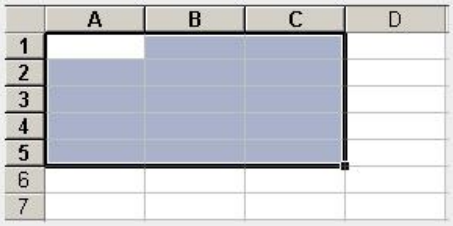
RESIZE Property

This property is useful for extending or re-defining a new size range.

To extend this range



by one row and one column to



use the code snippet below:

```
Sub ResizeRange()
   Dim rows As Integer
    Dim cols As Integer
    cols = Selection.Columns.Count
    rows = Selection.rows.Count
    Selection.Resize(rows + 1, cols + 1).Select
End Sub
```

Resizing a range can be increased, decreased or change the configuration (*shape*) by combining positive and negative values inside the **Resize** property's arguments.

OFFSET Property

This property is used in many procedures as it controls references to other cells and navigation.

Two arguments are passed into this property that is then compounded with either another property or a method.

```
Selection.OffSet(1, 2).Select
ActiveCell.OffSet(0, -1).Value = "X"
```

> Consider referring to an offset position rather than physically navigating to it – this will speed up the execution of code particularly while iterating.

For example:

```
Sub OffSetExample1()
    Dim intCount As Integer
    Do Until intCount = 10
        ActiveCell.Offset(intCount, 0).Value = "X"
        intCount = intCount + 1
    Loop
End Sub
```

is quicker to execute than:

```
Sub OffSetExample2()
    Dim intCount As Integer
    Do Until intCount = 10
        ActiveCell.Value = "X"
        ActiveCell.Offset(1, 0).Select
        intCount = intCount + 1
    Loop
End Sub
```

Do...Loops (iterations) are covered later in this guide

The above two examples produce the same result but instead of telling Excel to move to the active cell and then enter a value, it is more efficient to refer (*or point*) to the resulting cell and remain in the same position.

- A positive value for the row argument refers to a row downwards.
- A positive value for the column argument refers to a column to its right.
- A negative value for the row argument refers to a row upwards.
- A negative value for the column argument refers to a column to its left.

Be careful to introduce error-handling procedures when working with the '*Offset*' property as if you navigate or refer to a position outside the scope of the fixed dimensions of a worksheet, this will certainly cause a run time error (See *Error Handling & Debugging*).

ACTIVATE Method

This method should not be confused with the **Select** method as commonly used in VBA.

The **Select** method means go and navigate to it.

```
Range("A1").Select.
Range("A1:C10").Select
```

The **Activate** method selects a cell within a selection.

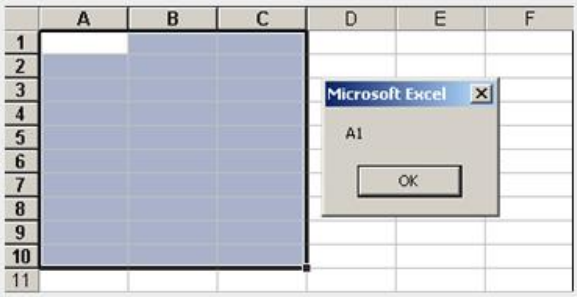
By default, in a selection of cells, the first (*top left position*) is the active cell *(white cell in a block*).

Example:

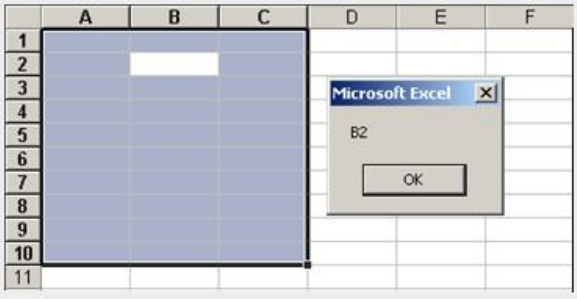
```
Sub ActivateMethodExample()
    'select a fixed range
    Range("A1:C10").Select
    MsgBox ActiveCell.Address(False, False)
    Range("B2").Activate
    MsgBox ActiveCell.Address(False, False)
End Sub
```

The above procedure selects a fixed range of cells with a message box confirming the address of the active cell. Then, using the **Activate** method, move the active cell to address B2.

From



to



CLEAR Methods

There are six variations of this method:

1. **Clear** – all attributes are cleared and reset to default
2. **ClearComments** – clear comments only
3. **ClearContents** – clear contents only (delete key command)
4. **ClearFormats** – clear formats only (revert to general format)
5. **ClearNotes** – clear comments and sound notes only
6. **ClearOutline** – clear on outlines implemented

Simply locate the object and use of the above methods:

```
'Different ways to refer to a selection
```

```
Sub ClearMethodsExamples()
    Range("A1:C10").Clear
    Selection.ClearComments
    Selection.CurrentRegion.ClearContents
    ActiveCell.ClearFormats
    Range(Cells(1, 1), Cells(5, 3)).ClearNotes
    Columns("A:E").ClearOutline
End Sub
```

CUT, COPY and PASTESPECIAL Methods

These methods simulate the windows clipboard cut, copy and paste commands.

There are a few different types of these methods where most arguments are optional and by changing the argument settings, will change the behaviour of the method.

Some examples:

```
'Simple Copy and Paste
Sub CopyPasteData1()
    Range("A1").Copy
    Range("B1").PasteSpecial xlPasteAll
End Sub
```

```
'Copy and Paste Values only (no format)
Sub CopyPasteData2()
    Range("A1").Copy
    Range("B1").PasteSpecial xlPasteValues
End Sub
```

```
'Simple Cut and Paste
Sub CutPasteData()
    Range("A1").Cut Range("B1")
End Sub
```

If the copy and cut methods omit the argument **Destination**, the item is copied to the windows clipboard.

INSERT and DELETE Methods

These methods can add or remove cells, rows or columns and is best used with other properties to help establish which action to execute.

Some examples:

```
'Inserts an entire row at the active cell
Sub InsertRow()
     ActiveCell.EntireRow.Insert 'or EntireColumn
End Sub
```

```
'Deletes an entire row at the active cell
Sub DeleteRow()
     ActiveCell.EntireRow.Delete 'or EntireColumn
End Sub
```

```
'Inserts an entire row at row 4
Sub InsertAtRow4()
     ActiveSheet.rows(4).Insert
End Sub
```

```
'Insert columns and move data to the right
Sub InsertColumns()
     Range("A1:C5").Insert Shift:=xlShiftToRight
End Sub
```

Using the SET Keyword Command

Users can create and set a range object instead and like all other object declarations, use the **Set** command (which is used for object variable declarations).

```
'Alternative way of referring to a range
Sub RangeObject()
    Dim rng As Range
    Set rng = Range("A1:B2")
    With rng
        .Value = "X"
        .Font.Bold = True
        .Borders.LineStyle = xlDouble
        'any other properties.........
    End With
    Set rng = Nothing
End Sub
```

This is an alternative way of working with the range object and is sometimes preferred as it exposes more members (properties and methods).

For example, using a **For...Loop** (*see* For...Loop *section about this control flow*), iterating in a collection is carried out by declaring and setting a variable as a **Range** object:

```
'Loops through the elements of the Range object
Sub IterateRangeObject()
    Dim r1 As Range
    Dim c As Object
    Set r1 = Range("A1:C10")
    For Each c In r1
        If c.Value = Empty Then
            c.Value = "0"
        End If
    Next c
End Sub
```

The above procedure checks each cell in a fixed range (*A1 to C10*) and determines its value, placing a 0 (*zero*) if the cell is empty.

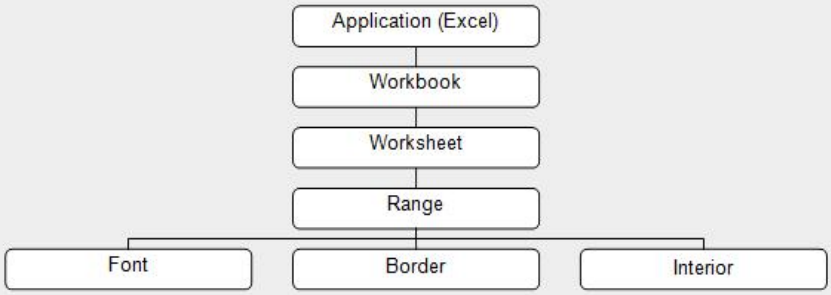
Next Topic: Object hierarchy

Want to teach yourself Access? Free online guide at About Access Databases

Object Hierarchy

VBA Keywords: Application, WorkBook(s), WorkSheet(s), Range, Font, Border, Interior, Select, ActiveCell, Add.

Most applications consist of many objects arranged in a hierarchy.



Objects, Methods, Properties and Variables

Each line of code generally has the same structure (which is also known as **Syntax**). VBA is loosely based around the concept of **Object Orientated Programming** (OOP) and the following syntax is used to define how you write code using any of the libraries that are loaded.

OBJECT.*Identifier*[.*sub_Identifier*]

The square brackets wrapped around the *sub_Identifier* is the convention meaning it is optional and therefore not always required.

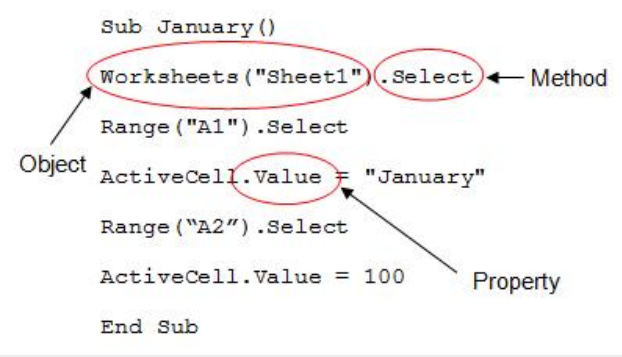
An *Identifier* and *sub_Identifier* can be one of three types:

1. Property
2. Method
3. Event

Similar to physical objects such as a car or a chair, the application objects, as listed above, have **Properties** and **Methods** (*as well as **Events***)

Object	Property	Method
`Car`	`Colour`	`Accelerate`
`ActiveCell`	`Value`	
`Worksheets("Sheet1")`		`Select`

Identifying the Objects, Methods and Properties from the previous example.



Examples

1. Create a new blank workbook.

2. Click on the **Tools** menu, select **Macro** and choose Visual Basic Editor.
3. Click on the **Insert** menu and select **Module**.

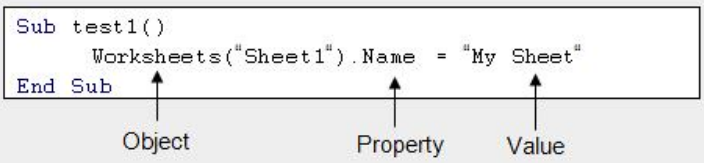
Properties

A **Property** is an attribute of an object, e.g. the colour of a car, the name of a worksheet.

Object.Property = Value

```
Car.Colour = Red
Worksheets("Sheet1").Name = "My Sheet"
```

The following example will change the name of "Sheet1" to "My Sheet".



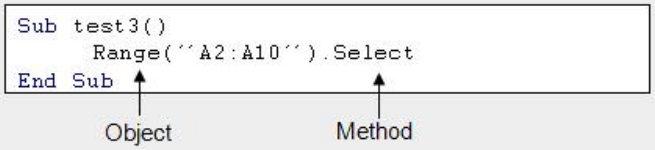
Methods

A Method is an activity that an object can be told to do, e.g. accelerate the car, select a cell, insert a worksheet, delete a worksheet.

Object.Method

```
Car.Accelerate
Range("A2:A10").Select
```

The following example will select a range of cells (A2 to A10) in the current worksheet.



Methods that contain arguments

There are methods that contain many arguments, for example inserting a worksheet(s). The numerous arguments contain information about how many worksheets you would like to insert, the position of the worksheet(s) and the type of the worksheet(s).

Object.Method Argument1,Argument2,...

Example:

```
Worksheets.Add Before, After, Count, Type
```

Add	Add a new worksheet.
Before/After	Before which worksheet? After which worksheet?
Count	How many worksheets
Type	What type of worksheet ie worksheet, chart sheet etc

The following example will place 2 new sheets after Sheet 2.

```
Worksheets.Add, Sheets("Sheet2"), 2
```

§ The comma after **Add** represents the "**Before**" argument.

§ If "Type" is omitted, then it will assume the Default Type. The Default Type is **xlworksheet**.

```
Sub Insert2Sheets()
    Worksheets.Add Sheets("Sheet2"), 2
End Sub
```

Events

Events are like Methods the only difference being when and who/what calls this action. It is an action like a method but the system will trigger it for you instead of the user invoking the action.

This is very useful when a system changes state and needs to automate a procedure on behalf of the user.

In fact, there are many events being triggered all the time; users are simply not aware of this unless there is a procedure to trigger it. The system constantly listens for the event to take place.

When you use standard Excel features like **Data Validation** or **Conditional Formatting**, the system automatically creates code for an event so when users enter a value in a cell it will automatically trigger the feature and validate and format the active cell without a user calling the a macro manually. This type of event is normally known as '**Enter**' for a Worksheet.

There are many predefines events which have no code just a starting and ending signature and users need to add code in between these signatures. Take a look at Event Handling for more information.

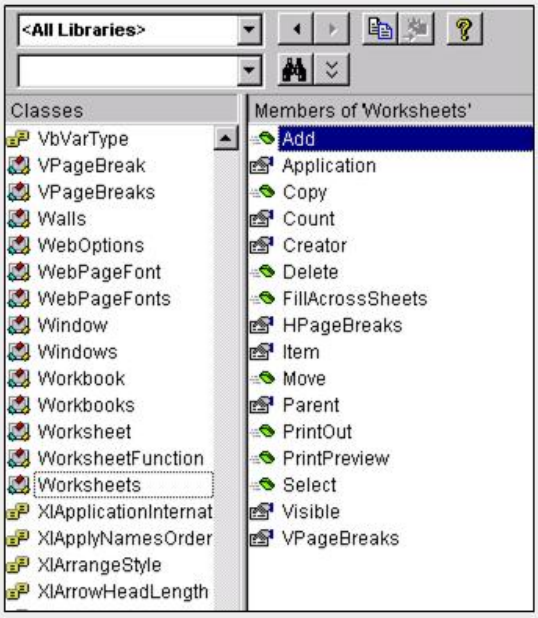
Next Topic: Object Browser

Object Browser

The Object Browser enables you to see a list of all the different objects with their methods, properties, events and constants.

In the VBE editor:

1. Insert menu and select a Module.
2. Click on the **View** menu and select **Object Browser** (shortcut key: **F2**).
3. Make sure it's set to **'<All Libraries>'**.



Notice **Add**([*Before*], [*After*], [*Count*], [*Type*]) is one of the examples previously seen.

The main two panes contain on the left **Classes** (also known as Objects) and on the right **Members** (also known as Identifiers).

By selecting a class, you display its members which are of three key types; **Properties**, **Methods** and **Events**.

Libraries

Libraries are the application divisions of a collection of classes (objects). Therefore, you will have a class for **Excel**, **VBA** and other applications you wish to have a reference to. The **Excel** (the host), **VBA** and **VBAProject** are mandatory and can not disabled. All other library files can be switched on or off as required.

In order to code to another applications (for example, Microsoft Word) you will need to load its library first.

To switch between libraries or show all libraries, choose the '*Project/Library*' drop down box:

The default libraries available:

1. **Excel** – A collection of classes available in Excel i.e. workbook, worksheet, range, chart, etc…

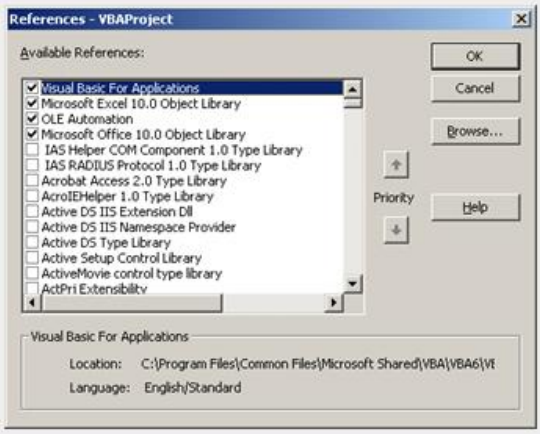
Tip: You can press **F2** function key to load the Object Browser.

2. **Office** – A collection of classes generic to all office applications i.e. command bar, command icon, help assistance, etc…
3. **stdole** – A collection of standard OLE classes which allow other OLE applications to share information (Not covered in this manual).
4. **VBA** – A collection of classes which allow generic functions to be used i.e. MsgBox, InputBox, conversion functions, string functions, etc…
5. **VBAProject** – A collection of classes local to the active workbook project, which includes sheets, workbook and any user, defined classes.

Other libraries are also available but require to be enabled before they can be used which include **Word**, **Outlook**, **DAO, ADODB** and many others.

By enabling the additional libraries, developers can start to code and communicate with other applications and processes, which start to reveal the potential power of Visual Basic (VBA).

To enable a library, from the Visual Basic Editor, choose **Tools** menu and select **References…**



Scroll down to find the required library or choose the **Browse…** button to locate the required library.

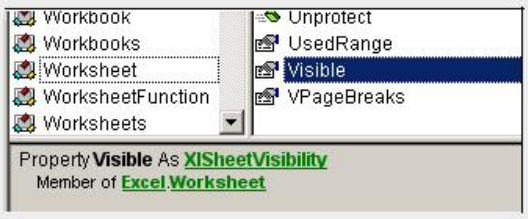
Excluding the top two libraries, a library priority order matters that is why users can re-arrange the order using the **Priority** buttons. The way the system works is when a procedure is executed, it checks to see which library is required in order to execute the line-by-line code. In some cases, a method or class can be duplicated between libraries and it is therefore important to be able to call the correct method or class first superseding the lower level references.

Structure of a Library

Each **Library** will typically have a collection of **classes**. A **class** or **object class** is in essence the object i.e. Worksheet.

Each object class will have a collection of **members**, which could be a collection of properties, methods and events.

When looking at the Object Browser, users will see on the left hand side many classes to the active library. To the right hand pane, all members of the selected class will reveal properties, methods and events.



The above illustration shows the **Excel** library, Worksheet class and the **Visible** property highlighted (**Excel.Worksheet.Visible**).

Right mouse click the selected item to choose the **Help** command and go straight to the offline help documentation.

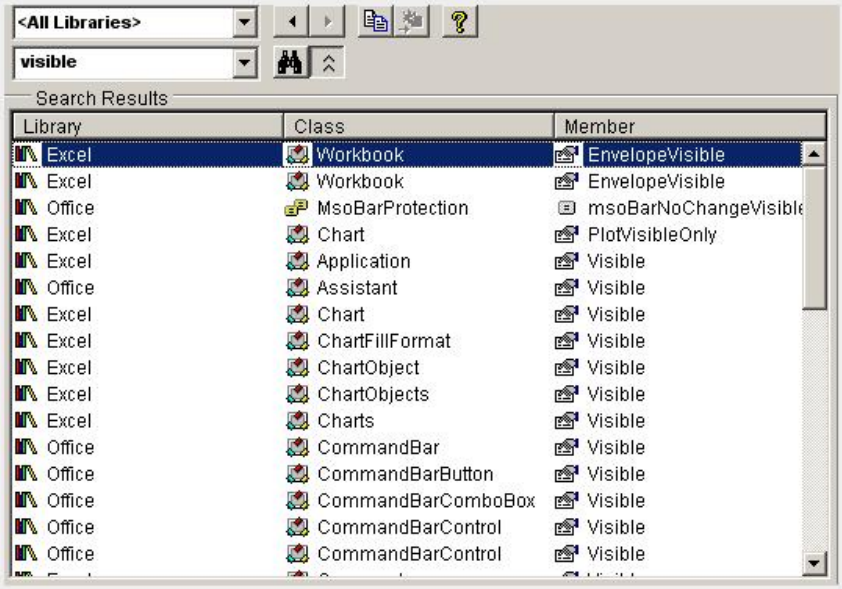
Browsing the right hand and pane of the Object Browser, users will see three different icons to help identify the member type:

Member Icon	Member Type
	Property
	Method
	Event

Browser Searching

The search tool allows users to locate if a keyword firstly exists and secondly where it could possibly reside

At the top half of the browser window, type the desired keyword and then click the search button:



The above example looked for the keyword **visible** across all libraries.

After locating the correct item (*selecting the item*), users can use the copy button function and then paste into a code window.

Next Topic: Chart objects

Charts Objects

VBA Keyword: ActiveSheet, AddChart, Select, SetSourceData, With...End With, ChartType, ChartObjects, SeriesCollections, NewSeries & Set.

When you add a chart when recording a macro, the code generated follows the menu and command that users manually call when adding a chart which means there is a discipline to run the macro in exactly the same way or face the potential of landing up with different results or even errors.

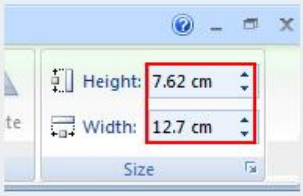
Typical code for a recorded macro:

```
Sub RecordedMacroChart()
'
' RecordedMacroChart Macro
'
    ActiveSheet.Shapes.AddChart.Select
    ActiveChart.SetSourceData Source:=Range("'Sheet2'!$A$2:$D$12")
    ActiveChart.ChartType = xlLineMarkers
End Sub
```

The other issue with the above example code is typically the reference to the source data (which is an absolute string reference **`Sheet2'!$A$2:$D$12`**). The user may want more flexibility in controlling where this reference is by using and passing variables.

Note: Previous versions of Excel records macros using the object and method **`Charts.Add`** but it still gives the inflexibility in terms of control.

The above code generates a standard size chart within a worksheet and there is no room for setting properties until users edit the properties of an existing chart which just adds more code and becomes inefficient.



Standard dimensions for chart object

Create Chart Objects

By creating your own written procedure and introduce **`ChartObjects`** keyword with supporting methods and properties, you have more control and can be flexible in passing arguments thus reducing extra lines of code.

An example:

```
Sub WrittenMacroChart()
'
' WrittenMacroChart Macro
'
    With ActiveSheet.ChartObjects.Add _
            (Left:=100, Width:=400, Top:=100, Height:=250)
        .Chart.SetSourceData Source:=Sheets("Sheet2").Range("A2:D12")
        .Chart.ChartType = xlXYScatterLines
    End With
End Sub
```

The above example allows the chart object to be positioned and sized (measured in pixels) accordingly using the **`Add`** method and its arguments.

Defining object variables for longer based procedures makes the code more clinical and efficient to write even though we must first declare a new object (as **`ChartObject`**).

```
Sub WrittenMacroChartObject()
    Dim ChrtObj As ChartObject
    Set ChrtObj = ActiveSheet.ChartObjects.Add _
            (Left:=100, Width:=400, Top:=100, Height:=250)
    ChrtObj.Chart.SetSourceData Source:=Sheets("Sheet2").Range("A2:D12")
    ChrtObj.Chart.ChartType = xlXYScatterLines
End Sub
```

The above example is the same as the previous code snippet but using the object variable **`ChrtObj`**

as declared and set.

The other useful method is **SetSourceData** as you can add as many series as required (one at a time) enabling what ranges you want to set and not let Excel make as assumption.

Adding Series

When recording a macro adding a series each line of code is created for a name, y-axis values and x-axis values if required using **SeriesCollection** and **NewSeries** keywords.

An example of recorded macro which adds three series (names and values) and an y-axis to an existing empty chart on a worksheet:

```
Sub AddingSeries()
     ActiveChart.SeriesCollection.NewSeries
     ActiveChart.SeriesCollection(1).Name = "='Sheet2'!$B$2"
     ActiveChart.SeriesCollection(1).Values = "='Sheet2'!$B$3:$B$12"
     ActiveChart.SeriesCollection.NewSeries
     ActiveChart.SeriesCollection(2).Name = "='Sheet2'!$C$2"
     ActiveChart.SeriesCollection(2).Values = "='Sheet2'!$C$3:$C$12"
     ActiveChart.SeriesCollection.NewSeries
     ActiveChart.SeriesCollection(3).Name = "='Sheet2'!$D$2"
     ActiveChart.SeriesCollection(3).Values = "='Sheet2'!$D$3:$D$12"
     ActiveChart.SeriesCollection(3).XValues = "='Sheet2'!$A$3:$A$12"
End Sub
```

Using the With...End With statement will refine the code and make it easier to understand. Also, introducing your own objects for a series just gives you better control should you wish to assign variables and arrays to it.

The same as the above example but a better practice:

```
Sub AddingSeriesObjects()
    Dim ChrtSrs1 As Series, ChrtSrs2 As Series, ChrtSrs3 As Series
    Set ChrtSrs1 = ActiveChart.SeriesCollection.NewSeries
    With ChrtSrs1
        .Name = "='Sheet2'!$B$2"
        .Values = "='Sheet2'!$B$3:$B$12"
        .XValues = "='Sheet2'!$A$3:$A$12"
    End With
    Set ChrtSrs2 = ActiveChart.SeriesCollection.NewSeries
    With ChrtSrs2
        .Name = "='Sheet2'!$C$2"
        .Values = "='Sheet2'!$C$3:$C$12"
        .XValues = "='Sheet2'!$A$3:$A$12"
    End With
    Set ChrtSrs3 = ActiveChart.SeriesCollection.NewSeries
    With ChrtSrs3
        .Name = "='Sheet2'!$D$2"
        .Values = "='Sheet2'!$D$3:$D$12"
        .XValues = "='Sheet2'!$A$3:$A$12"
    End With
End Sub
```

Remember, you can pass variables better into the above example code (not illustrated).

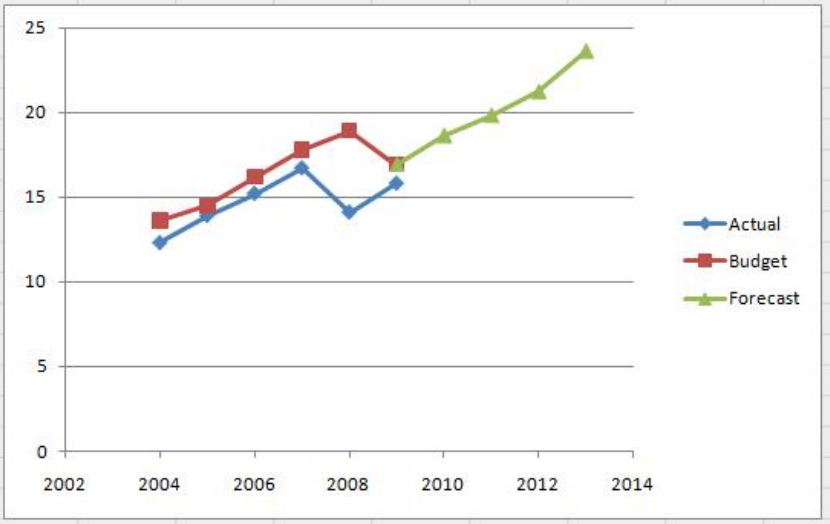
To delete a series use **ActiveChart.SeriesCollection(1).Delete** where the index (brackets with a 1) represents the first series for the active chart.

Using arrays and collections, you have better control especially when wanting to handle multiple charts in one go.

Data used for the above chart:

	A	B	C	D
1	Sales Forecast			
2		Actual	Budget	Forecast
3	2004	12.3	13.6	
4	2005	13.9	14.5	
5	2006	15.2	16.2	
6	2007	16.7	17.8	
7	2008	14.1	18.9	
8	2009	15.8	16.9	16.9
9	2010			18.6
10	2011			19.8
11	2012			21.2
12	2013			23.6

Chart object looks something like:



Next Topic: Pivot Table objects

Want to teach yourself Access? Free online guide at About Access Databases

Pivot Table Objects

VBA Keywords: PivotCaches, CreatePivotTable, Sheets, Cells, Select, ActiveSheet, Dim, Set & CurrentRegion.

Pivot Tables are a very popular and powerful Excel feature and most users generate this type of object using the standard wizard (pre 2007) or Insert action (2007 or later) command.

Once again, recording a macro is a good starting point but the code, efficiency and interpretation is sometimes difficult to manage and can cause errors when running a recorded macro. Instead, users can always call and Excel **Pivot Tables** *object* which is a member of the **Pivot Tables** *Collection*.

Here is an example recorded macro based on some data located in a worksheet called '*Sales List*' which has a range *A1:M306*:

```
Sub SummaryPivotReport()

    Sheets.Add
    ActiveWorkbook.PivotCaches.Create(SourceType:=xlDatabase, _
        SourceData:="Sales List!R1C1:R306C13", _
        Version:=xlPivotTableVersion10).CreatePivotTable _
        TableDestination:="Sheet1!R3C1", _
        TableName:="PivotTable1", DefaultVersion:=xlPivotTableVersion10
    Sheets("Sheet1").Select
    Cells(3, 1).Select
    With ActiveSheet.PivotTables("PivotTable1").PivotFields("Product")
        .Orientation = xlRowField
        .Position = 1
    End With
    With ActiveSheet.PivotTables("PivotTable1").PivotFields("Assistant")
        .Orientation = xlColumnField
        .Position = 1
    End With
    ActiveSheet.PivotTables("PivotTable1").AddDataField _
        ActiveSheet.PivotTables("PivotTable1").PivotFields("TOTAL"), _
        "Sum of TOTAL", xlSum
    With ActiveSheet.PivotTables("PivotTable1").PivotFields("Method")
        .Orientation = xlPageField
        .Position = 1
    End With

End Sub
```

Try running it and you will discover one of several errors.

The errors generated is not down to the **Pivot Table** object, **PivotCaches** or **PivotTables** collection failing but the absolute references to either a worksheet or ranges being called.

Even if you are prepared to keep the recorded macro as above and just simply change the references, then you have made a start and a reason for editing this macro.

For example, changing the range reference (which is absolute) and handling the absolute worksheet name to be more dynamic and relative:

When the system adds a new worksheet `Sheets.Add` it generates a unique name each time (which is absolute). Later in the procedure it refers to the name of new added worksheet (which is why it fails when running the macro).

Instead of referring to `TableDestination:="Sheet1!R3C1"` in the Pivot objects **TableDestination** argument, consider using this code `TableDestination:=ActiveSheet.Cells(1, 1)` instead which picks the current worksheets cell A1.

Remove the line **`Sheets("Sheet1").Select`** and the rest of the code should now work as recorded.

Now lets take a look at the Pivot Table object itself and build the knowledge so that you understand the elements and arguments correctly.

Use the **Create** and **CreatePivotTable** methods of the **PivotCaches** object:

```
ActiveWorkbook.PivotCaches.Create(SourceType:=xlDatabase, _
        SourceData:="Sales List!R1C1:R306C13", _
        Version:=xlPivotTableVersion10).CreatePivotTable _
        TableDestination:=ActiveSheet.Cells(1, 1), _
        TableName:="PivotTable1", DefaultVersion:=xlPivotTableVersion10
```

Note: In Excel 2003 (or earlier) users tend to use the **Add** method instead of the **Create** method which has its own set of arguments.

The **SourceData** argument is a range reference to the data list source. Also note this reference in the above example is an absolute reference too and should be careful should the data source change and grow dynamically. Consider using a variable or object variable to hold the current address of a region of data to pass into the **Create** method.

*For example (before adding the new worksheet - **Sheets.Add**) include the following:*

```
Selection.CurrentRegion.Select
MyDataRef = Selection.Address
```

Or, consider an object reference instead:

```
Set rngSource As ActiveSheet.Range("A1").CurrentRegion
```

The **TableDestination** argument pinpoints where the starting cell in a worksheet (normally a new worksheet) is and should really be dynamic and relative (as previously mentioned).

The remaining arguments are optional but as the macro records distinct settings they have been included. Refer to the Excel VBA help files for further information.

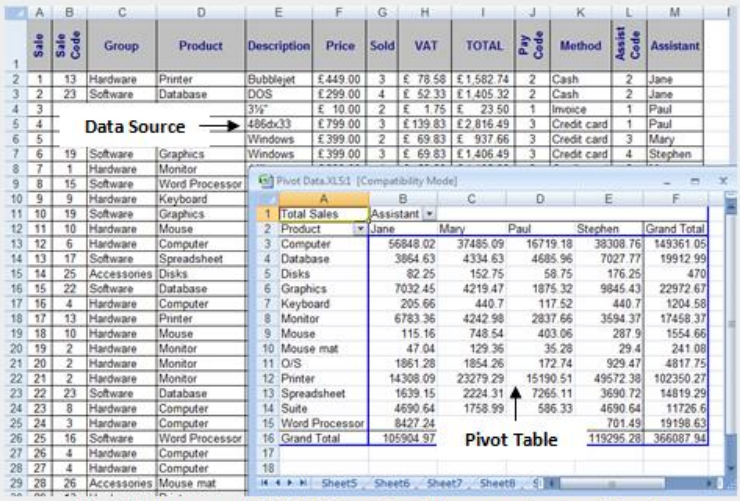
Pivot Tables are objects and can given a unique name as in the above example shown called "PivotTable1".

Using the named object, you can then refer to elements in a Pivot Table which include **PivotFields** *collections* and their properties; **Orientation**, **Position** and calculated functions.

Here is a revised piece of code for the same above procedure:

```
Sub SummaryPivotReport()

    Dim rngSource As Range
    Dim wksTrans As Worksheet
    Dim ptTrans As PivotTable

    'Create and set objects
    Set rngSource = ActiveSheet.Range("A1").CurrentRegion
    Set wksTrans = Worksheets.Add(after:=ActiveSheet)

    'Generate a new Pivot Table report using the above objects
    ActiveWorkbook.PivotCaches.Create(SourceType:=xlDatabase, _
        SourceData:=rngSource, _
        Version:=xlPivotTableVersion10).CreatePivotTable _
        TableDestination:=wksTrans.Range("A1"), TableName:="Trans", _
        DefaultVersion:=xlPivotTableVersion10

    'set an instance for the new pivit table
    Set ptTrans = wksTrans.PivotTables("Trans")

    'set and add fields to the new pivot table (Trans)
    ptTrans.AddFields RowFields:="Product", ColumnFields:="Assistant"

    'Adding the sum function to the main data section
    ptTrans.PivotFields("TOTAL").Orientation = xlDataField
    ptTrans.DataFields(1).Function = xlSum
    ptTrans.DataFields(1).Name = "Total Sales"

End Sub
```

The above example will give you better control and more flexibility in defining a Pivot Table without have to ensure your have added a new worksheet, placed the cursor in the right position and any duplicate name references (worksheets).



Pivot Table generated via VBA from the data source (new worksheet)

Next Topic: Formulas

Want to teach yourself Access? Free online guide at About Access Databases

Formulas

Note: Your knowledge of Excel formulae and functions will help derive VBA calculations.

Expressions

An Expression is a value or group of values that expresses a single item (*variable*, *object*), which evaluates to a value or result.

For example, an expression of **2+2** will result **4**.

Expressions are made up of either of the following:

- Constants
- Variables
- Operators
- Arrays
- Functions

Operators

Operators are used to combine expressions that will manipulate the expression and will return a value (*answer*). Typical operators:

**= + - / * > < , . <> <= => ^ & **

True, False, Or, Not, Null, Empty, Is, Like, Mod,

There are more - refer to on-line Help or Users Guide.

Operators have precedence, which will affect the value (*result*) if not carefully used. For example, the following two expressions will result to a different value

1 + 2 * 3 / 2 - 6 = -2

((1 + 2) * 3) / 2 - 6 = -1.5

A bracket changes the order in which the expression is calculated.

The following table will give an indication of how expressions are calculated:

Operator	Priority Level (in highest order)
( ) Brackets	**1**
^ Power sign	**2**
-ve Negative values	**3**
*, / Multiply and Divide	**4**
+, - Addition/Subtraction	**5**
<, <=, >, >=	**6**
Like	**7**

There are more operators that fit in between the above table. Refer to on-line Help or Users Guide.

Concatenate (&)

Concatenate is the term used to join two or strings (text values) together. You use the **&** (ampersand) as the operator (*the glue*) to connect one string to another.

This is commonly used in VBA to build string messages and narratives which will part static and dynamic (variables) enhancing the procedure to a more user friendlier environment.

For example, to join my first name with my surname with a space between the two using two variables and a static string (space) between:

```
Sub ConcatenateExample()
    Dim FName As String, SName As String
```

```
    FName = "Ben"
    SName  = "Beitler"
    MsgBox "Welcome " & FName & " " & SName
End Sub
```

Constants

A Constant is a value in a macro that does not change. By using constants, is similar to variables where a value is assigned to the variable.

For example:

Variable known as ***TitleHeading*** can have a value assigned to it.

TitleHeading = “SALES CASH FLOW for JANUARY”

Every time the value is used, by using the variable it is easier to handle, update changes and therefore more efficient.

To assign a named constant, proceed with the command **Const** before the variable.

For example

Variable known as ***pi*** can have a value of **3.1415926** assigned to it. Since this is a fixed value for ***pi***, assign the **Const** command to it.

```
Const PI = 3.1415926
```

Functions

Functions, like Excel functions are used to return a value.

For example:

```
MySName = "Anderson"
Surname = UCase(MySName)
```

Where **Surname** variable uses an UCase function to convert the variable, **MySName** data to upper case.

Example:

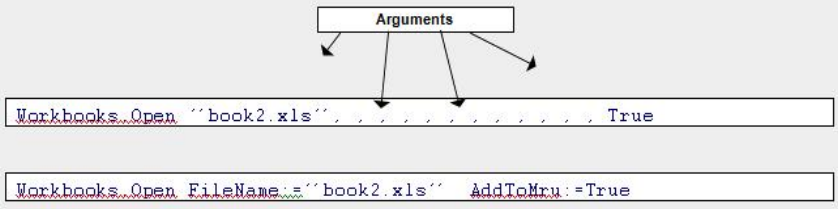
The following example will put a date and time formula in to the active cell and then resize the column.

```
Sub formula1()
    ActiveCell.Formula = "=Now()"
    ActiveCell.EntireColumn.AutoFit
End Sub
```

Arguments

Some functions and statements will contain required and optional arguments. An argument is an element of that function or statement which is required to apply that command. Arguments either use a comma separator or can use its syntax name.

For example:



Absolute Formula

Similar to the Absolute Macros we looked at earlier, an Absolute Formula will refer to specific cell references.

```
Sub formula()
    ActiveCell.Formula = "=Sum(A1:A4)"
End Sub
```

Relative Formula

Similar to the Relative Macros we looked at earlier, a Relative Formula will refer to the active cell and the other cells around it.

```
Sub formula()
    ActiveCell.FormulaR1C1 = "=Sum(r[-4]c:r[-1]c)"
End Sub
```

Instead of using cell references in the formula, i.e. **`A1:A4`**, the relative formula uses row and column references, i.e. **`r[-4]c:r[-1]c`**.

In the above example, **`r[-4]c`** refers to the cell 4 rows above the active cell in the same column and **`r[-1]c`** refers to the cell 1 row above the active cell in the same column.

Next Topic: Visual Basic Functions

Visual Basic Functions

Using VBA is a combination of utilising libraries available, mainly Excel and VBA.

The following list of functions is a selection of commonly used functions, which are classes of the **VBA** library.

CONVERSION Class

This class contains various functions to help cast and convert values from one data type to another.

Some functions in this class:

CBool(*expression*) – convert to a Boolean (true/false) value
CByte(*expression*) – convert to a Byte value
CCur(*expression*) – convert to a Currency value
CDate(*expression*) – convert to Date value
CDbl(*expression*) – convert to a Double value
CDec(*expression*) – convert to a Decimal value
CInt(*expression*) – convert to an Integer value
CLng(*expression*) – convert to a Long value
CSng(*expression*) – convert to a Single value
CStr(*expression*) – convert to a String value
CVar(*expression*) – convert to Variant value

The expression can be either a string or numeric value, which is converted to one of the above data types.

```
Sub ConvertValue()
    Dim strInput As String
    Dim intNumber As Integer
    strInput = InputBox("Enter a Number:")
    intNumber = CInt(strInput)
    ......................
End Sub
```

The above example takes a string variable, which the InputBox function returns as a string and converts it to an integer value and stores to the integer, variable.

VBA is intelligent enough to convert values without the need to apply conversion functions or explicitly declare variables. However, there are occasions when this rule doesn't work and to handle unforeseen errors, users need to handle data conversion (*as above*).

In future releases of VBA, data type declarations will become more stringent in how users can work with variables and will therefore need to use such functions (*as above*) to handle cast and conversion issues correctly.

DATETIME Class

This class is a collection of date/time conversions and interrogations.

Some functions in this class:

Date – return/sets the system's date

Now – returns/sets the system's date and time

Day(*Date*) – returns the day element of the date

Month(*Date*) – returns the month element of the date

Year(*Date*) – returns the year element of the date

DateDiff(*interval, date1, date2* [,*firstdayofweek*] [,*firstweekofyear*])
– returns the difference between two dates driven by the interval

DateSerial(*Year, Month, Day*) – returns a valid date from 3 separate values

DateValue(*Date*) – converts a string date into a date data type date

Weekday(*Date*, [*firstdayofweek*]) – Returns a string day of the week

```
'Converts a string date to date date type date
Sub DateExample1()
    Dim strDate As String
    strDate = "10 May 2010"
    MsgBox DateValue(strDate)
End Sub
```

Note: The VBA library is the top most reference followed by the Excel library and both can not be moved or disabled. Therefore, calling a function which exists in both libraries will always use the VBA reference (as implicit).

VBA Keywords: MsgBox, InpuBox, Application, WorksheetFunction and VBA Class (all functions).

```
'Works out the difference between two dates
'returns the number of months (interval)
Sub DateExample2()
    Dim dtmStartDate As Date
    dtmStartDate = #5/2/2010#
    MsgBox DateDiff("m", dtmStartDate, Date) & " Months"
End Sub
```

INFORMATION Class

This class is a collection of status functions to help evaluate conditions of variables and objects alike.

Some functions in the class:

IsArray(*variant***)** – returns true or false

IsDate(*expression***)** – returns true or false

IsError(*expression***)** – returns true or false

IsEmpty(*expression***)** – returns true or false

IsMissing(*variant***)** – returns true or false

IsNull(*expression***)** – returns true or false

IsNumeric(*expression***)** – returns true or false

IsObject(*expression***)** – returns true or false

An expression or variant is the variable being tested to see if it is **`True`** or **`False`**.

MATH Class

This class is a collection of mathematical functions that can be used to change variables with ease and without having to create your own functions.

Some functions in the class:

Abs(*Number***)** – returns the absolute number (always a positive value)

Rnd([*Number***])** – returns a random value

Round(*Number, [NumDigitsAfterDecimal]***)** – returns a rounded value

Sqr(*Number***)** – returns a square value (x^2)

```
'Generates a random value between 1 and 100.
Sub RandomNumber()
    Dim intNumber As Integer
    intNumber = Int((100 * Rnd) + 1)
    MsgBox intNumber
End Sub
```

STRING Class

This class is a collection of text (*string*) based functions that include conversion, extractions and concatenation.

Some functions in the class:

Asc(*String***)** – returns the numeric ASCII value of the character string

Chr(*CharCode***)** – returns the character string from the code supplied

Format(*Expression*, [*Format*], [*FirstDayOfWeek*], [*FirstWeekOfYear*]**)**

- returns the format presentation of the expression

InStr([*Start***], [***String1***], [***String2***], [***Compare***])** – returns the numeric position of the first character found from left to right

InStrRev(*StringCheck*, *StringMatch*, [*Start*], [*Compare*]**)** – returns the numeric position of the first character found from right to left

LCase(*String***)** – returns the string in lowercase

UCase(*String***)** – returns the string in uppercase

Left(*String*, *Length***)** – returns the remaining characters from the left of the length supplied

Right(*String*, *Length***)** – returns the remaining characters from the right of the length supplied

Len(*Expression***)** – returns a value of the length of characters supplied

Mid(*String*, *Start*, [*Length*]**)** – returns the portion of characters as determined by the start and end parameters supplied

Trim(*String***)** – removes unwanted spaces from left and right of the string supplied and extra spaces in between multiple strings

```
Sub StringExample1()
    Dim strString As String
    strString = "Microsoft Excel VBA"
    'Returns 17 (17th character starting from first character)
    MsgBox InStr(1, strString, "V", vbTextCompare)
    'Returns 7 (7th character from left starting 'at the sixth position)
    MsgBox InStr(6, strString, "o", vbTextCompare)
End Sub
```

```
Sub StrngExample2()
```

```
    MsgBox Format(12.5 * 1.175, "£0.00")
End Sub
```

Which Library?

Some functions in VBA may appear to be duplicates to functions known in Excel.

```
Sub WhichLibrary()
    'Excel's Round function
    MsgBox Application.WorksheetFunction.Round(10.2356, 2)
    'VBA's Round function
    MsgBox Round(10.2356, 2)
End Sub
```

The above example uses two functions that appear in both VBA and Excel.

By default, VBA is the library used when calling such a function.

To use the **Round()** function from Excel's library, users need to call the qualification (object hierarchy) first.

> The MsgBox statement and function and InputBox function are members of the **Interaction** class in the VBA library.

Next Topic: Creating Add-Ins

Creating Add-Ins

Add-In's are wrapped procedures in an independent file for distributing and loading into other Excel systems.

When creating an Add-In, prepare all the procedures, objects and other elements as you would normally prepare a spreadsheet and test it out thoroughly.

Therefore, it is not uncommon to have a normal Excel file (**xls/xlsx/xlsm/xlsb**) and eventually an add-in file (**xla/xlam**) of the same information. From the '*xls'* file, users save as an '*xla'* file that allows for easier upgrade and maintenance.

An Add-In file can reside anywhere as the user can control where to point and load the file.

An Add-In file is in essence and invisible loaded file which can not physically be edited.

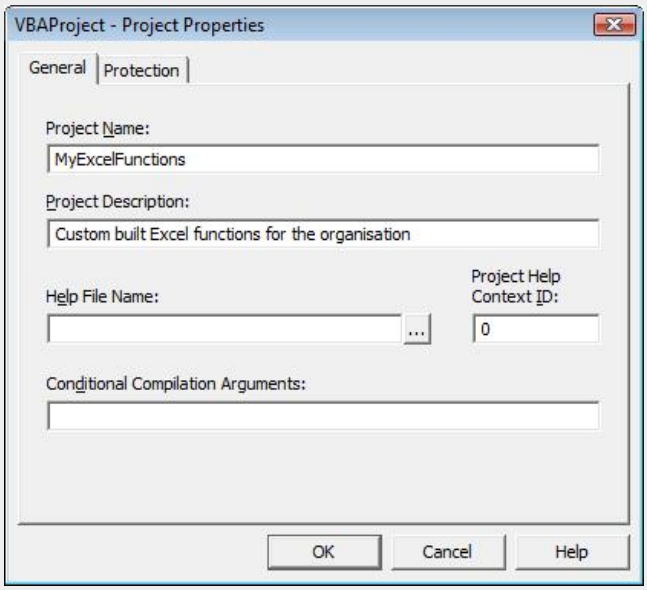
Creating the Add-In file

Prepare all the code including any functions, forms and other objects required for distribution.

The saving action is within the Excel interface (as you normally save a file) but you may want to change a few properties within the **module** especially applying a password protection to your code.

In the module, right-mouse click the **VBAProject** node in the Project Explorer view and choose **VBAProject Properties...** from the pop-up menu.

In the first tab, set the narratives as required.



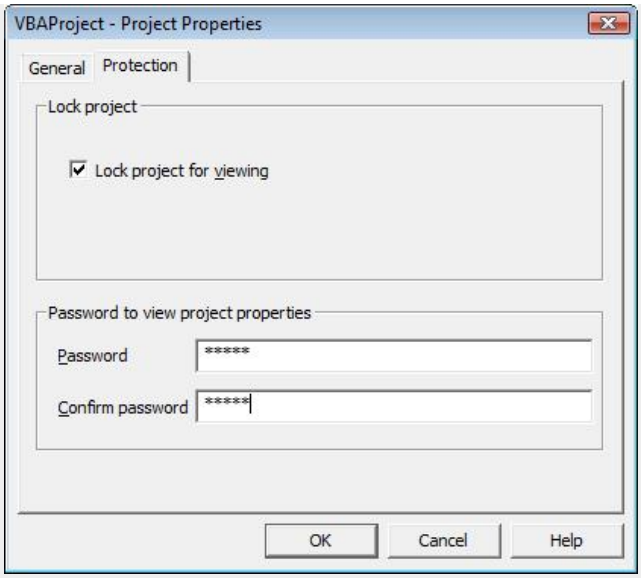
(**Note**: The **Project Name** property can not contain any spaces).

Click on the Protection tab and set a password which is case sensitive and make sure you enable the tick **Lock project for viewing**.

Tip: To switch between Excel interface and VBE window use **ALT + F11** shortcut keys.

Use **F12** function in Excel to run the 'Save As' command.

VBA Keywords: Application, CommandBars, Controls, Caption, Add, OnAction and Delete.



Choose the **OK** button to apply.

The next time a user tries to view the code by expanding the node they will be prompted for a password.

Back in Excel, save your current file (xls/xlsx/xlsm) as an open copy before choosing save as (**F12** function key).

Choose the **Add-In** file type and choose a location to store you copy file.

A copy is generated and the original remains as your working copy which is required should you want to re-generate a newer version.

All Add-In files are read-only and therefore can not be edited and saved.

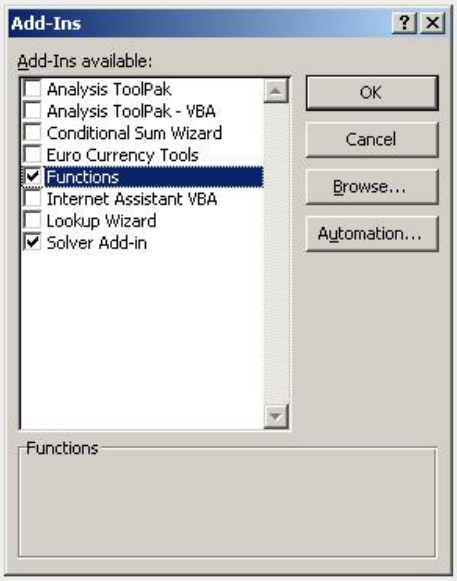
Loading the Add-In file

To load and install the file you use the Add-In manager tool (and not file, open).

Click on the Office button and choose **Excel Options** at the bottom of the menu.

Click on the **Add-Ins** category (on the left pane) followed by the **Go...** button (at the bottom of the main screen.

You are now taken to the **Add-Ins** dialog box.



Select the Add-In file and click **OK**.

Every time Excel starts, it loads all the Add-In files in the background enabling the functionality from your file and is seamless to the application (and user).

Add-In Workbook Events

Special reserved events exist for an **Add-In** file which is executed as the **Add-In** is loaded and unloaded into Excel.

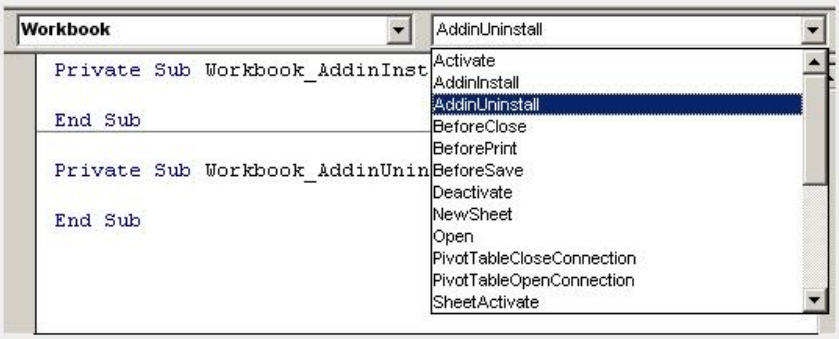
Developers typically use these events to control the initialisation and resetting of command bars and

menus amongst other object changes.

The two events are:

1. Workbook_AddinInstall()
2. Workbook_AddinUninstall()

The events can be found by loading the module to **ThisWorkbook** (node) and choose Workbook from the Object drop-down control.



On the right hand drop down box (*Procedure*), scroll for the above two events and type the code required.

Example use for the above events:

```
'Creates a menu item too the Tools menu
Private Sub Workbook_AddinInstall()
    Dim cb As CommandBarControl
    Set cb = Application.CommandBars("Tools") _
          .Controls.Add(Type:=msoControlButton)
    With cb .BeginGroup = True
        .Caption = "My Report..."
        .FaceId = 0
        .OnAction = "MyReport"
    End With
End Sub
```

The above procedure creates and appends a menu item to the existing **Tools** menu (Ribbon Bar).

The **OnAction** property assigns the macro procedure that is called when the item is clicked.

```
'Removes a menu item from the Tools menu
'If it can be found.
Private Sub Workbook_AddinUninstall()
    Dim cb As CommandBarControl
    Dim cbt As Integer, i As Integer
    i = 1 cbt = Application.CommandBars("Tools").Controls.Count
    Do Until i > cbt
        Set cb = Application.CommandBars("Tools") _
            .Controls.Item(i)
        If cb.Caption = "My Report..." Then
            cb.Delete
            Exit Do
        End If
    Loop
End Sub
```

The above procedure removes the custom menu item if it can be found, resetting the Tools menu bask to default.

Both procedures should be completed with error handling procedures to prevent unnecessary errors occurring.

(**Note**: The above example code to install and uninstall was based on Excel version 2003).

Next Topic: Variables & constants

Variables & Constants

Variables

A variable is a placeholder, which stores data, i.e. a storage area in memory. It can be recalled, reassigned or fixed throughout a procedure, function or during the lifetime of a module being executed.

Structure (syntax)

```
Variable Name = Value
Variable Name = Object.Property
```

For example:

```
Result = Activecell.Value
```

Where **`Result`** is the variable name which is assigned the value in the `Activecell`.

Declaring a Variable

Declaring a variable allows you to state the names of the variables you are going to use and also identify what type of data the variable is going to contain.

For example, if **`Result = 10`**, then the variable **`Result`** could be declared as being an **Integer**.

Explicit Declaration

Explicit Declaration is when you declare your variables to be a specific data type.

Variables can be declared as one of the following data types:

1. Boolean
2. Byte
3. Integer
4. Long
5. Currency
6. Single
7. Double
8. Date
9. String (for variable-length strings)
10. String * length (for fixed-length strings)
11. Object
12. Variant

Note that if you do not specify a data type, the **Variant** data type is assigned by default.

The Declaration Statement is written as follows:

```
Dim Result As Integer
Dim MyName As String
Dim Sales As Currency
Dim Data
```

Example:

The following example declares the `MySheet` variable to be a **String**.

```
Sub VariableTest()
    Dim MySheet As String
    MySheet = ActiveSheet.Name
    ActiveCell.Value = MySheet
End Sub
```

The position of the declaration statement is important as it determines whether the variable is available for use throughout the module or just within the current procedure (see *Understanding Scope & Visibility* later in this section).

Another Example:

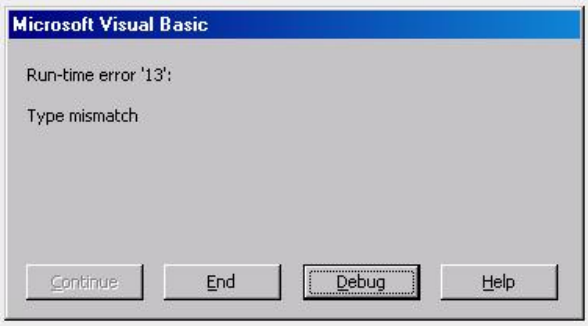
Tip: Remember to enable **Option Explicit** (it's good practice).

VBA Keywords: ActiveCell, ActiveSheet, ActiveWorkbook, MsgBox, Dim, Private, Public, ByVal, ByRef, Round.

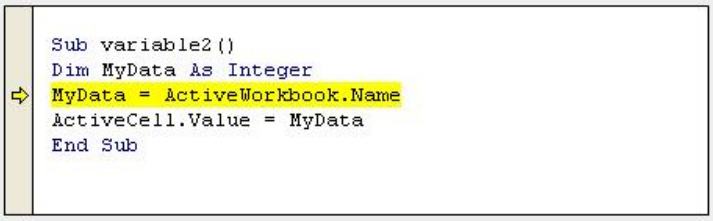
The following example declares the `MyData` variable to be an **Integer**.

```
Sub VariableTest()
    Dim MyData As Integer
    MyData = ActiveWorkbook.Name
    ActiveCell.Value = MyData
End Sub
```

However, when you run this macro, an error will occur because `MyData = ActiveWorkbook.Name` is invalid since `ActiveWorkbook.Name` is not an **Integer** but a **String**.



When you click on the **Debug** button, it will highlight incorrect code.



But if you declare `MyData as` **`String`** (i.e. text) then `MyData = ActiveWorkbook.Name` will become valid.

```
Dim MyData As String
```

Benefits of using Explicit Declaration

If you do not specify a data type, the **Variant** data type is assigned by default. Variant variables require more memory resources than most other variables. Therefore, your application will be more efficient if you declare the variables explicitly and with a specific data type.

Explicitly declaring all variables also reduces the incidence of naming-conflict errors and spelling mistakes.

Following the conventions set by Microsoft (see Rules and Conventions), variables too have a standard that in most cases is recommended.

A variable declared explicitly, should have a prefix in lowercase preceding the variable meaningful name. For example, a variable to store the vat amount of type **Double** may be shown as **dblVatAmount** where '**dbl**' is the prefix for a **Double** data type and the two word variable (initial capped) referring to the purpose of the variable.

The table below shows what prefixes could be used for each common data type:

Data type	Prefix	Example
Boolean	bln	blnContinue
Byte	byt	bytWeekdayValue
Collection object	col	colWidgets
Currency	cur	curCost
Date (Time)	dtm	dtmOrderDate
Double	dbl	dblRevenue
Error	err	errInvoiceNo
Integer	int	intQuantity
Long	lng	lngDistance
Object	obj	objWordDoc
Single	sng	sngVatRate
String	str	strContact
Variant	vnt	vntColumnData or varColumnData or ColumnData (*no prefix*).

By default, all variables not explicitly defined are **Variant** (the *largest memory allocation reserved*).

The exception to the above (*developers can decide whether to follow the above or not*) is when defining **Sub** and **Function** procedures with **arguments**. As part of the signature of such a procedure, it can be clearer to the end user to see meaningful named arguments rather than the conventions

stated in the above table.

For example, ***which is clearer for the end user to understand?***

```
Function GrossTotal(Net As Double, Qty As Long, Optional
                                          VatRate As Single) As Double
    ..........
End Function
```

```
Function GrossTotal(dblNet As Double, lngQty As Long, Optional
                                          sngVatRate As Single) As Double
    ..........
End Function
```

When calling this function within Excel, users will see the arguments via the **Insert Function** paste command:

Constants

Constants are values that do not change. They prevent you from having to repeatedly type in large pieces of text.

The following example declares the constant **MYFULLNAME** to equal "Ben Beitler". Therefore, wherever **MYFULLNAME** has been used, the value that will be returned will be "Ben Beitler".

```
Sub ConstantTest()
    Const MYFULLNAME As String = "Ben Beitler"
    ActiveCell.Value = MYFULLNAME
    ActiveCell.EntireColumn.AutoFit
End Sub
```

(**Note**: When using a constant, the convention is normally in uppercase).

Implicit Declaration

As previously mentioned, if you do not declare your variables and constants, they are assigned the **Variant** data type, which takes up more resources and spelling mistakes are not checked.

A **Variant** Variable/Constant can contain any type of data.

```
Data = 10
Data = "Fred"
Data = #01/01/2010#
```

When you run the following macro, the value in the active cell will be 10.

```
Sub ImplicitTest()
    data = 10
    ActiveCell.Value = data
End Sub
```

When you run the following macro, the value in the active cell will be Fred.

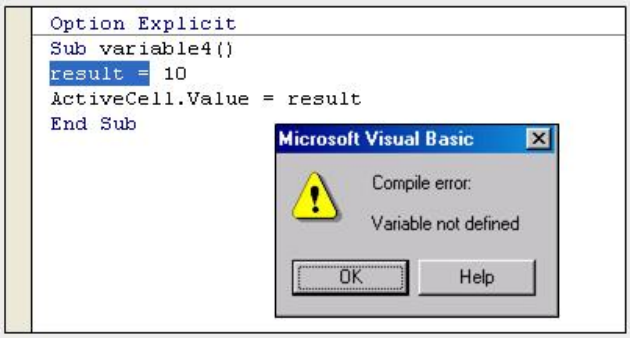
```
Sub ImplicitTest()
    data = "Fred"
    ActiveCell.Value = data
End Sub
```

This can lead to errors and memory abuse though VBA is relaxed in using variables this way - *it's just not good practice!*

Option Explicit (Declaration)

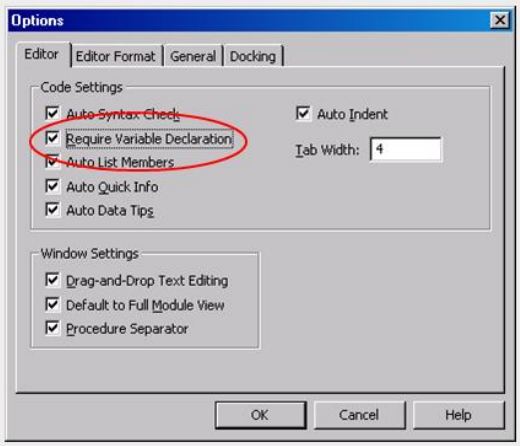
If you type **Option Explicit** at the top of the module sheet, you must declare all your variables and constants.

If you don't declare your variables/constants you will get the following message.



If you wish to ensure that **`Option Explicit`** is always entered at the top of the module:

1. Go into the Visual Basic Editor.
2. Click on the **Tools** menu, select **Options...**
3. Click on the **Editor** tab and select "**Require Variable Declaration**".



You now must always use the **`Dim`** keyword to declare any variable.

Understanding Scope & Visibility

Variables and procedures can be visible or invisible depending on the following keywords used:

1. **Private**
2. **Public**
3. **Static**
4. **Friend** (*Class Modules only – not covered in this guide*)

Depending where users use the above keywords, visibility can vary too within a module, class module or user-form.

In a standard module when using the keyword **`Dim`** to declare a variable.If the variable is outside a procedure and in the main area of the module, this variable is automatically visible to all procedures in that module. The lifetime of this variable is governed by the calling procedure(s) and can be carried forward into the next procedure within the same module.

If a variable declared with the **`Dim`** keyword is within a procedure, it is visible for that procedure only and the lifetime of the variable expires when the procedure ends.

The **`Dim`** keyword can be used in either the module or procedure level and are both deemed as private to the module or procedure.

Instead of using the **`Dim`** keyword, it is better practice to use the **`Private`** keyword when declaring variables at the module level. Users must continue to use the **`Dim`** keyword within a procedure.

Using **`Public`** to declare a variable at the module level is potentially unsafe as it is exposed beyond this module to all other modules and user-forms. It may also provide confusion if the two variables with the same name exist across two modules. When a variable is declared **`Public`** , users should take caution and try and be explicit in the use of the variable.

For example:

Module A

```
Option Explicit
Public intMonth As Integer
code continues.........
```

Module B

```
Option Explicit
Private intMonth As Integer


Sub ScopeVariables()
    intMonth = 10 'This is ModuleB's variable
```

```
    ModuleA.intMonth = 10 'This is ModuleA's variable (explicit)
End Sub
code continues.....
```

Two variables with the same name and data type were declared in both module A and B. A procedure in module B calls the local variable and then explicitly calls the public variable declared in module A. Users must therefore use the notation of the name of the module followed by the period separator (**.**) and its variable.

Public and **Private** keywords can also be applied to a procedure. By default, in a standard module, all procedures are **Public** unless explicitly defined as **Private**.

It is good practice to apply either **Public** or **Private** for each procedure as later releases of Visual Basic may insist on this convention.

If a procedure is **Private**, it can only be used within the module it resides. This is particularly designed for internal procedures being called and then discarded as part of a branching routine (*nested procedures*).

If users mark a procedure as **Private**, it cannot be seen in the macros dialog box in Excel.

Static Variables

Using the **Static** keyword allows users to declare variables that retain the value from the previous calling procedure.

Example using `Dim`**:**

```
'Standard variable (non-static).
Sub TotalValue()
    Dim intValue As Integer
    intValue = 10
    Msgbox intValue
End Sub
```

Example using `Static`:

```
'Standard variable (non-static).
Sub TotalValue()
    Static intValue As Integer
    intValue = 10
    Msgbox intValue
End Sub
```

Running the first example will simply display the value 10 and the variable's value will be lost when the procedure ends.

Running the second example will display the value 10 but it will retain the variable and its value in memory so that when running it once more, the value displayed now equals 20 and so on until the file is either closed or the reset command is executed.

`Static` can only be used in a procedure and is therefore private.

Do not confuse `Static` with `Const` (*constant*).

Use the `Const` keyword to fix a value for lifecycle of the module or procedure. Users will not be able to modify the value at run time as with conventional variables.

Example:

Public

```
'Vat Rate is currently fixed at 17.5%
Public Const VATRATE As Single = 0.175
```

or **Private**

```
'Vat Rate is currently fixed at 17.5%
Const VATRATE As Single = 0.175
```

Using the constant

```
Sub GrossTotal()
    Dim dblNet As Double
    dblNet = 100
    MsgBox Round(dblNet * (1 + VATRATE), 2)
End Sub
```

(**Note**: Round is a VBA function)

It is acceptable to use uppercase convention for constants.

`Const` keyword can be public or private (*private by default*) declared at the module and private only at the procedure level.

User forms, which allow users to design custom form interfaces, also have scope issues using `Private` and `Public`

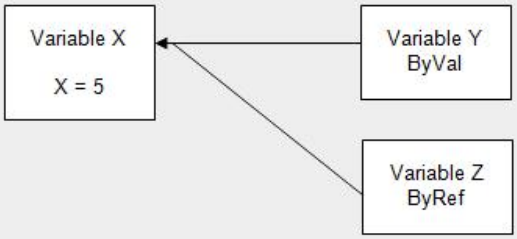
By default, any control's event that is drawn onto a form will be private as the form should be the only object able to call the procedure.

Other event driven procedures, which can be found in a worksheet or a workbook, will also be private by default.

ByVal versus ByRef

Passing arguments is a procedure can be either by value (`ByVal`) or by reference (`ByRef`). Both keywords precede the argument name and data type and if omitted is `ByRef` by default.

When passing an argument by value, the value is passed into a calling procedure and then lost when the procedure ends or reverts back to the original value as it returns to the original procedure.



Variable X is declared and set to the value of 5.

Variable Y is declared and set to the value of 5 (BVal).

Variable Z is declared and set to the reference pointer of variable X that is the value of 5 (ByRef).

Therefore, all three variables are equal.

When the value of variable X changes value (i.e. X = 10), variable Y remains unchanged but variable Z changes to the value of 10.

`ByVal` *example:*

```
Sub CustomSub(ByVal AddTo As Integer)
    AddTo = AddTo + 5
    MsgBox AddTo
End Sub

Sub TestVariable()
    Dim x As Integer
    x = 5
    Call CustomSub(x)
    MsgBox x
End Sub
```

The procedure **TestVariable** starts by seting *x = 5*. It's **CustomSub** procedure is called passing the variable's value of *x* and incremented by 5. The first message box seen is via the **CustomSub** procedure (*shows the value 10*). The second message box is via the **TestVariable** procedure which follows as it returns the focus (shows the value 5). Therefore the **`ByVal`** **`AddTo`** variable stored is lost as it is passed back into the original call procedure (**TestVariable**) resulting in *x* being equal to 5 again.

`ByRef` *example:*

```
Sub CustomSub(ByRef AddTo As Integer)
    AddTo = AddTo + 5
    MsgBox AddTo
End Sub

Sub TestVariable()
    Dim x As Integer
    x = 5
    Call CustomSub(x)
    MsgBox x
End Sub
```

The procedure **TestVariable** starts by seting *x = 5*. It's **CustomSub** procedure is called passing the variable's value of *x* and incremented by 5. The first message box seen is via the **CustomSub** procedure (shows the value 10). The second message box is via the **TestVariable** procedure which follows as it returns the focus (shows the value 10 again). Therefore the **`ByRef`** **`AddTo`** variable stored is not lost as it is passed back into the original call procedure (**TestVariable**) resulting in *x* now being equal to 10.

Next Topic: Object variables

Object Variables

An **Object Variable** is a variable that represents an entire object, such as a Range or a Worksheet.

Object Variables are important because:

- They can simplify the code significantly
- They can make the code execute more quickly.

You use this type of variable for creating a new instance of an object which will be necessary should you wish to communicate with other applications namely Microsoft Word, PowerPoint or any other external library.

Declaring Object Variables

Object Variables, similar to normal variables, are declared with the **Dim** or **Public** statement, for example:

```
Dim mycell As Range.
```

Assigning Object Variables

To assign an object expression to an object variable, use the **Set** keyword.

For example:

```
Set ObjectVariable = ObjectExpression

Set MyCell = Worksheets("Sheet1").Range("A1")
```

Example:

The following procedure will select the cell A1 on Sheet1, input the value 100 and format it with Bold, Italic and Underline.

```
Sub ObjectVariable()
    Worksheets("Sheet1").Range("A1").Value = 100
    Worksheets("Sheet1").Range("A1").Font.Bold = True
    Worksheets("Sheet1").Range("A1").Font.Italic = True
    Worksheets("Sheet1").Range("A1").Font.Underline = XlSingle
End Sub
```

The line of code **Worksheets("Sheet1").Range("A1")** is repeated four times within this procedure.

If we declare an **Object Variable** called *mycell* to be a range, we can then set *mycell* to be equal to **Worksheets("Sheet1").Range("A1")**.

The procedure would then become:

```
Sub ObjectVariable()
    Dim mycell As Range
    Set mycell = Worksheets("Sheet1").Range("A1")
    mycell.Value = 100
    mycell.Font.Bold = True
    mycell.Font.Italic = True
    mycell.Font.Underline = xlSingle
End Sub
```

It is good practice to set all object variables to **Nothing** at the end of the lifecycle of the variable even though Visual Basic will destroy all variables and pointers automatically once the routine has ended.

It is possible to have an object variable allocated to memory after the event of an error occurring. If the error handler allows the procedure to continue, it may be necessary to re-set the same object variable. This is when an object should be destroyed and then re-initialised.

At the end of a procedure, destroy all object variables in the following manner:

```
Set objWSheet = Nothing
```

Note: You may want to search for more information on Object Variables especially to understand the difference between **Late** and **Early Binding**.

Generic objects are useful when you do not know the specific type of object the variable will contain or when the variable contains objects from several different classes.

For example:

```
Dim mycell As Object
Set mycell = Application.Worksheets("Sheet1").Range("A1")
```

Next Topic: Arrays

Arrays

Arrays are a set of indexed elements for the same data type variable. Each element is independent but belongs to the same group variable and is better than have several single variables of the same type.

These are declared like a standard variable with the option of setting a data type and its scope.

Arrays are either declared in design time with the number of elements defined or at run time making it dynamic.

Example of Fixed Array:

```
'Declaring a fixed array
Sub FixedArray()
    Dim strWeek(7) As String
    strWeek(0) = "Sunday"
    strWeek(1) = "Monday"
    strWeek(2) = "Tuesday"
    strWeek(3) = "Wednesday"
    strWeek(4) = "Thursday"
    strWeek(5) = "Friday"
    strWeek(6) = "Saturday"
   ...................
End Sub
```

The value entered between the parentheses defines the number of elements for the array variable starting at point zero. Therefore, for an array variable of seven elements, the starting element number will be zero and the last will be six.

Example of Dynamic Array:

```
'Declaring a fixed array
Sub DynamicArray()
    Dim strWeek() As String
    ...................
    ReDim strWeek(7)

    strWeek(0) = "Sunday"
    strWeek(1) = "Monday"
    strWeek(2) = "Tuesday"
    strWeek(3) = "Wednesday"
    strWeek(4) = "Thursday"
    strWeek(5) = "Friday"
    strWeek(6) = "Saturday"

    ...................
End Sub
```

Dynamic arrays allow array variables to grow during the run time of the procedure. This may be required, as the process may not know the full size of the intended variable.

The keyword **ReDim** allows array variables to be re-declared to a new size. The above example declares an array variable of unknown size and then uses the **ReDim** command to redefine the size.

When using the **ReDim** command, any previous sizes and values that may be present are lost and set to nothing.

In the event of preserving the previous size array and wanting to extend the size, users can use **Preserve** keyword.

```
ReDim Preserve strWeek(14)
```

Multi-Dimension Array

Arrays can also be multi-dimensioned and can store up to 60 sets of elements.

```
Sub MulitArray()
    Dim intMulti(1 To 5, 1 To 10) As Integer
    intMulti(1, 1) = 10
    intMulti(2, 1) = 20
    intMulti(3, 1) = 30
    intMulti(4, 1) = 40
    intMulti(5, 1) = 50
    intMulti(1, 2) = 60
    intMulti(2, 2) = 70
    ........................
    intMulti(5, 10) = 500
```

Note: When you work with arrays (variables), users like to use the distinction between groups and single variables where a single variable is also known as a ***scalar*** variable.

VBA Keywords: MsgBox, IsArray, UBound, LBound, Array, Option Base 1.

```
    ......................
End Sub
```

Each dimension group has been set to start at 1 (instead of the default 0). In fact users can start at any integer value providing the stop value is greater than the start value.

Array Function

Another way to set values to an array variable is to use the VBA **Array** function:

```
Sub ArrayFunctionExample()
    Dim strWeek As Variant
    Dim strDay As String
    strWeek = Array("Sunday", "Monday", "Tuesday", _
              "Wednesday", "Thursday", "Friday", "Saturday")
    strDay = strWeek(2) ' strDay now contains "Tuesday".
    ......................
End Sub
```

The **Array** function returns a **Variant** data type and therefore must be declared as a variant. Each element can be converted into a string type as required. The first item in the **Array** function equals element zero and so on.

Setting Option Base

In some programming languages, it is not uncommon to have the first element of an array to be equal to one instead of zero. In situations that the procedure needs to simulate this environment, users can use the following statement:

```
Option Base 1
```

This is declared at the top of the module before the first procedure and affects the entire module. The value 1 changes the base element to one.

IsArray Function

This function can be used to test if a variable is an array and return either `True` or `False`.

```
Sub ArrayFunctionExample()
    Dim strWeek As Variant
    Dim strDay As String
    strWeek = Array("Sunday", "Monday", "Tuesday", _
              "Wednesday", "Thursday", "Friday", "Saturday")
    MsgBox IsArray(strWeek) 'True.
    MsgBox IsArray(strDay) 'False.
    ......................
End Sub
```

UBound and LBound Functions

Both bound functions return a **Long** value of the highest (*upper*) and the lowest (*lower*) element number for an array variable.

Two arguments, one optional:

Variable = **UBound**(ArrayName [, Dimension])

Variable = **LBound**(ArrayName [, Dimension])

The optional second argument only applies if the array is a multi-dimensioned array variable.

```
Sub UpperArray()
    Dim intMulti(1 To 5, 1 To 10) As Integer
    Dim intUpper As Integer
    intMulti(1, 1) = 10
    intMulti(2, 1) = 20
    intMulti(3, 1) = 30
    intMulti(4, 1) = 40
    intMulti(5, 1) = 50
    intMulti(1, 2) = 60
    intMulti(2, 2) = 70
    ......................
    intMulti(5, 10) = 500
    ......................
    MsgBox UBound(intMulti, 1) 'shows 5
    MsgBox UBound(intMulti, 2) 'shows 10
    ......................
End Sub
```

Next Topic: Collections

Collections

VBA Keywords: MsgBox, For...Each, If...Then, Worksheets, Workbooks, ThisWorkbook, Close, Sheets and Add.

Using the analogy of arrays that have elements, a collection is also a group of elements referring to objects in Excel.

Different types of collections exist to define a group of elements for the individual object of the same type.

Examples of Collections:

- A workbook is a member of the collection – Workbooks.
- A worksheet is a member of the collection – Worksheets.
- A cell is a member of the collection – Cells (*Range*).
- A range of cells is a member of the collection – Ranges.
- A command button is a member of the collection – Controls.

A good indicator as to whether a collection exists is to look in the Object Browser (**F2** function key) and scroll down the **Classes** section to view any class file which is a plural. This will more than likely be a collection of the singular named class i.e. Workbooks and Workbook.

There are many others types of collections – *refer to Excel VBA help for more information.*

Example1:

```
'Working through the active workbook and identifying all worksheets
Sub HowManyWorksheets()
    Dim w As Worksheet
    For Each w In Worksheets
        MsgBox w.Name
    Next w
End Sub
```

The variable **w** is explicitly declared as a worksheet object. Using a ***For...Loop*** statement, we can iterate through each element (**w**) until the collection is completed.

Using the message box, one of the element's properties (**Name**) simply displays each worksheet name.

Even if the variable **w** is implicit, it would still understand what variable **w** was because it becomes a member of the collection Worksheets.

Example2:

```
'This example saves changes to and closes all workbooks except
'the one that's running the example.
Sub CloseWorkbooks()
    Dim w As Workbook
    For Each w In Workbooks
        If w.Name <> ThisWorkbook.Name Then
            w.Close savechanges:=True
        End If
    Next w
End Sub
```

The above example will close all workbooks in Excel excluding the active workbook and automatically save any changes.

Like arrays, elements in a collection can also be referred to directly as an independent item. For example, to refer to the first worksheet in a workbook:

```
Sheets(1).Name
Worksheets(1).Name
```

The array element starts at 1 and increments for each member known to the collection. An error will happen if the element number is not known (*a zero or a number higher than the upper bound number*).

There are many methods and properties to many objects as the Object Browser has shown. Collections are being used in a lot of situations without the user even being aware.

For example:

```
Worksheets.Add Count:=2, Before:=Sheets(1)
```

The **Add** method and some of its arguments inserts two new worksheets to the collection worksheets and places them before the first element of the worksheets collection.

Next Topic: Message Box

VBA Keyword: MsgBox

Message Box (MsgBox)

A **Message Box (MsgBox)** displays a message alert box with optional set of buttons, icon and other arguments settings

.

There are two types of message boxes:

1. **MsgBox Statement** - *One way communication to the user.*

```
MsgBox prompt,[buttons],[title],[helpfile],[context]
```

2. **MsgBox Function** - *Two way communication which the system returns a value.*

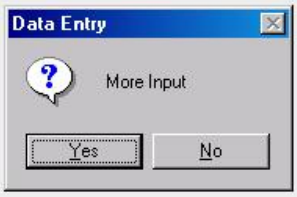
```
Variable = MsgBox(prompt,[buttons],[title],[helpfile],[context])
```

Example1 - MsgBox Statement:

```
Sub MessageBoxTest()
    'brackets are not required for a statement
    MsgBox "The task is now complete"
End Sub
```

Example2 - MsgBox Function:

```
Sub MessageBoxTest ()
    Dim answer
    answer = MsgBox("More Input", vbYesNo + vbQuestion, "Data Entry")
End Sub
```



In the above example, the **vbYesNo** is the command used to create the Yes and No buttons and the + **vbQuestion** is the command to create the Question Mark image.

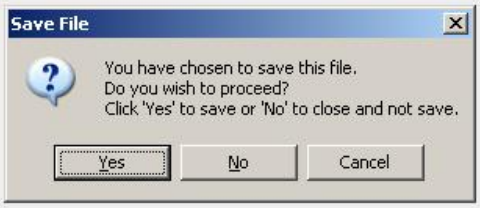
§ If the user clicks on Yes, the message box will return the constant **vbYes**.

§ If the user clicks on No, the message box will returns the constant **vbNo**.

Example2 - MsgBox Function (Multiple Lines):

```
Sub MultiLineMessageBox()
    Dim intResponse As Integer
    intResponse = MsgBox("You have chosen to save this file." _
        & vbNewLine & "Do you wish to proceed?" & vbNewLine & _
        "Click 'Yes' to save or 'No' to close and not save.", _
        vbQuestion + vbYesNoCancel, "Save File")
    ....................................
End Sub
```

The above example will display multiple lines in the message box using the constant **vbNewLine**.



Constants for MsgBox

Buttons and icons are combined for the **Buttons** argument which have a unique value that drives the output of how users use and see button combinations:

Constant	Value	Description
vbOKOnly	0	Display **OK** button only.
vbOKCancel	1	Display **OK** and **Cancel** buttons.
vbAbortRetryIgnore	2	Display **Abort**, **Retry**, and **Ignore** buttons.
vbYesNoCancel	3	Display **Yes**, **No**, and **Cancel** buttons.
vbYesNo	4	Display **Yes** and **No** buttons.
vbRetryCancel	5	Display **Retry** and **Cancel** buttons.
vbCritical	16	Display **Critical Message** icon.
vbQuestion	32	Display **Warning Query** icon.
vbExclamation	48	Display **Warning Message** icon.
vbInformation	64	Display **Information Message** icon.
vbDefaultButton1	0	First button is default.
vbDefaultButton2	256	Second button is default.
vbDefaultButton3	512	Third button is default.
vbDefaultButton4	768	Fourth button is default.
vbApplicationModal	0	Application modal; the user must respond to the message box before continuing work in the current application.
vbSystemModal	4096	System modal; all applications are suspended until the user responds to the message box.
vbMsgBoxHelpButton	16384	Adds Help button to the message box.
VbMsgBoxSetForeground	65536	Specifies the message box window as the foreground window.
vbMsgBoxRight	524288	Text is right aligned.
vbMsgBoxRtlReading	1048576	Specifies text should appear as right-to-left reading on Hebrew and Arabic systems.

The following applies to the **MsgBox** Function when the user clicks a button returning a unique value.

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

While a the **MsgBox** is being displayed, the macro procedure is paused waiting for the user to click a button whether it is a statement or a function.

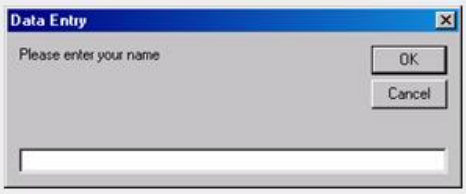
Note the difference between the two types regarding when parenthesis are used and can be ignored. Also, be aware any function must be placed to the right side of an **=** (equal sign) because it returns an answer.

Next Topic: VBA Input Box

VBA Keyword: InuptBox, ActiveCell, If...Then, Exit Sub

VBA Input Box

An **Input Box** (**InputBox**) is a function that allows the user to enter a value into a dialog box. The result of an Input Box is stored normally to a variable and remebered for later use in a procedure. Note that the result of an Input Box is always returns a **String** value.



Structure (syntax):

```
Variable = InputBox (Prompt, [Title], [Default], [XPos], [YPos])
```

The Arguments for an **InputBox**:

Prompt	Text on the Input Box
Title	Title bar text (optional)
Default	Default value of the Input Box (optional)
XPos/Ypos	Position of the Input Box. If you leave them blank, the Input Box will appear in the centre of the screen (optional)

Example1 - Text Input Box:

```
Sub Box1()
    Dim strMyName As String
    strMyName = InputBox("Please enter your name", "Data Entry")
    ActiveCell.Value = "My name is " & strMyName
End Sub
```

If you click on the Cancel button, it will return an empty string " " so the result will be "My name is"

Example2 - Using named arguments:

This allows you to put the arguments in any order.

```
Sub Box2()
    Dim strResult As String
    strResult = InputBox(prompt:="Please enter amount", _
                                        Title:="Data Entry")
    ActiveCell.Value = strResult
End Sub
```

You really need to handle the **Cancel** button which always returns an empty **String**. Even if you click the **OK** button with no value this too will return an empty **String**.

In most cases, the following code should be added immediately after the **InputBox** function call:

```
If [Variable] = Empty Then Exit Sub
```

The above piece of code will terminate the procedure if the **String** variable is empty.

So the the previous example would look like:

```
Sub Box2()
    Dim strResult As String
    strResult = InputBox(prompt:="Please enter amount", _
                                        Title:="Data Entry")
     If strResult = Empty Then Exit Sub 'Terminates here if empty
     ActiveCell.Value = strResult
End Sub
```

While a the **InputBox** function s being displayed, the macro procedure is paused waiting for the user

to click a button.

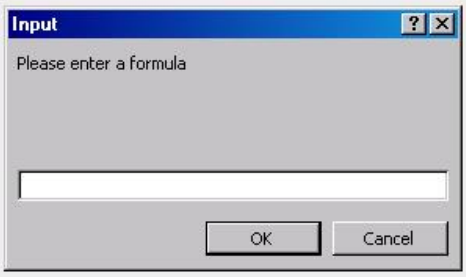
Next Topic: Excel Input Box

Want to teach yourself Access? Free online guide at About Access Databases

Excel Input Box

VBA Keyword: InuptBox, Application and ActiveCell

An **Excel Input Box (InputBox)** function is different from a VBA Input Box because you can specify what you would like the result of the Input Box to be. If the **Type** argument is omitted, the Input Box will return at text (**String**) value.



Structure (syntax):

```
Variable = Application.InputBox (Prompt, [Title], [Default],
           [XPos],[Ypos], [HelpFile], [HelpContextID], [Type])
```

Prompt	Text on the Input Box
Title	Title bar text (optional)
Default	Default value of the Input Box (optional)
XPos/Ypos	Position of the Input Box. If you leave them blank, the Input Box will appear in the centre of the screen (optional)
HelpFile	Associated help document attachment (optional)
HelpContextID	Unique identifier for the help document - bookmark (optional)
Type	Defines what data type to return (optional)

The following **Types** may be used:

Value	Meaning
0	A formula
1	A number
2	Text (a string)
4	A logical value (True or False)
8	A cell reference, as a Range object
16	An error value, such as #N/A
64	An array of values

If you wish the **Input Box** to accept both text and numbers, set the **Type** argument to 1 + 2.

To call the **Excel InputBox** and not the standard VBA InputBox, you need to call the **Application** object keyword which calls this function from the Excel library where it belongs.

```
Application.InputBox(...
```

For the following examples, we will declare the variables as Variant.

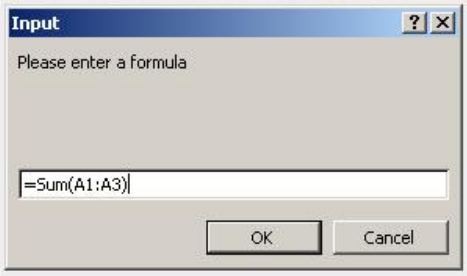
Example1 - Text Input Box:

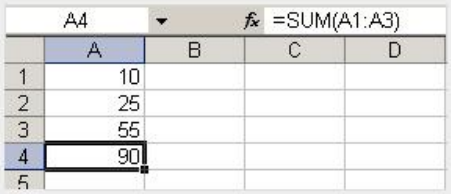
```
Sub Box1()
    Dim x
    x = Application.InputBox("Please enter a number", , , , , , , 1)
    ActiveCell.Value = x
End Sub
```

Example2 - Formula Input Box:

```
Sub box4()
    Dim y
```

```
    y = Application.InputBox("Please enter a formula", , , , , , , 0)
    ActiveCell.Value = y
End Sub
```





While a the **Excel InputBox** function s being displayed, the macro procedure is paused waiting for the user to click a button.

Next Topic: Making Decisions (If...Then...Else...End If)

Want to teach yourself Access? Free online guide at About Access Databases

Making Decisions (If...Then...Else...End If)

This is one of the '*control flows*' VBA provides which is required to make your VBA code procedures more flexible, reduce the amount of code required and make the system think for itself!

Without any of these '*control flows*', your code is very linear and rigid which never is really suitable when trying to mimic '*real world*' processes.

The four '*control flows*' are:

1. If...Then...Else...End If
2. Select Case...End Select
3. Do...Until/While...Loop
4. For...Counter/Each...Next

If Statement

This is the logical (conditional test) statement that runs code the yields either a `True` or `False` value.

There are four flavours of the **`If`** I have referred to help understand how this structure is implemented in VBA:

1. 'One Line' If
2. 'True' If
3. 'Standard' If
4. 'Multiple/Nested' If

A logical test can be a value, expression, function or an object property that returns a `True` or `False` value using the logical operators <, >, <=, >=, =, <>, Not
(*see* Formulas).

One Liner If

As its name suggests, it's all on one line and requires no **`End If`** block.

Structure (syntax):

```
If Condition [= True] Then Execute code if true
```

It is used as a quick way to add an extra single calling command or calling an additional procedure if the condition is true.

I use it as a test to see if the procedure should continue and if not to terminate here.

```
Sub OneLinerIfTest()
    Dim strResult As String
    strResult = InputBox(prompt:="Please enter amount", _
                                          Title:="Data Entry")
     If strResult = Empty Then Exit Sub 'Terminates here if empty

     Continues here if not empty...

End Sub
```

True If

This is used in include extra code if true but is used in an **`If`** block so that multiple lines of code can be applied here.

Tip: Revert to the **Select Case** statement for multiple **If's** that exceed 5 condtiotns.

VBA Keywords: ActiveCell, Font, Bold, Italic, Underline, Color, ClearFormats, InputBox IsNumeric and Exit Sub.

Structure (syntax):

```
If Condition [= True] Then
    Execute multiple lines of code if true
    .....
End If
```

```
Sub TrueIfTest()

    Code runs here first before it enters the If block...

    If ActiveCell.Value < 0 Then
       ActiveCell.Font.Bold = True
       ActiveCell.Font.Color = vbRed

       Additional code continues here only if true...

    End If

    Code continues here whether true or not!...

End Sub
```

Standard If

This is more common type of If block as it provides a `True` and `False` option and will therefore (logically) choose one procedure to call/run.

This can be compared to the more familiar If function in Excel where users specify a `True` and False returning value.

Structure (syntax):

```
If Condition [= True] Then
    Execute multiple lines of code if true
    .....
Else
    Execute multiple lines of code if false
    .....
End IF
```

```
Sub StandardIfTest()

    Code runs here first before it enters the If block...
    ActiveCell.ClearFormats

    If ActiveCell.Value < 0 Then
       ActiveCell.Font.Bold = True
       ActiveCell.Font.Color = vbRed

       Additional code continues here only if true...

    Else 'FALSE
       ActiveCell.Font.Italic = True
       ActiveCell.Font.Color = vbRed

       Additional code continues here only if false...

    End If

    Code continues here whether true or false...

End Sub
```

The Else keyword is the new addition and acts as the **`False`** (the catch) should the **`true`** fail.

Multiple/Nested If

What about have more than one set of true conditions with a false (as a catch)?
Nested or multiple If's can be as many as required and run in order of their logical conditions.

Structure (syntax):

```
If Condition 1 [= True] Then
    Execute multiple lines of code if true (1)
    .....
ElseIf Condition 2 [= True] Then
    Execute multiple lines of code if true (2)
    .....
ElseIf Condition N [= True] Then
    Execute multiple lines of code if true (N)
    .....
Else
    Execute multiple lines of code if false
    .....
End If
```

```
Sub MultipleIfTest()

    Code runs here first before it enters the If block...
    ActiveCell.ClearFormats

    If ActiveCell.Value < 0 Then
       ActiveCell.Font.Bold = True
       ActiveCell.Font.Color = vbRed

       Additional code continues here only if true 1...

    ElseIf ActiveCell.Value = 0 And ActiveCell.Value <= 100 Then
       ActiveCell.Font.Underline = xlSingle
       ActiveCell.Font.Color = vbBlue

       Additional code continues here only if true 2...

    Else 'FALSE - catch for non true values
       ActiveCell.Font.Italic = True
       ActiveCell.Font.Color = vbRed

       Additional code continues here only if false...

    End If

    Code continues here whether true or false...

End Sub
```

The first condition is tested and if `True` stops and runs code in that block. If the first condition is `False` then the second If test condition is tested. Therefore, the second condition is only executed if the first condition failed.

You can use the **`ElseIf`** keyword as many times for each separate new condition but if you intend to have more than five different conditions then switching to the Select Case statement is the better practice as it's quicker and cleaner to write.

A nested `If` is one which starts a new block inside another `If` block:

```
Sub NestedIfTest()

    Code runs here first before it enters the If block...
    ActiveCell.ClearFormats

    If IsNumeric(ActiveCell.Value) Then 'Is it a number?

       'Nested If Block inside the first If Block
       If ActiveCell.Value < 0 Then
           ActiveCell.Font.Bold = True
           ActiveCell.Font.Color = vbRed
     Else
```

```
            ActiveCell.Font.Bold = False
            ActiveCell.Font.Color = vbBlack
        End If

        Additional code continues here only if true...

    Else 'FALSE
        ActiveCell.Font.Italic = True
        ActiveCell.Font.Color = vbRed

        Additional code continues here only if false...

    End If

    Code continues here whether true or false...

End Sub
```

Next Topic: Making Decisions (Select Case...End Select)

Making Decisions (Select Case...End Select)

VBA Keywords: ActiveCell, Font, Bold, Italic, Underline, Color, ClearFormats, IsNumeric and Exit Sub.

This is one of the '*control flows*' VBA provides which is required to make your VBA code procedures more flexible, reduce the amount of code required and make the system think for itself!

Without any of these '*control flows*', your code is very linear and rigid which never is really suitable when trying to mimic '*real world*' processes.

The four '*control flows*' are:

1. If...Then...Else...End If
2. Select Case...End Select
3. Do...Until/While...Loop
4. For...Counter/Each...Next

Select Case statement

This is an alternative way to write logical statements and designed for multiple and similar conditions tested in a *look up* table together.

It is deemed faster than a conventional If statement and more clinical to write and understand.

Structure (syntax):

```
Select Case Grade/Expression

    Case Value 1
        Execute multiple lines of code if true 1
        .....

    Case Value 2
        Execute multiple lines of code if true 2
        .....

    Case Value 3
        Execute multiple lines of code if true 3
        .....

    Case Value N
        Execute multiple lines of code if true N
        .....

    Case Else

        Execute multiple lines of code if false
        .....

End Select
```

`Case Value` is the value being test logically against the ***Grade*** or ***Expression*** for a true match. It continues down the list in order until it finds a true match with a catch using `Case Else` as the false option.

```
Sub SelectCaseExample()

    'Code runs here first before it enters the Select Case block...
```

```
    Select Case ActiveCell.Value

        Case Is < 0 'Value tested to see if grade < 0
            ActiveCell.Font.Bold = True
            ....
        Case Is < 10 'Value tested to see if grade < 10
            ActiveCell.Font.Italic = True
            ....
        Case Is < 100 'Value tested to see if grade < 100
            ActiveCell.Font.Color = vbRed
            ....
        Case 100 To 200 'Value tested to see grade is in range 100 - 200
            ActiveCell.Font.Color = vbRed
            ....
        Case 201, 203, 205 'Value tested to see grade is 201 Or 203 Or 205
            ActiveCell.Font.Color = vbRed
            ....
        Case IsNumeric(ActiveCell.Value) 'Is it a number?
            ActiveCell.ClearFormats

        Case Else 'FASLE (if all the above is not true!)
            ActiveCell.Font.Underline = xlSingle
            ....
    End Select

    'Code continues here after the Select Case block...

End Sub
```

It sits all within the **Select Case** block using the **End Select** to terminate the logical test environment.

Note the keyword **To** meaning a range (the **And** operator) which is much easier to write as a condition. Also, using the **,** (comma) which acts as the **Or** operator only enhances the way you can use this statement quickly.

Next Topic: Looping (Do...Until/While...Loop)

Looping (Do...Until/While...Loop)

This is one of the '*control flows*' VBA provides which is required to make your VBA code procedures more flexible, reduce the amount of code required and make the system think for itself!

Without any of these '*control flows*', your code is very linear and rigid which never is really suitable when trying to mimic '*real world*' processes.

The four '*control flows*' are:

1. If...Then...Else...End If
2. Select Case...End Select
3. Do...Until/While...Loop
4. For...Counter/Each...Next

Do...Until/While...Loop

A **Do...Loop** is used when you wish to repeat a piece of code a number of times.

This type of loop works by using a logical test to determine if the loop should repeat or terminate and move onto the next calling procedure.

There four variations that can be used and they all have slight differences:

Structure (syntax):

UNTIL Keyword

```
Do
    Code executed here...
    ....
    [Exit Do]
Loop Until Condition [= True]

Do Until Condition [= True]
    Code executed here...
    ....
    [Exit Do]
Loop
```

WHILE Keyword

```
Do
    Code executed here...
    ....
    [Exit Do]
Loop While Condition [= True]

Do While Condition [= True]
    Code executed here...
    ....
    [Exit Do]
Loop
```

Whichever keyword structure (UNTIL or WHILE) you use is a personal choice as there is no difference

Tip: Use an **Exit Do** to terminate a block early and speed up your procedures.

To break a loop during running your code use **CTRL + PAUSE/BREAK**.

Save your work before running a looping piece of code!

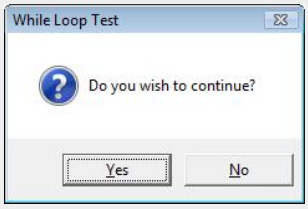
VBA Keywords: ActiveCell, Offset, MsgBox and Exit Do..

in performance or structure; one is the logical inverse of the other.

You can loop **While** a condition is **True/False** or **Until** a condition is **True/False** - it's a simple choice and hopefully the following two examples can make it very clear:

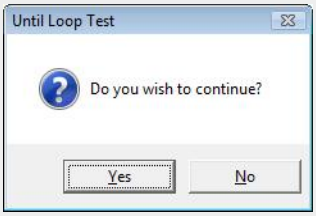
```
Sub WhileConditionLoop()

    Code runs here first before it enters the Loop block...

    Do

       MyResponse = MsgBox("Do you wish to continue?", _
                        vbQuestion + vbYesNo, "While Loop Test")

    Loop While MyResponse = vbYes

    Code continues here whether once the Loop has ended...

End Sub
```

The above will run and repeat a message box function **while** the user has chosen '*Yes*' to keep the loop going.



Or putting it another way, **until** the user chooses the '*No*' option.

```
Sub UntilConditionLoop()

    Code runs here first before it enters the Loop block...

    Do

       MyResponse = MsgBox("Do you wish to continue?", _
                        vbQuestion + vbYesNo, "Until Loop Test")

    Loop Until MyResponse = vbNo

    Code continues here whether once the Loop has ended...

End Sub
```



Both the above examples displays a message box and will respond in the same way.

Notice the other two variations being very similar other than where the condition is. It's tested first and not at the end of the loop block. In practical terms, a structure which tests a condition may never run the code in the block if the condition is not satisfied.

Therefore, the question is "*Do you want the procedure to run at least once or potentially not at all?*" This answer is where you place the condition at the beginning or the end of the loop structure.

The next example, will test a condition before running the code in the block based on if the **ActiveCell** is empty or not before moving down to the next row.

```
Sub PositionCursor()

    Code runs here first before it enters the Loop block...

    'Place the cursor in the next available blank (cell) row
    Do Until ActiveCell.Value = Empty

       ActiveCell.Offset(1, 0).Select 'move down one row
```

```
    Loop

    Code continues here whether once the Loop has ended...

End Sub
```

If the **ActiveCell** was blank before entering the loop, why would you need to move the cursor down a row? Which is why it is tested first and not at the end of the loop block.

Exit Do

Exit Do keywords are included in a block should you wish to terminate a loop block early without having wait to the end of iteration period.

It can speed up your procedure if there are various tests in a loop that may unexpectedly change state and act as a catch (error handler of some kind).

It is commonly found with `If` blocks nested in a loop of this kind.

Do...Loop - *no condition!!*

Make sure you have a condition set in any loop block otherwise it will loop infinitely until it runs out of memory or an object fails.

Do not write this:

```
Sub LoopForever()

    Code runs here first before it enters the Loop block...

    Do

        'Code here...

    Loop

    'Code will not reach this point - it will have failed!

End Sub
```

Where's the condition in the above block?

Next Topic: Looping (For...Counter/Each...Loop)

Looping (For...Counter/Each...Loop)

This is one of the '*control flows*' VBA provides which is required to make your VBA code procedures more flexible, reduce the amount of code required and make the system think for itself!

Without any of these '*control flows*', your code is very linear and rigid which never is really suitable when trying to mimic '*real world*' processes.

The four '*control flows*' are:

1. If...Then...Else...End If
2. Select Case...End Select
3. Do...Until/While...Loop
4. For...Counter/Each...Next

There two variations of this type of loop both which are *controlled* counting loops which is neither driven by the user or the system.

There are:

1. For...Counter...Loop
2. For...Each...Loop

Before any of the above loops are called, we know how many times the system will repeat the code inside the block as opposed to a *conditional* type loop (Do...Until/While...Loop).

For...Counter...Loop

This type of loop is defined by the user telling the system how many times to repeat or the system using an object method or function to identify the number of iterations.

Structure (syntax):

```
For Counter = Start To End [Step N]
    Code executed here...
    ....
    [Exit For]
Next [Counter]
```

The ***Counter*** is a variable keeping count of the current number using the **Start** and **End** as its range. As soon as the ***Counter*** = **End** then the loop is finished and the code jumps out of the block.

Example:

	A	B	C	D
1	Sales figures for 2010			
2	Month 1			
3	Month 2			
4	Month 3			
5	Month 4			
6	Month 5			
7	Month 6			
8	Month 7			
9	Month 8			
10	Month 9			
11	Month 10			
12	Month 11			
13	Month 12			
14				

The following example will display an input box requesting the sales figure for Month 1, Month 2 etc and input the results into the relevant cell on the spreadsheet. The procedure will therefore loop 12

Tip: Use an **Exit For** to terminate a block early and speed up your procedures.

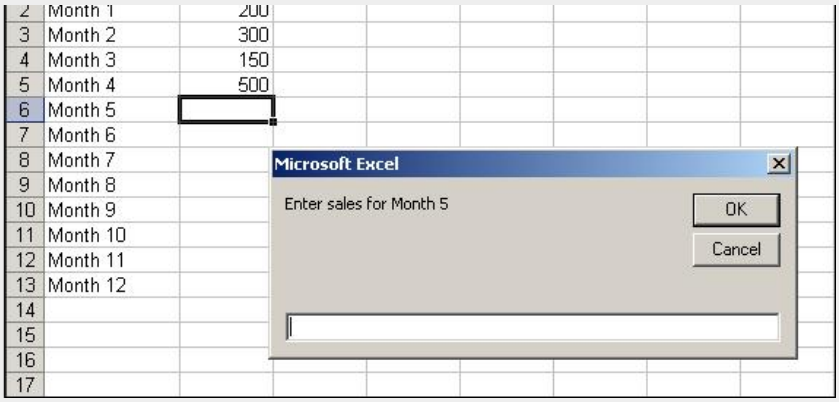
To break a loop during running your code use **CTRL + PAUSE/BREAK**.

Save your work before running a looping piece of code!

VBA Keywords: ActiveCell, Offset, MsgBox, ActiveWorkbook, Worksheets, WorkBooks, InputBox, Close and Exit For.

times.

```
Sub MonthsForLoop()
    Dim MonthlySales As String
    Dim num As Integer

    For num = 1 To 12
        MonthlySales = InputBox("Enter sales for Month " & num)
        ActiveCell.Value = MonthlySales
        ActiveCell.Offset(1, 0).Select
    Next num
End Sub
```



The InputBox Function is not a practical solution for the above example but it displays how the For...Loop works.

The ***Counter*** variable at the end of the loop (after the **Next** keyword) is optional and can be left out but personally it makes it very clear to what is incremented (in the example by 1).

The **Step** argument is optional too and by default means the variable (***Counter***) will increment by 1. If your want to change the increment or use the decrement action (downward count) then you need to add the **Step** keyword with the value you wish to increment or decrement.

For example:

```
'Positive increment of 10
Sub ForLoopStepPositive()
    Dim counter As Integer
    For counter = 10 To 100 Step 10
        MsgBox counter
    Next counter
End Sub
```

```
'Negative decrement of 10
Sub ForLoopStepPositive()
    Dim counter As Integer
    For counter = 100 To 10 Step -10
        MsgBox counter
    Next counter
End Sub
```

You will also need to make sure the range **Start To End** is synchronised with the direction of the **Step** value otherwise it will cause an error.

For...Each...Loop

This type of loop is a *self-counting* loop based on a array variable (which use an index), Collections (which is Excel array to their objects) or by an object member method (like the Count method).

This is commonly used with Collections and therefore the number of times a loop occurs is driven by the current collection array.

Structure (syntax):

```
For Each Element In Group

    Code executed here...
    ....
    [Exit For]

Next [Element]
```

A typical example could be to loop through the current number of Worksheets in the ActiveWorkbook:

```
'Loops through each worksheet in the ActiveWorkbook
Sub HowManySheets()
    Dim item As Worksheet
```

```
    For Each item In ActiveWorkbook.Worksheets
        MsgBox item.Name
    Next item
End Sub
```

If there are 5 worksheets in the ActiveWorkbook, the procedure would loop 5 times.

It uses the **`ActiveWorkbook.Worksheets`** Collection to determine the number if elements (note the word is plural) in the *group*. The Item variable is its *element* which needs to be the singular matching object which is in this case **`Worksheet`** (singular keyword).

Another example could be to close all workbooks in Excel:

```
'Loops through each workbook in open in Excel and closes it except the
'current workbook (which contains this piece of code!
Sub closebooks()
    Dim wb As Workbook
    For Each wb In Application.Workbooks
        If wb.Name <> ThisWorkbook.Name Then
            wb.Close
        End If
    Next wb
End Sub
```

The If statement is used to test if the workbook collections current element is the current workbook (containing the above code) as we do not want to close it.

Exit For

`Exit For` keywords are included in a block should you wish to terminate a loop block early without having wait to the end of iteration period.

It can speed up your procedure if there are various tests in a loop that may unexpectedly change state and act as a catch (error handler of some kind).

It is commonly found with `If` blocks nested in a loop of this kind.

Next Topic: With...End With blocks

With...End With Blocks

VBA Keywords: Selection, Name, Font, Bold, Italic, ColorIndex and Size..

The **With…End With** block instruction enables you to perform multiple operations on a single object. This is another way to make the code execute more quickly and code styles more efficient.

The following procedure will format the selected cells with the font '*Times New Roman*', font size 12, Bold, Italic and the colour Blue.

```
Sub ChangeFont()
    Selection.Font.Name = "Times New Roman"
    Selection.Font.Size = 12
    Selection.Font.Bold = True
    Selection.Font.Italic = True
    Selection.Font.ColorIndex = 5
End Sub
```

The above procedure can be rewritten using a **With…End With** block as follows:

```
Sub ChangeFont()
    With Selection.Font
        .Name = "Times New Roman"
        .Size = 12
        .Bold = True
        .Italic = True
        .ColorIndex = 5
    End With
End Sub
```

Using the this type of block, your code is cleaner and easier to maintain. The With...End With block encapsulates the object and member without the need to repeat unnecessary (duplicate) code.

If fact, when you record a macro and you navigate through the dialog boxes making various changes before choosing the OK button you in fact capture the code using the above structure. Try the **Font** Dialog box whilst recording the macro.

Next Topic: User Defined Functions (UDF's)

User Defined Functions (UDF's)

A user defined function can be used when the built in Excel VBA functions do not meet the user's requirements. The user defined function can then be used in formulas in the same way as a built in Excel function is utilised. User defined functions are limited to doing just calculations that result in a single return of a value.

The syntax of a user-defined function is as follows:

```
Function NameofFunction([Optional] Argument1 [As Type], _
                        [Optional] Argument2 [As Type], _
                        ... [Optional] ArgumentN [As Type] ) [As Type]
    Statements here ...
    ....
    NameofFunction = Value being returned

End Function
```

NameofFunction	The name of the function.
Arguments	The arguments of the function. If an argument is to be optional, enter the word `Optional` before the name of the argument. The `As Type` option allows you to specify the data type for the return value.
Statements	The various lines of code.
NameofFunction=Value	Name is the name used in the first line of the function. Expression is the return value of the function.

Note: The square brackets wrapped around a keyword in the above syntax denotes as optional and can be left out altogether.

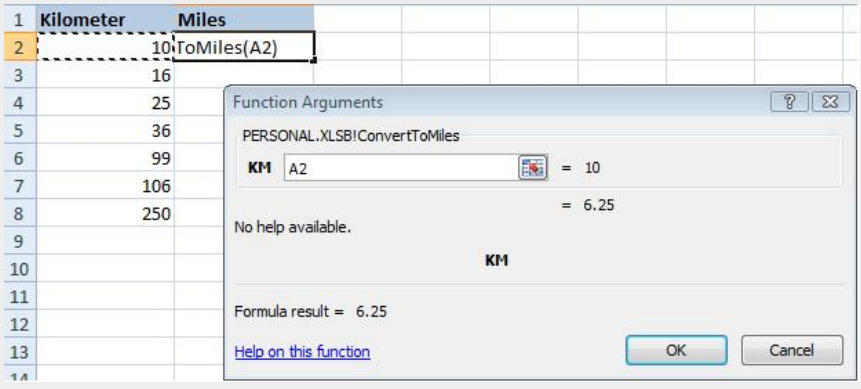
Creating a User Defined Function

The following is a simple function example to convert *Kilometers* recorded into *Miles*.

1. From the **Developer** tab on the Ribbon Bar, click the **Visual Basic** icon.
2. Click on the **Insert** menu and select **Module**.
3. Enter the following code:

```
Function ConvertToMiles(KM)
    ConvertToMiles=KM / 1.6
End Function
```

4. Back in the Excel spreadsheet, click on the **Formula** tab on the Ribbon Bar and click the **Insert Function** icon.
5. From the list of **Categories**, select **User Defined**.
6. Select **ConvertToMiles** and click on **OK**.
7. Enter the cell reference of the Kilometer value you wish to convert into miles, into the **KM** field and click on **OK**.



VBA Keyword: Function...End Function, Application, Round, IsNumeric, If...Then.End If.

Tip: When creating user defined functions, you may want to wrap them into a separate Excel file and create an Add-In so they can be easily distributed to other users.

Using built-in functions

It is possible to use built in Excel functions within a user defined function.

The syntax used for built in Excel Functions is as follows:

`Application.NameofFunction(Arguments Required)`

An example which incorporates the Excel **Round** function to the above user defined function (*ConvertToMiles*).

```
Function  ConverToMiles(KM)
    ConvertToMiles = Applications.Round(KM / 1.6, 0)
End Function
```

The above amended code rounds the resulting returning value to zero decimal places using the standard Excel built-in **Round** function.

Using The Optional Argument

The **`Optional`** keyword preceding the argument name flags the argument as an optional parameter to the function call.

A lot of built-in Excel functions have optional arguments which always follow on from the *mandatory* arguments listed in a function and can therefore be omitted defaulting to a value the function procedure knows how to handle if left out.

This makes function more flexible and can give different returning values (answers) and/or change the behaviour of how the function will run. Think of the **VLOOKUP** function in Excel, see it's syntax below:

= VLookUp (Value , Range , Offset Column [, Type])

The last argument (wrapped in square brackets) is optional and always appears after all mandatory arguments (3 in this example) which can be omitted and still work. The optional argument is a value of either **`True`** or **`False`** which defaults to **`True`** if omitted and simply changes the way how this function will calculate.

An example - following on from the above code snippet above, I want a second argument (as optional) which allows the user to choose a positive whole number (**`Byte`** data type) as its value to represent the number of decimal places to pass into the calculation. If omitted, it defaults to 0 decimal places round to the nearest whole number:

```
Function ConvertToMiles(KM, Optional DecPlaces As Byte)
    If DecPlaces < 0 Then
        ConvertToMiles = KM / 1.6
    Else
        ConvertToMiles = Application.Round(KM / 1.6, DecPlaces)
    End If
End Function
```

	A	B	C	D	E	F
1	**Kilometer**	**Miles**		*Formula in use*		
2	125.5	78		=ConvertToMiles(A2)		
3	125.5	78		=ConvertToMiles(A3,0)		
4	125.5	78.44		=ConvertToMiles(A4,2)		
5						
6						

The user can now either omit the second argument (cell B2), add a value of 0 to represent no decimal places (cell B3) or add a positive number to pass into the Excel **Round** function (cell B4).

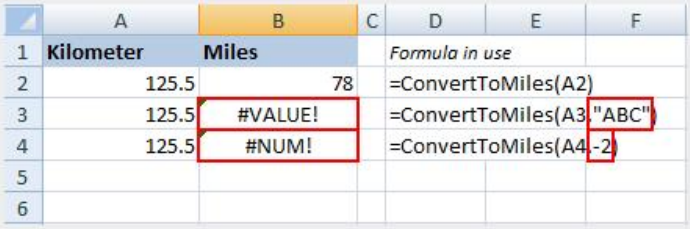
Using the As Type option

Optionally, the **`As Type`** keywords can be included to define a certain data type the argument and/or the function is controlled.

If omitted it will default to **`Variant`** (any data type it inherits) and can be open to abuse and more importantly errors.

You define a data type (see Variables & constants for more information) for each argument in the function and for the function's returning value too. If left out, you will need to add more code to handle different data input scenarios.

Let's take a look at what happens if the last above example function is abused.



	A	B	C	D	E	F
1	**Kilometer**	**Miles**		*Formula in use*		
2	125.5	78		=ConvertToMiles(A2)		
3	125.5	#VALUE!		=ConvertToMiles(A3,"ABC")		
4	125.5	#NUM!		=ConvertToMiles(A4,-2)		
5						
6						

In cell B3, setting the optional second argument to a **`String`** value "ABC" causes the **#VALUE!** error (a non numeric data input).

In cell B4, setting the optional second argument to a negative number causes another error **#NUM!** even though it's a number but the argument data type **`Byte`** only accepts positive numbers between 0 and 255 as its range.

The whole function is also expected to return a number which can be a larger than 255 and we therefore could apply the **`Integer`** as it's returning data type.

```
Function ConvertToMiles(KM, Optional DecPlaces As Byte) As Integer
    If DecPlaces < 0 Then
        ConvertToMiles = KM / 1.6
    Else
        ConvertToMiles = Application.Round(KM / 1.6, DecPlaces)
    End If
End Function
```

Notice I have left out the argument ***KM*** data type which defaults to **`Variant`**. Personally, I prefer to test for a data type in the code itself when the user or system passes a value to calculate.

An example:

```
Function ConvertToMiles(KM, Optional DecPlaces As Byte) As Integer
    If IsNumeric(KM) Then
        If DecPlaces < 0 Then
            ConvertToMiles = KM / 1.6
        Else
            ConvertToMiles = Application.Round(KM / 1.6, DecPlaces)
        End If
    Else
        ConvertToMiles = 0 'If it fails return a 0
    End If
End Function
```

I have tested to see if ***KM*** argument is a number by using the **`IsNumeric`** VB function.

All user defined functions can be called in Excel (as explained above) or into a calling Sub procedure like a VB or Excel function.

Next Topic: Event handling

Want to teach yourself Access? Free online guide at About Access Databases

Event Handling

VBA Keyword: InuptBox, MsgBox, Range, WeekdayName, Weekday, Now, Date, ActiveWindow, WindowState, EntireColumn, AutoFit, Font & Bold.

An Event is something that happens in a program such as:

- Opening or closing a workbook
- Saving a workbook
- Activating or deactivating a worksheet or window
- Pressing a key or key combinations
- Entering/Editing data in the worksheet
- Clicking the mouse on a control/object
- Double clicking on a cell
- Data in a chart is updated
- Recalculating the worksheet
- A particular time of day occurs

You can therefore run a procedure automatically when a certain Event in Excel occurs.

There are different objects (and therefore different levels) when Excel automatically triggers a procedure as the system is constantly *listening* for the event to occur.

Workbook Events

Open Event

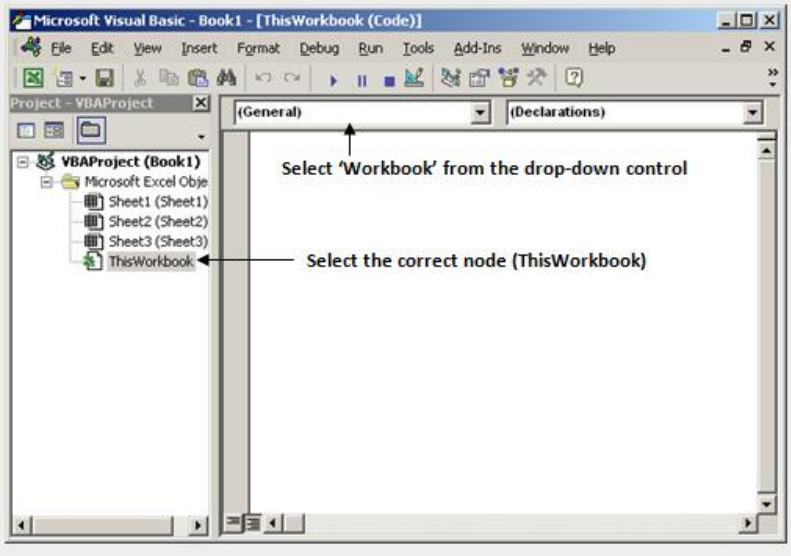
The most common type of Open Event is **`Workbook_Open`**. This procedure is executed when the workbook is opened and is often used for the tasks such as:

- Displaying a welcome message
- Opening other workbooks
- Setting up custom menus and toolbars
- Activating a particular sheet or cell

Example:

Every time the user opens the workbook, they are greeted with a message box displaying the day of the week. If it is a Friday, a message box will remind the user to submit their timesheet.

1. Open the required workbook.
2. Switch to the Visual Basic Editor.
3. Double click on **ThisWorkbook** from within the Project Explorer.



4. Click on the **Object** drop down list and select **Workbook**
5. Enter the following between the signature **`Private Sub Workbook_Open()`** and **`End Sub`** keywords:

```
Private Sub Workbook_Open()
    MsgBox "Today is " & WeekdayName(Weekday(Now), False, vbSunday)
    If Weekday(Now) = vbFriday Then
        MsgBox "Don't forget to submit your timesheet"
    End If
End Sub
```

Note: **`Private`** means that the procedure won't appear in the Run Procedure dialog box (i.e. Macros dialog). See Scope & Visibility in Variables & Constants for information.

Workbook Activate Event

The procedure is executed whenever the workbook is activated (gets the focus).

Example:

Call the signature **`Private Sub Workbook_Activate()`** using the same methods as previously explained above.

Enter the following code:

```
Private Sub Workbook_Activate()
    ActiveWindow.WindowState = xlMaximized
End Sub
```

Now the window will always maximise when the workbook gets the focus.

Note: Deleting an event (the signature) will not harm the system as it is re-generated each time you call one of the pre-defined signatures.

Example:

Using the **`Private Sub Workbook_SheetActivate(ByVal Sh As Object)`** signature is triggered across any worksheet in the active workbook.

Enter the following code:

```
Private Sub Workbook_SheetActivate(ByVal Sh As Object)
    Range("A1").Value = Date 'Enters the current date in A1
    Range("A2").Select 'Position the cursor in A2
End Sub
```

The **`'Sh'`** argument can also be used to refer to which worksheet is being called should you wish to control the index or name of a particular worksheet or group of worksheets.

By including a code line: **`If Sh.Name = "Sheet3" Then...`** it will handle the logic and control flow for 'Sheet3'.

Worksheet Events

Worksheet Activate Event

Within a workbook you also have separate nodes for each added worksheet chart sheet which contain a private (local) module over an above standard modules in a VBA project.

Example:

Every time the user clicks on 'Sheet1' if the first cell (A1) is empty then prompt the user with an InputBox function to enter a title.

```
Private Sub Worksheet_Activate()
    If Trim(Range("A1").value) = Empty Then
        Range("A1").Value = Trim(InputBox("Enter title:"))
        Range("A1").EntireColumn.AutoFit
    End If
End Sub
```

Note: If there are events at both the worksheet and workbook level which point to the same object (worksheet), then it's the worksheet level will run first followed by the workbook event.

Other Events

There are other ways to get Excel to trigger a macro using other events from other objects or controls.

It is possible to attach procedures to the ActiveX Controls so that whenever the user clicks onto a control, the procedure will run.

Example:

When the user clicks on the **Command Button**, a message box will appear.

1. From Excel, click on the **Developer** tab (Ribbon Bar), select **Insert** icon and choose **Button** icon from the Form Control section.

2. Draw the Command Button onto the spreadsheet.
3. The **Assign Macro** dialog box appears, Click the **New...** button.
4. Enter the following code:

```
Sub Button37_Click()
    MsgBox "Button click event!"
End Sub
```

Any control drawn on a worksheet or user form will have pre-defined events that can be coded to respond by the system.

How do you think features like *conditional formatting* and *data validation* work in a worksheet when set in Excel? When the user enters a value in a cell, the **Change** event is triggered:

```
Private Sub Worksheet_Change(ByVal Target As Range)
    If Target = Range("A2") Then Range("A2").Font.Bold = True
End Sub
```

Target is the argument to test which cell address is being changed.

Next Topic: Error Handling

Error Handling

No matter how thorough you are when writing code, errors can and will happen.

There are steps that developers can take to help reduce unwanted errors and this is considered just as important as the actual process of the procedure.

Before understanding and applying error-handling routines, planning to avoid errors should be undertaken.

- Design the procedure's process electronically or on paper – flow chart and paper test.
- Creating smaller portions of code – snippets to be called and re-used
- Using the Option Explicit statement – declaring your variables officially.
- Syntax checking – user defined commands and functions.
- Comments – remarking your code at various points.
- Testing application – functional and usability.

Note: Some of the above points are methodologies which are outside the scope of this reference guide.

There are three different types of errors:

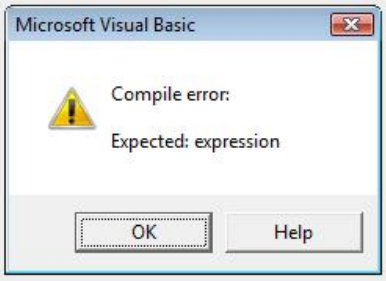
1. Design Time Errors
2. Run Time Errors
3. Logical Errors

The order of the above progressively is harder to find leaving the last item the most challenging!

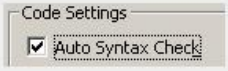
Design Time Errors

The simplest form of error and often caused by typing (typo's) mistakes.

When typing a known keyword or statement, VBA will turn the text to red (*default colour*) and if the option is enabled, provide a prompt:



To switch off the above prompt, go to **Tools** select **Options…** and deselect **Auto Syntax Check** option.



The routine will instantly cause a run time error if not corrected at the design time and must but resolved before macros can run.

Run Time Errors

When executing code, no matter how thorough the debugging process has been, code may encounter errors while running.

There is only one way of doing this - **On Error GoTo** instruction. It is not a very sophisticated function, but it allows the flow of the code to continue and also where applicable, prevent infinite loops (*when the computer keeps on calculating never coming to an end*).

Three variations are available:

1. **On Error GoTo** *LabelName*
2. **On Error Resume Next**
3. **On Error GoTo 0**

Tip: Save you work before running error examples that contain loops (which try again).

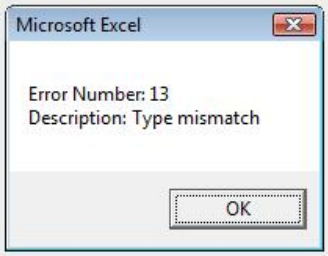
VBA Keywords: On Error GoTo, MsgBox, InputBox, CInt, Dim, Resume, Resume Next, Round, If...Then...Else.

On Error GoTo *LabelName* branches to the portion of the code with the label *LabelName* (*'LabelName' must be a text string and not a value*).

These commands are usually placed at the beginning of the procedure and when the error occurs, the macro will branch to another part of the procedure and continue executing code or end, depending on the instruction given.

```
'Simple Error handler with Err Object
Sub ErrorTestOne()
    On Error GoTo myHandler

    Dim intDay As Integer
    intDay = "Monday"
    MsgBox intDay
    Exit Sub

myHandler:
    MsgBox "Error Number: " & Err.Number & vbNewLine _
                            & "Description: " & Err.Description
End Sub
```

The above procedure will cause an error when executed and users will see:



myHandler is a user defined label (*must not use known keywords*) which listens for any errors that may occur. When an error is detected, the procedure jumps to a bookmark of the same label with a colon (**:**) (**myHandler:**) and executes from that point forward.

Using the **Err** object, developers can return two common properties '**Number**' and '**Description**'. The example message box concatenates these two properties into a user-friendly message (*see above*).

It is important to include the **Exit Sub** statement prior to the bookmark label otherwise the procedure will execute to the very end of the sub routine and should only be executed when an error has genuinely occurred.

The error above was due to a type mismatch. In other words I declared a variable **intDay** as an **integer** and assigned a string value to it.

Another example:

```
'Error to handle incorrect InpuBox value.
Sub ErrorTestTwo()
    On Error GoTo myHandler

    Dim intInput As Integer
    Dim strResponse As String
    Dim blnErr As Boolean
    intInput = CInt(InputBox("Enter your age:"))
    blnErr = False
    If Not blnErr Then
        If intInput > 64 Then
            strResponse = "You are at the retirement age!"
        Else
            strResponse = "You have " & (65 - intInput) & _
                " year(s) remaining until retirement."
        End If
    Else
        strResponse = "Unknown error entered!"
    End If
    MsgBox strResponse
    Exit Sub

myHandler:
    intInput = 0
    blnErr = True
    Resume Next
End Sub
```

The above example illustrates how to gracefully handle incorrect (*type mismatched)* values and then resume the next line of execution using `Resume Next` statement.

The variable `blnErr` is flagged as true if an error occurs which is then tested with an **If** statement.

If the `Resume Next` is replaced with just the `Resume` statement, you will find the input box will loop itself until the correct data is entered. Be careful before testing this out due to infinite loops that may occur (*if you edit the wrong part of the procedure*).

The statement **On Error GoTo 0** (*zero*) simply disables the error command during the procedure.

Should users wish to switch off this feature? To switch it back on, just introduce a new statement line of either:

1. **On Error Goto** *myLabel*

2. **On Error Resume**
3. **On Error Resume Next**

Any code can be written to handle errors gracefully which can include **`If`** and **`Case`** statements. It is common to have a **`Case`** statement to test which error was fired and deal with it in a separate calling procedure (*branch out another procedure*).

Logical Errors

This type of error is the most difficult to trace as its syntax is correct and it runs without any run time errors.

A **logical error** is one that does not give users any indication that an error has occurred due to the fact that a logical error is the *process of logic* and not the code itself.

Performing a calculation in a spreadsheet using a function will return an answer but is it the correct answer?

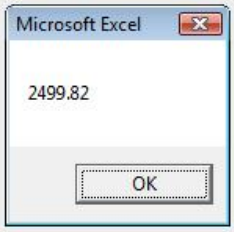
Example:

```
'Logical Error Test Example
Sub LogicalErrorTest()
    Dim lngQty As Long
    Dim dblNet As Double
    Dim sngDiscount As Single

    lngQty = 10
    dblUPrice = 250
    sngDiscount = 0.15

    'Calculate gross (inc VAT @ 17.5%)
    'Logically INCORRECT!
    MsgBox Round(lngQty * dblUPrice * 1 - sngDiscount * 1.175, 2)

    'Logically CORRECT!
    MsgBox Round(((lngQty * dblUPrice) * (1 - sngDiscount)))
End Sub
```

The above procedure showed a quantity (**`intQty`**) of goods, with a unit price (**`dblUPrice`**), a discount (**`sngDiscount`**) at a fixed vat rate of 17.5%.

To calculate the correct gross value, there is an order of which operands are calculated (see Formulas) first and without the care of using brackets, the system follows the rules of mathematics and looks at the operator's precedence automatically.

The first message box shows:



WRONG!

Followed by the second message box:

CORRECT!

Both calculations worked but the first was illogical to the objective of the process (workflow).

How we find such errors? ***Debugging tools!***

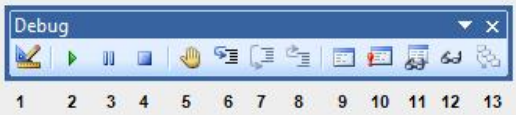
Next Topic: Debugging

Want to teach yourself Access? Free online guide at About Access Databases

Debugging

Debugging is the process of stepping through the code line by line and checking the reaction of each line to help trace errors that may be difficult to find at run time especially logical errors.

The **Debug** toolbar allows users to step in, out, over or watch certain variables change state in a controlled manner and can be switched on or off in the Visual Basic Editor window.



1	**Design Mode**.
2	**Run Sub/User Form** starts the macros where the insertion point is or displays a Macro Dialog Box.
3	**Break** pauses the macro while it's running and switches to break mode.
4	**Reset** current macro clearing all breaks, step into/over procedures and variables.
5	**Toggle Breakpoint** allows marking a line of code at which point a macro will stop.
6	**Step Into** a macro one line at a time.
7	**Step Over** a macro one line at a time ignoring any other sub routines.
8	**Step Out** over a macro and continue running the rest of that macro.
9	**Locals Window** is displayed showing all variables and expressions with values for the procedure currently running.
10	**Immediate Window** is displayed allowing pasting of code to the window and testing the code by using the ENTER key (*cannot save contents*).
11	**Watch Window** is displayed allowing drag 'n' drop of expressions into it to monitor their values.
12	**Quick Watch** displays a Dialog Box showing the current line of codes value.
13	**Call Stack** displays a Dialog Box listing all active calls statement to the current procedure. This option is used when using a step procedure.

The most effective way to debug a procedure is to learn some keystrokes and mark breakpoints in the code.

To add breakpoints, place the mouse pointer to the left grey margin at the point where you wish to pause the procedure and click once with the left mouse button, click button 5 (*as above*) or press **F9** function key (toggles on/off).

```
Sub CalcPay()
    On Error GoTo HandleError
    Dim hours
    Dim hourlyPay
    Dim payPerWeek
    hours = InputBox("Please enter number of hours worked", "Hours Worked")
    hourlyPay = InputBox("Please enter hourly pay", "Pay Rate")
    payPerWeek = CCur(hours * hourlyPay)
    MsgBox "Pay is:    " & Format(payPerWeek, "£##,##0.00"), , "Total Pay"
HandleError: 'any error - gracefully end
End Sub
```

When you run the procedure or press the **F5** key, the procedure will pause at the first highlighted break:

```
Sub CalcPay()
    On Error GoTo HandleError
    Dim hours
    Dim hourlyPay
    Dim payPerWeek
    hours = InputBox("Please enter number of hours worked", "Hours Worked")
    hourlyPay = InputBox("Please enter hourly pay", "Pay Rate")
    payPerWeek = CCur(hours * hourlyPay)
    MsgBox "Pay is: hours = Empty mat(payPerWeek, "£##,##0.00"), , "Total Pay"
HandleError: 'any error - gracefully end
End Sub
```

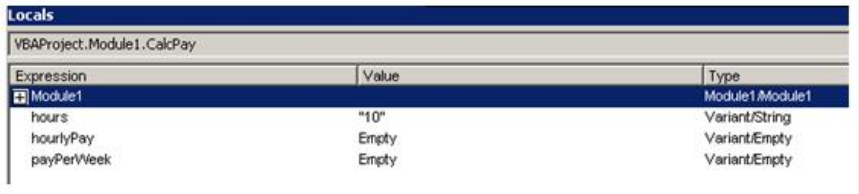
At this point, users can either continue to run the remaining procedure (*press F5 key*) or step through line by line by pressing the **F8** key.

Tip: Keyboard shortcuts are quick and simple. Learn F5, F8 and F9.

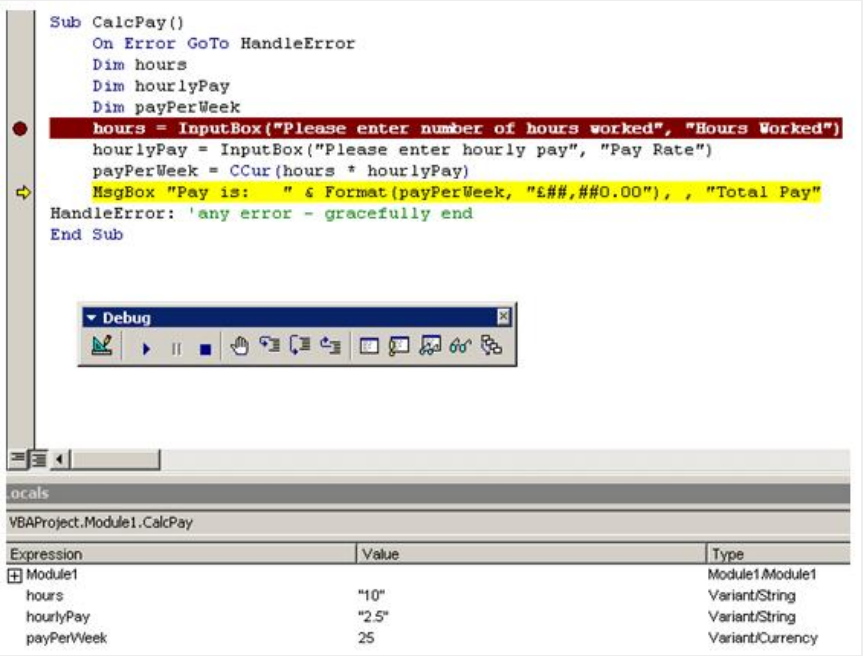
VBA Keywords: On Error GoTo, MsgBox, InputBox, Debug.Print & CCur.

By placing the mouse pointer over any variable or object property, the user will, after a few seconds, see the current value assigned.

Alternatively, by revealing the **Locals Window** (*button 9 above*), users can see all variables and property's values:



After a few steps (**F8** key):



Debugging between calling procedures can be controlled as the **F8** key steps in order line by line across more than one procedure.

To step out of a sub procedure and carry on with the main procedure, press the **SHIFT + F8** keys.

Debug.Print Command

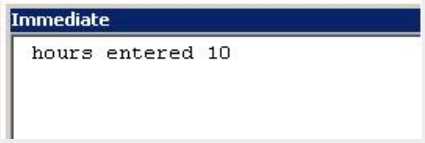
A return value will be printed to the **Immediate Window** (*button 10 above or CTRL + G*).

Two ways to print an output value in the immediate window:

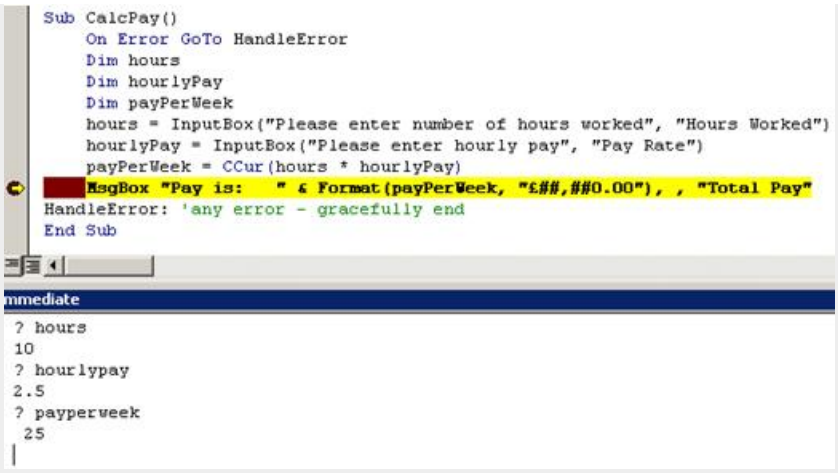
1. **Debug.Print** *Expression*
2. **?** *Expression* (*within the Immediate Window*)

```
Sub CalcPay()
    On Error GoTo HandleError

    Dim hours
    Dim hourlyPay
    Dim payPerWeek
    hours = InputBox("Please enter number of hours worked", _
                                                    "Hours Worked")

    Debug.Print "hours entered " & hours

    hourlyPay = InputBox("Please enter hourly pay", "Pay Rate")
    payPerWeek = CCur(hours * hourlyPay)
    MsgBox "Pay is: " & Format(payPerWeek, "£##,##0.00"), _
                                                    , "Total Pay"
HandleError: 'any error - gracefully end

End Sub
```

The above will print the 'hours' variable to the immediate window:



If you set a breakpoint and have the **Immediate Window** visible, you can use a different method to reveal the current values of any known variable or property:



Type a question mark (**?**) followed by a space and then the variable or property and press the enter key to reveal the output value.

Next Topic: Creating User Forms

Want to teach yourself Access? Free online guide at About Access Databases

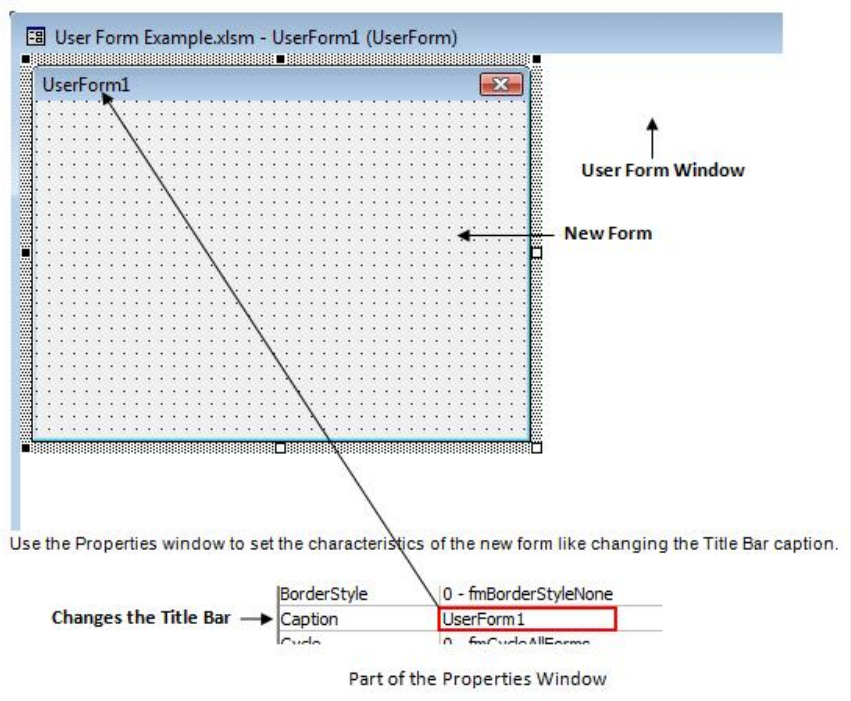
Creating User Forms

Pre-defined **Dialog Boxes** like InputBox and MsgBox functions are useful and quick to use. However, designing your own **Dialog Boxes** (or **User Form**), allows you to add other controls and personalise your application.

Creating a new User Form

Make sure you are in the Visual Basic Editor and not Excel.

Select **Insert**, **UserForm**:



Note: To load the Properties Window, press the **F4** function key.

In addition to the **Properties** window, when active on the new form, the **Toolbox** Toolbar automatically appears. (*If this is missing, use the View, Toolbox command to show it.*)

Also, you may want to display the 'UserForm' toolbar to align and rearrange the controls on the form. Select **View**, **Toolbars** and choose **UserForm**.

Userform Toolbar

This toolbar is only available for designing and arranging objects when creating or modifying forms (*user forms*).

1. **Bring to Front** moves the selected object to the front of all other objects.
2. **Send to Back** moves the selected object to the back of all other objects.
3. **Group** two or more selected objects together as one.
4. **Ungroup** where a single object was made up of two or more objects.
5. **Alignments** of selected objects to various alignments - *see below*.
6. **Vert/Horiz Alignments** of selected objects - *see below*.
7. **Sizes** a number of selected objects to the same dimensions - *see below*.

Tip: Press the **F5** function key to run and preview a form during the design time environment.

VBA Keywords: Show, If...Then...Else, MsgBox, RGB & Unload.

8 **Zoom** the User Form by magnifying/diminishing by percentage.

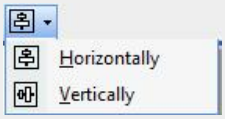
Alignments (Button 5)

Choose from one of the alignments as to how a number of selected objects will be placed together.

This keeps controls on a form or Dialog Box symmetrically aligned and therefore professional looking.

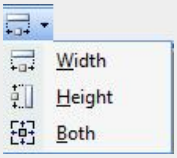
Objects can also be numerically set using the Properties Window.

Vertical/Horizontal Alignments (Button 6)



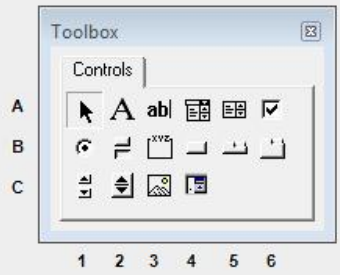
These are two repeated types of alignments as mentioned above allowing objects to be centred.

Sizes (Button 7)



Changes the size of selected objects to the same dimension as each other. This can also be set from the Properties Window.

Toolbox Toolbar



A1 **Select Objects**: When there is no control to draw, this mode allows you to select other controls.

A2 **Label**: Allows you to create text (*caption*) that a user does not change.

A3 **TextBox**: Allows you to create an edit box which a user types into.

A4 **ComboBox**: Allows the user to select from a drop down box items predefined.

A5 **ListBox**: As above but shows many item in one view with a vertical scroll bar.

A6 **CheckBox**: Allows the user to create a CheckBox where an item can only have a yes or no (*true or false*) answer.

B1 **OptionButton**: Allows you to display multiple options with a frame where only one can be selected at a time.

B2 **ToggleButton**: Like a CheckBox, but a button version.

B3 **Frame**: Allows you to create a frame to store controls in one group (*usually option buttons*).

B4 **CommandButton**: Creates a button like the OK, Cancel and Other... Buttons.

B5 **TabStrip**: Allows you to create multiple pages of the same Dialog Box controls.

B6 **MultiPage**: Allows you to create multiple pages of different controls (*multi- tab Dialog Boxes*).

C1 **ScrollBar**: Provides a graphical scroll bar to allow scrolling through a list of values.

C2 **SpinButton**: This button allows you to set values by scrolling up or down through ranges.

C3 **Image**: Allows the user to store graphics in a Dialog Box.

C4 **RefEdit**: Allows a range to be plotted into this control from a spreadsheet. (*Not available in Excel 97*).

Adding Controls to a Custom Dialog Box

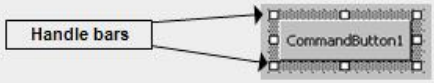
By using the **Toolbox**, standard controls (*command buttons, text boxes and others*) can be added to a

user form.

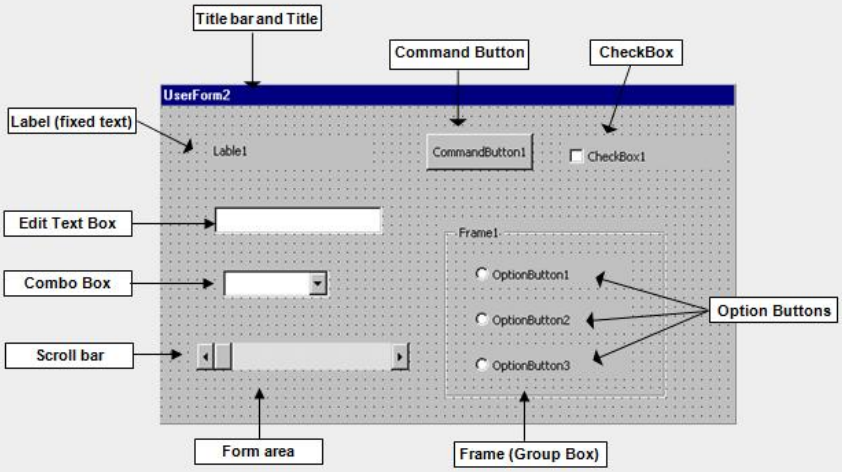
Make sure the form is the active window.

Click on the required control and then click on the user form roughly where the control is to be positioned or drag and drop the control from the **Toolbox** to the area on the form.

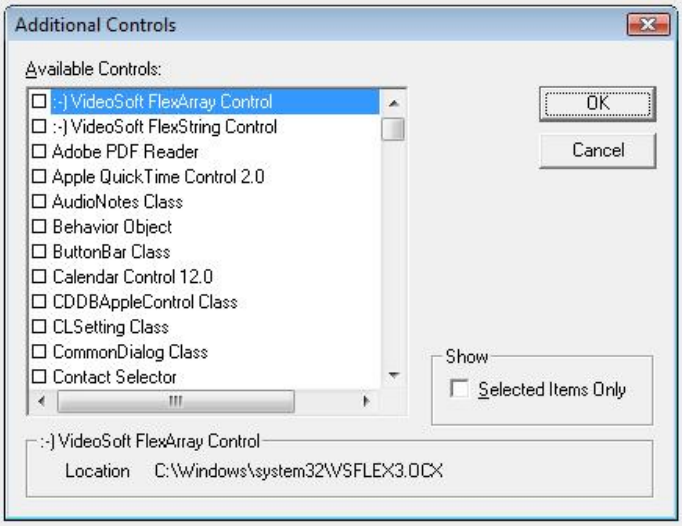
Handlebars appear around the selected control. This allows control(s) to be resized and positioned.



Common Controls



There are many more controls than the standard set which is installed with Microsoft Excel and can be added. Select **Tools**, **Additional Controls....**

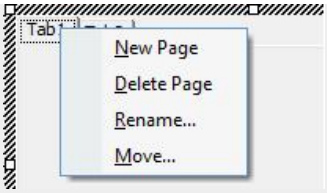


Tick the required item and choose the **OK** Button. This updates the **Toolbox** toolbar.

Creating Tabs for Userforms

If a single user form needs to handle a large number of controls that can be sorted into categories, use a **Tab** Control.

Tab pages can be added and deleted using the properties of the tab (itself) of the control itself by right-mouse clicking the item.



Setting Controls with Properties (*Design Time*)

Some controls are better set during the design side of a user form by using the **Properties** window.
Typical examples of setting these controls:

- Sorting out the Tab Order of controls.
- Setting the default values to edit boxes, checkboxes and many more.
- Creating control tool tips and captions.
- Setting the Accelerator key (*underscored letter*) of a control.
- Other colours and graphics that really do not need code handling.

Initialising Controls with Code (*Run Time*)

Some controls (*some included as above*) can be set as the userform is running or before the form is displayed by setting properties with code.
Typical examples of setting controls with code:

- Setting initial values of controls like an Edit Box or a Combo Box.
- Setting the focus of a control.
- Validating values in a Dialog Box.
- Changing values in a Dialog Box while it is running.
- Showing and hiding other controls.
- Enabling and disabling controls.

These controls are set to different events of a control. These include:

- Click (*and double click*) of a control - command buttons, checkboxes.
- Change of a control - edit boxes.
- Initialising of a control - as the form starts up.
- Exiting a control.

Example:

```
Sub MyDialogBox_Initialize()
    TextBox1.Text = "Sales"
    Checkbox1.Enabled = True
    Me.Command1.SetFocus
    OptionCommand1.Visible = False
    Checkbox2.Value = False
End Sub
```

Five controls are set when the form (*MyDialogBox*) is shown (*Initialised*).

1. **TextBox1** displays “Sales” in it.
2. **Chexkbox1** control is active.
3. **Command1** button has the focus.
4. **OptionCommand1** is not visible (*hidden*).
5. **Checkbox2** is not ticked (*False value*).

Displaying a Userform

Once the userform has been created, the next stage is to test to see how the user form will look. You can use the **Run** (**F5** function key in design time mode) command when the active form is displayed. But, writing code is ultimately how a user form will be used.

Decide where the code is to be stored (*in a Module, Worksheet or Workbook*).

Use the name of the form with the **Show** method command.

Example:

```
Sub DisplayMyUserform()
    MyUserForm.Show
End Sub
```

The user form is known, as ‘*MyUserForm’* and the **Show** method will display the user form.

There is one optional argument called **Modal** which can be explicitly defined and has a value of **0** or **1** .

1 = a modal state which means that users have to complete the form and can not click anywhere else (in the background).

0 = a modeless state which allows users to click outside the form area.

The default is **1** if omitted.

`MyUserForm.Show 0`

`MyUserForm.Show 1` *or* **`MyUserForm.Show`**

Adding Code to respond with User Forms

Each control will have its own set of events. These events store the code and are executed when that event is triggered.

For example, a **Button** recognises the *Click_Event*, a **Form** recognises an *Intitialize_Event* and a **Combo Box** recognises a *Change_Event*.

To assign code to a control, display the form and double click on that control. This opens the module and the main event allowing code to be written:

Example:

When the **OK** Button is clicked...

```
Private Sub cmdOK_Click()
    Range("NameResult").Select
    EnterText.Hide
    If txtName.Text = "" Then
        MsgBox "Must enter a name. Try again."
        EnterText.Show
    End If
    ActiveCell.Value = txtName
    Unload Me
End Sub
```

When the userform is displayed, if the **OK** Button is clicked, the above code is executed and checks to see if this Textbox (*txtName*) is empty or not using the **`If`** statement. If false, it displays a message prompt and shows the user form again. If true, it enters the data into a spreadsheet (*range - NameResult*).

`Unload Me` is the way to close a form (itself)

The Me Property

The **Me** property returns a reference of the form itself that the code is currently running. This is used as shorthand for the full reference of a form.

Example:

Suppose you have the following procedure in a module:

```
Sub ChangeFormColour(FormName As Form)
    FormName.BackColor = RGB(Rnd * 256, Rnd * 256, Rnd * 256)
End Sub
```

You can call this procedure and pass the current instance of the Form as an argument using the following statement:

```
Sub cmdColour_Click()
    ChangeFormColour Me
End Sub
```

The **ChangeFormColour** procedure is passed to the **Me** property in the current form running which therefore changes the colour of a specified control(s) to the colour defined.

`RGB(Rnd * 256, Rnd * 256, Rnd * 256)` is a Red, Green and Blue colour function.

To see an example, click on userform example

Next Topic: DAO/ADO Objects

DAO/ADO Objects

There are several ways to connect to a database in VBA whether it is a relational database (RDBMS) or a flat-file database (like Excel).

Also, where and what type of database application/server it is will pretty much determine which is considered best for the job.

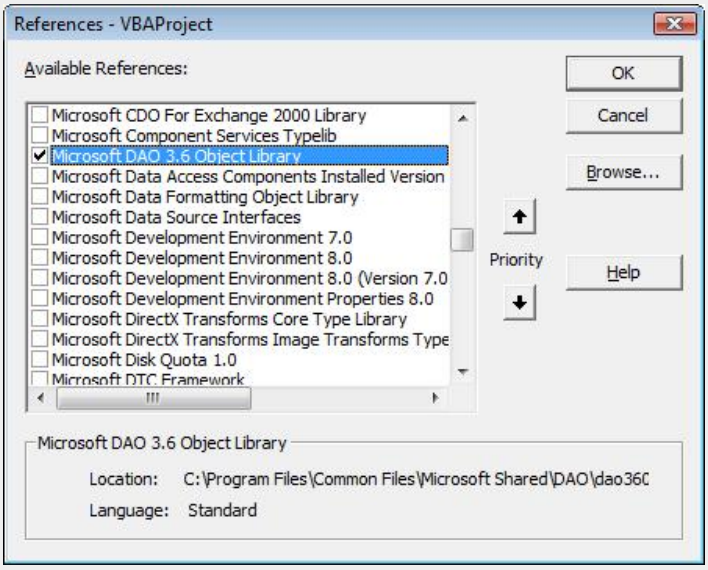
The two I'm going to mention in the article (**DAO** and **ADO**) is considered the more popular techniques deployed but for your reference you may want to investigate the older **RDO** (*Remote Data Object*) which has been really replaced with **DAO**, **OLE-DB** and **ODBC** to help establish which would be best for your solution.

DAO

DAO stands for **D***ata* **A***ccess* **O***bjects* and is one of the technologies to allow communications to external applications (*mainly databases*).

In order to use this feature, users will need to add the **DAO** library to the project.

Choose from the **Tools** menu and select **Reference...**



This library will than allow objects to be created to interrogate a database, tables, fields and return information to populate a spreadsheet. This will also allow users to add, edit, update and delete data to an external file without the need to open the associated application.

An advanced feature of this library will even allow users to create, modify and delete structures of a database whether a table, query, stored procedure or fields.

Using the control flow techniques as discussed in this manual, the user can fully control how data should be handled - opening the potential power of VBA.

Note: In order to test this section, users will need an Access database and will need to familiarise themselves with the database. It is not essential to have Microsoft Access loaded as this reference uses the *backdoor* but it will be difficult to check the database without it!

Example - Connecting to a database:

```
Sub ConnectDB()
    Dim db As Database
    Dim rst As Recordset

    Set db = OpenDatabase("C:\db1.mdb")
    Set rst = db.OpenRecordset("Customers")

    'displays the first record and first field
    MsgBox rst.Fields(0)

    'close the objects
    rst.Close
    db.Close

    'destroy the variables
```

Tip: Decide on which library to use and stock with it. Don't mix the two together (ADO and DAO) though it can still work but the order of referencing will matter.

VBA Keywords: DAO, ADODB, Connection, Recordset, Open, Update, EditMode, AddNew, Fields, Set, New & With...End With

```
    Set rst = Nothing
    Set db = Nothing
End Sub
```

The above example opens an Access database (db1.mdb) in memory and sets a reference to one of its known tables using the '*OpenRecordset*' method. It then displays the first row and first field of the table:

Same information from the 'Customer' table in Access.

Customers : Table

Customer ID	Company Name
ALWAO	Always Open Quick Mart
ANDRC	Andre's Continental Food
ANTHB	Anthony's Beer and Ale

The property **`Fields`** of the **`RecordSet`** object is a collection (*or array*) that is held in memory and by changing the element number, users can return a different field (column of the table).

`rst.Fields(1)`

The above illustration would show the customer's name instead of the ID number.

A good discipline is to close and set an object to **`Nothing`** that releases memory, hence the last four lines of code.

Example - Working with records:

```
'Opens a connection to the Customers table
'and populates a blank worksheet.
Sub PopulateCustomers()
     Dim db As Database
     Dim rst As Recordset
     Dim i As Long

     Set db = OpenDatabase("C:\db1.mdb")
     Set rst = db.OpenRecordset("Customers")

     'look through each record and populate
     'ID, Name and Country into a worksheet.
     Do Until rst.EOF
        ActiveCell.Offset(i, 0).Value = rst.Fields(0)
        ActiveCell.Offset(i, 1).Value = rst.Fields(1)
        ActiveCell.Offset(i, 2).Value = rst.Fields(8)
        i = i + 1
        rst.MoveNext
     Loop

     'close the objects
     rst.Close
     db.Close

     'destroy the variables
     Set rst = Nothing
     Set db = Nothing
End Sub
```

The above example once again opens the table 'Customers'. Using a conditional loop at which point the property **`EOF`** (*End Of File*) returns **`True`** or **`False`** every time the record changes using the **`MoveNext`** method, three columns in the worksheet from the starting active cell are populated by three different field indexes.

Even though the above example used **`rst.Fields(8)`** to determine the ninth column, it may be fair to say that users may not know the position number of the field but instead know its fieldname. In this case, users can refer to the name of the field as a string argument.

`rst.Fields("Post Code").`

Note: Be careful to include a command to increment the collection (**`MoveNext`** method) otherwise this would cause the procedure to loop infinitely or run out of worksheet rows firing an error. **Save your work first before testing the above.**

When interrogating a table in a database, it may be required to test to see if the table actually has records in it before iterating through each record.

Wrap an **`If`** statement around the loop to test this out:

```
If Not rst.EOF And Not rst.BOF Then
      [code here]...
End If
```

If this returns **True** then at least one record is present. If both **EOF** and **BOF** are **True**, it means the cursor is positioned at the beginning and at the end of the record set (*which means it's empty*).

The **Not** keyword inverses the returning value which means that in the above example, both must be **False** if this is to run any code in between the statement.

Example - Editing records in a database:

Not only can users populate data from an external database, but also it is possible to change data in an external database.

```
'Opens a connection to the table Customers
'and adds a new record and then updates and closes
Sub AddNewRecord()
    Dim db As Database
    Dim rst As Recordset

    Set db = OpenDatabase("C:\db1.mdb")
    Set rst = db.OpenRecordset("Customers")

    rst.AddNew
    rst.Fields("Customer ID") = "XYZ"
    rst.Fields("Company Name") = "XYZ Foods Ltd"
    rst.Fields("Post Code") = "NW1 8PY"
    rst.Update

    'close the objects
    rst.Close
    db.Close

    'destroy the variables
    Set rst = Nothing
    Set db = Nothing
End Sub
```

The above example once again opens a connection to the '*Customer*' table and then uses two methods to add and update the new record.

The **Add** method triggers the mode to add the record but does not save it to the table until you call the **Update** method.

Note: Be careful to consider the table's structure and database rules that are often implemented such as primary keys and foreign indexes. The above example would fail if the customer id field was a unique primary key and the table already had such a reference.

Further coding would be required to test to see if the record number existed, before adding and updating the record.

To edit a record, users must first locate the record (*if it can be found*) and then use the **Edit** method.

```
rst.Edit
rst.Fields("Customer ID") = "XYZ"
rst.Fields("Company Name") = "XYZ Foods Ltd"
rst.Fields("Post Code") = "W12 6RF"
rst.Update
```

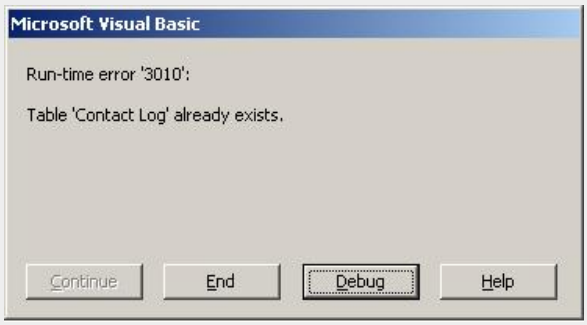
Example - Creating a table:

```
'Opens a connection to the table Customers
'and adds a new record and then updates and closes
Sub CreateTable()
    Dim db As Database
    Dim rst As Recordset
    Dim tbl As TableDef

    Set db = OpenDatabase("C:\db1.mdb")
    Set tbl = db.CreateTableDef("Contact Log")

    With tbl
        .Fields.Append .CreateField("Log ID", dbInteger)
        .Fields.Append .CreateField("Date", dbDate)
        .Fields.Append .CreateField("Caller", dbText)
        .Fields.Append .CreateField("Comment", dbText)
        .Fields.Append .CreateField("Completed", dbBoolean)
        db.TableDefs.Append tbl
    End With

    Set rst = db.OpenRecordset("Contact Log")
    rst.AddNew
    rst.Fields("Log ID") = 1
    rst.Fields("Date") = Date
    rst.Fields("Caller") = "Ben Beitler"
    rst.Fields("Comment") = "Arranged VBA training next week."
    rst.Fields("Completed") = True
    rst.Update

    'close the objects
    rst.Close
    db.Close

    'destroy the variables
    Set rst = Nothing
    Set tbl = Nothing
    Set db = Nothing
```

```
End Sub
```

The above example will create a new table '*Contact Log*', create new fields and then bind it to the new table using **`db.TableDefs.Append tbl`**. Next it will add a single record using the correct data to match the data types as defined in the table.

This procedure will only run once and then cause an error if executed again. This is due to the fact this database cannot contain duplicate named tables.



Therefore, users need to add error-handling procedures as well as testing to see if the table exists.

To delete a table along with its records, use **`db.TableDefs.Delete "Contact Log"`**.

Again an error will be fired if the system cannot locate the table (*misspelling or already deleted*).

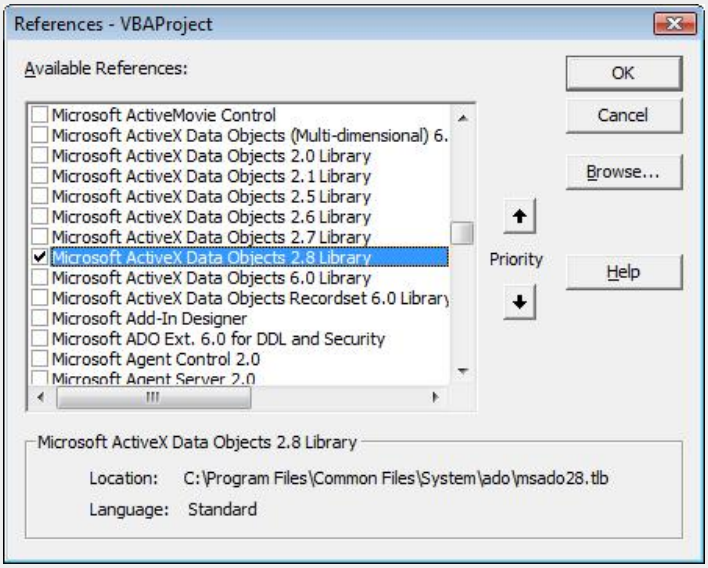
There are many properties and methods of **DAO** which are not covered in this guide. This library allows many ways to produce the same effect which include writing SQL (structured query language).

ADO

ADO stands for **A***ctiveX* **D***ata* **O***bjects* is an alternative method of connecting to a database.

In order to use this feature, users will need to add the **ADO** library to the project.

Choose from the **Tools** menu and select **Reference...**



Note: You may have noticed that there several versions of ActiveX Data Objects in the illustration above. Generally, you should choose the latest version but depending on which version of Excel (or more accurately Windows operating system) try and pick the best fit version. For example 2.8 is for those running on Windows XP where as users would choose 6.0 for Windows Vista.

This library will than allow objects to be created to interrogate a database, tables, fields and return information to populate a spreadsheet. This will also allow users to add, edit, update and delete data to an external file without the need to open the associated application.

An advanced feature of this library (**ADOX**) will even allow users to create, modify and delete structures of a database whether a table, query, stored procedure or fields.

Using the control flow techniques as discussed in this manual, the user can fully control how data should be handled - opening the potential power of VBA.

Note: In order to test this section, users will need an Access database and will need to familiarise themselves with the database. It is not essential to have Microsoft Access loaded as this reference uses the *backdoor* but it will be difficult to check the database without it!

Example - Connecting to a database:

```
Sub ConnectExcelDB()
    Dim cn as ADODB.Connection

    Set cn = New ADODB.Connection
```

```
    With cn
.Provider = "Microsoft.Jet.OLEDB.4.0"
.ConnectionString = "Data Source=C:\pivot data.xls;" & _
             "Extended Properties=Excel 8.0;"
.Open
    End With
End Sub
```

The above example creates a connection and open the workbook 'pivot data'. It requires the **Extended Properties=Excel 8.0** argument (which users need to adjust for their own version of Excel).

```
Sub ConnectAccessDB()
    Dim cn As ADODB.Connection

    Set cn = New ADODB.Connection

    With cn
.Provider = "Microsoft.Jet.OLEDB.4.0"
.ConnectionString = "Data Source=C:\db2.mdb;"
.Open
    End With
End Sub
```

The above example connects to an Access database (db2).

There other ways to connect as well as setting optional arguments which control the method of connection (using **ODBC** or **DSN**-Less etc) which is beyond this article.

Example - Reading from a database:

Using an Access database, users can connect to table, query or write SQL (structured query language) into the calling object.

```
Sub ReadingData()
    Dim cn As ADODB.Connection
    Dim rs As ADODB.Recordset

    Set cn = New ADODB.Connection

    With cn
.Provider = "Microsoft.Jet.OLEDB.4.0"
.ConnectionString = "Data Source=C:\db2.mdb;"
.Open
    End With

    Set rs = New ADODB.Recordset
    'opens a connection to a table called customers.
    rs.Open "Customers", cn, adOpenKeyset, adLockOptimistic, adCmdTable
    'show the second fields value, first record (column 2, row 1).
    Debug.Print rs.Fields(1).Value 'second columns - starts at 0

    rs.Close
    cn.Close

    Set rs = Nothing
    Set cn = Nothing
End Sub
```

The above example creates a connection. It then creates another new object (**rs**) which the recordset of a table, query or SQL source and opens it too.

Now you have a collection of data (all records in that file). Using a property (**Fields**), you can pass either an index or string name into it to refer to any field in that source file and return one of several values (in this case the data value).

Make sure you close and dispose of the objects when finished (and in the correct order) though it will clear and dispose of all objects when the procedure comes to an end - *just good habits of programming!*

To refer to an actual field instead of an index, use **Fields("Customer Name").**

It is good practice to narrow down the recordset to the smallest amount of data in memory which the above example fails to do (all records). Instead, consider passing a query or SQL statement instead:

```
rs.Open "Select * From Customers Where Country='UK';", cn .....
```

There are optional arguments which also help performance and restrictions to an open connection which I've used in my example above **adOpenKeyset, adLockOptimistic, adCmdTable** and will require further investigation to help establish the rule (refer to VBA help for more information).

Example - Writing to a database:

Create a connection and open a recordset (table) to add, edit and delete records.

```
Sub EditingData()
    Dim cn As ADODB.Connection
    Dim rs As ADODB.Recordset

    Set cn = New ADODB.Connection
```

```
    With cn
.Provider = "Microsoft.Jet.OLEDB.4.0"
.ConnectionString = "Data Source=C:\db2.mdb;"
.Open
    End With

    Set rs = New ADODB.Recordset
    rs.Open "Customers", cn, adOpenDynamic, adLockOptimistic
    'edit the second field, first record's value.
    rs.Fields(1).Value = "Always Open QM"
    rs.Update 'save the changes.

    rs.Close
    cn.Close

    Set rs = Nothing
    Set cn = Nothing
End Sub
```

Using the **rs.Update** property enforces any changes to be saved and written to the database.

If you wish add a new record, you can use **rs.AddNew** method but it will still need to use **rs.Update** to save the changes.

Example:

```
Sub NewRecord()
    Dim cn As ADODB.Connection
    Dim rs As ADODB.Recordset

    Set cn = New ADODB.Connection

    With cn
.Provider = "Microsoft.Jet.OLEDB.4.0"
.ConnectionString = "Data Source=C:\db2.mdb;"
.Open
    End With

    Set rs = New ADODB.Recordset

    rs.Open "Customers", cn, adOpenDynamic, adLockOptimistic
    'adding a new record.
    rs.AddNew
    rs.Fields(0).Value = "XYZ" 'customer ID field
    rs.Fields(1).Value = "XYZ Limited" 'customer name field
    rs.Fields(5).Value = "London" 'city field
    rs.Fields(8).Value = "UK" 'country field
    rs.Update 'save the changes.

    rs.Close
    cn.Close

    Set rs = Nothing
    Set cn = Nothing
End Sub
```

The above example populates new values to four fields and then saves the changes. Make sure any record being added satisfies the rules of the data source which is being used to store the data which will include indexing (which is generally a mandatory field).

Other useful methods include **EOF** (*end of file*) and **BOF** (*beginning of file*) which allows you to iterate through records using loops. - *look at the help for more information.*

There is much, much more on this subject (*I've not done this justice*) and users should now be confident to go off and investigate further using various other resources (books and the web!).

Finally, which one to use **DAO** or **ADO**?

There are many arguments which one should use but as a general rule if you are going to communicate with Microsoft '*Jet*' engine (Access, SQL etc) then using **DAO** is quicker and easier to master.

Consider using **ADO** for across platform applications typically over the web (server) and non-Microsoft Window environments which have the capability to create DSN-less connections. It also handles multiple databases at the same time and is considered the standard with other programming languages.

Both have similar members (methods and properties) and can conflict if both are being referenced in the same module.

Next Topic: Input/Output Files

Input/Output Files

VBA Keywords: FreeFile, Do...Loop, Write, Input, Output, EOF, Dubug.Print & Close.

VBA already includes commands to allow data to read or write to external text files. This is more commonly known as **I/O** (Input / Output) and is used to store files in the formats such as 'txt', 'csv' and 'ini' files.

Example of Output Data:

```
Sub BuildTextFile()
    Dim fnum

    fnum = FreeFile()
    Open "C:\vba.txt" For Output As #fnum

    Write #fnum, "Excel VBA", "Day 1"
    Write #fnum, "Excel VBA", "Day 2"
    Write #fnum, "Excel VBA Workshop Q&A", "Day 3"

    Close #fnum
End Sub
```

The above example creates an instance of a file using the **`FreeFile`** function, which returns a unique number (as its handler). The **`Open`** method is used to locate and open the file.

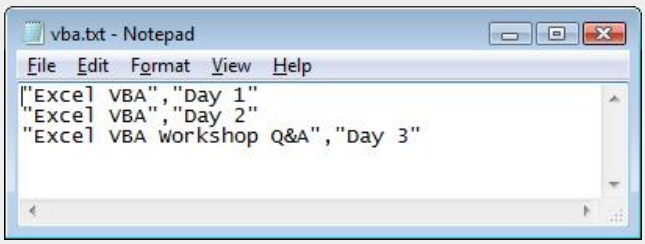
The **`Output`** property tells the system that data is to be written to the named file using the pointer **`#fnum`**.

The **`Write`** method adds line-by-line data to the pointer and then is lost with the **`Close`** method.

Even if the file name does not exist, it will create this file in the specified path but the path must exist.

If the filename already exists, this routine will overwrite (no prompt) and the previous file will be lost.

The file generated is a 'txt' file:



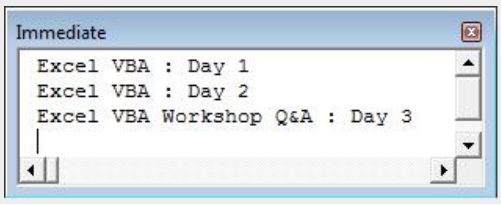
Example of Input Data:

```
Sub ReadTextFile()
    Dim fnum
    Dim strField1 As String, strField2 As String

    fnum = FreeFile()
    Open "C:\vba.txt" For Input As #fnum

    Do Until EOF(fnum)
        Input #fnum, strField1, strField2
        Debug.Print strField1 & " : " & strField2
    Loop

    Close #fnum
End Sub
```

The above example uses the **`Input`** property instead to change the direction of the flow of data (*read from*).

Using the **`EOF`** method, the procedure loops through the delimiter line break until it reaches the end of the file.

To view the results, open the **Immediate** Window (**Ctrl + G**) before running the above procedure:



The above two examples demonstrates how to read and write data to and from external files and will require a little more coding to deal with interaction and variables to make this more flexible (and practical).

Want to teach yourself Access? Free online guide at About Access Databases

Example Code - Snippets

Now that you have (hopefully) reviewed the previous articles on this website VBA reference guide, you may want to browse some example snippets of code which can be used to build up your knowledge and personal library of Excel VBA.

The following links will take you to a particular section to help you find some reference that maybe of interest to you (which can be as simple as a one line piece of code):

Used range of cells - worksheet protection by value type

This sub procedure looks at every cell on the active worksheet and if the cell does not have a formula, a date or text and the cell is numeric; it unlocks the cell and makes the font blue.

For everything else, it locks the cell and makes the font black. It then protects the worksheet.

This has the effect of allowing someone to edit the numbers but they cannot change the text, dates or formulas.

```
Sub SetProtection()
    On Error GoTo errorHandler

    Dim myDoc As Worksheet
    Dim cel As Range
    Set myDoc = ActiveSheet
    myDoc.Unprotect
    For Each cel In myDoc.UsedRange
        If Not cel.HasFormula And _
                        Not TypeName(cel.Value) = "Date" And _
                                        Application.IsNumber(cel) Then
            cel.Locked = False
            cel.Font.ColorIndex = 5
       Else
            cel.Locked = True
            cel.Font.ColorIndex = xlColorIndexAutomatic
       End If
    Next
    myDoc.Protect
    Exit Sub

errorHandler:
     MsgBox "Error"
End Sub
```

Back to top

Basic calculation (Sum) in a range of cells

Enters a value into 10 cells in a column and then sums the values (range) using the sum function.

```
Sub SumRange()
    Dim i As Integer
    Dim cel As Range
    Set cel = ActiveCell
    For i = 1 To 10
```

```
        cel(i).Value = 100
    Next i
    cel(i).Value = "=SUM(R[-10]C:R[-1]C)"
End Sub
```

Other functions can be used as well as changing the range and values to suit.

Another way to write a formula:

```
Sub CalculateFormula()
    Dim s As String

    ActiveCell.Formula = "=" & _
        ActiveCell.Offset(0, -3).Address(False, False) & "/6"
    s = ActiveCell.Offset(0, -16).Address(False, False) _
        & ":" & ActiveCell.Offset(0, -5).Address(False, False) _
            ActiveCell.Formula = "=SUM(" & s & ")/12"
    ActiveCell.Formula = s
End Sub
```

Back to top

Nested For...Next with an If statement

This sub checks values in a range of 10 rows by 5 columns moving left to right, top to bottom, switching the values 'X' and 'O'.

Set a range of 10 x 5 cells with a mixture of 'X's and 'O's.

```
Sub ToggleValues()
    Dim rowIndex As Integer
    Dim colIndex As Integer

    For rowIndex = 1 To 10
        For colIndex = 1 To 5
            If Cells(rowIndex, colIndex).Value = "X" Then
                Cells(rowIndex, colIndex).Value = "O"
            Else
                Cells(rowIndex, colIndex).Value = "X"
            End If
        Next colIndex
    Next rowIndex
End Sub
```

Back to top

Loop through worksheets in a workbook for set ranges

Loops through all worksheets in a workbook and reset values in a specific range(s) on each worksheet to zero where it is not a formula and the cell value is not equal to zero.

```
Sub SetValuesAllSheets()
    Dim wSht As Worksheet
    Dim myRng As Range
    Dim allwShts As Sheets
    Dim cel As Range

    Set allwShts = Worksheets
    For Each wSht In allwShts
        Set myRng = wSht.Range("A1:A5, B6:B10, C1:C5, D4:D10")
        For Each cel In myRng
            If Not cel.HasFormula And cel.Value <> 0 Then
                cel.Value = 0
            End If
        Next cel
    Next wSht
End Sub
```

Change the ranges using a comma separator for each union range.

Modify the condition and its returning value to suit.

Back to top

Worksheet - hidden and visible properties

The distinction between **Hide**(False) and the **xlVeryHidden** constant.

Visible = xlVeryHidden - Sheet/Unhide is greyed out. To unhide sheet, you must set the Visible property to True.

Visible = Hide(or False) - Sheet/Unhide is not greyed out

To hide specific (second) worksheet

```
Sub HideSheet()
    Worksheets(2).Visible = Hide  'you can use Hide or False
End Sub
```

To make a specific (second) worksheet very hidden

```
Sub VeryHiddenSheet()
    Worksheets(2).Visible = xlVeryHidden 'menu item is not available
End Sub
```

To unhide a specific worksheet

```
Sub UnHideSheet()
    Worksheets(2).Visible = True
End Sub
```

To toggle between hidden and visible

```
Sub ToggleHiddenVisible()
    Worksheets(2).Visible = Not Worksheets(2).Visible
End Sub
```

Toggle opposite visibility (error will happen as all worksheets cannot be hidden, at least one must be visible in a workbook).

```
Sub ToggleAllSheets()
    On Error Goto errorHandler
    Dim wSh As Worksheet

    For Each wSh In Worksheets
        wSh.Visible = Not wSh.Visible
    Next
    Exit Sub
errorHandler:

End Sub
```

To set the visible property to True on all sheets in a workbook.

```
Sub UnHideAll()
    Dim wSh As Worksheet

    For Each sh In Worksheets
        wSh.Visible = True
    Next
End Sub
```

Back to top

Inserting worksheets avoiding duplicate names, naming & validations

Checks to see if sheet already exists with the name '*MySheet*' and does not add it again as Excel cannot store duplicate worksheet names in a workbook.

Validation if name already exists or no name stored or if it is a number as its name.

```
Sub AddUniqueSheet()
    Dim ws As Worksheet
    Dim newSheetName As String

    newSheetName = "MySheet" 'Substitute your name here
    For Each ws In Worksheets
        If ws.Name = newSheetName Or newSheetName = "" Or _
                                      IsNumeric(newSheetName) Then
            MsgBox "Sheet '" & newSheetName & "' already exists _
                                     or name is invalid", vbInformation
            Exit Sub
        End If
    Next
    Sheets.Add Type:="Worksheet"
    With ActiveSheet 'Move to last position
        .Move After:=Worksheets(Worksheets.Count)
        .Name = newSheetName
    End With
End Sub
```

Adds new worksheet with the month and year as its name and sets the range("A1:A5") from Sheet1 to new worksheet.

This can only be executed once for the same period due to excel not allowing duplicate worksheets names.

Make sure you have a worksheet called '*Sheet1*' and that its range '*A1:A5*' has some content which to copy across.

```
Sub AddSheet()
    Dim wSht As Worksheet
    Dim shtName As String

    shtName = Format(Now, "mmmm_yyyy") 'current month & year
    For Each wSht In Worksheets
        If wSht.Name = shtName Then
            MsgBox "Sheet already exists...Make necessary corrections _
                                                    and try again."
            Exit Sub
        End If
    Next wSht
    Sheets.Add.Name = shtName
    Sheets(shtName).Move After:=Sheets(Sheets.Count)
    Sheets("Sheet1").Range("A1:A5").Copy _
                Sheets(shtName).Range("C1") 'range("C1") = starting point
End Sub
```

Copies the contents of the first positioned worksheet to a new worksheet ('*NewSheet*') validating if sheet exists first.

```
Sub CopySheet()
    Dim wSht As Worksheet
    Dim shtName As String
```

```
    shtName = "NewSheet" 'change the name if required
    For Each wSht In Worksheets
        If wSht.Name = shtName Then
            MsgBox "Sheet already exists...Make necessary " & _
                                            "corrections and try again."
            Exit Sub
        End If
     Next wSht
     Sheets(1).Copy Before:=Sheets(1)
     Sheets(1).Name = shtName
     Sheets(shtName).Move After:=Sheets(Sheets.Count)
End Sub
```

Index number for a sheet can be used instead of the actual string name. This is useful if name is not known or you want to control the order position of the sheet in question.

Back to top

InputBox and Message Box examples

```
Sub CalcPay()
    On Error GoTo HandleError

    Dim hours
    Dim hourlyPay
    Dim payPerWeek

    hours = InputBox("Please enter number of hours worked", "Hours Worked")
    hourlyPay = InputBox("Please enter hourly pay", "Pay Rate")
    payPerWeek = CCur(hours * hourlyPay)
    MsgBox "Pay is: " & Format(payPerWeek, "£##,##0.00"), , "Total Pay"
HandleError: 'any error - gracefully end
End Sub
```

No communication with Excel is required for this example and can be started from within the VB Editor.

To split a single line of execution into multiple lines, use the underscore character (_).

What impact will this have if you use the integer function (Int()) instead of the currency functions (CCur)?

Other functions: CDbl (double) and CSng (single).

Date Entry & Formula with **InputBox** which prompts the user for the number of times to iterate, creates heading and calculates gross values with final totals at the end of the columns.

```
Sub ProcessTransactions()
    ActiveCell.Value = "NET"
    ActiveCell.Offset(0, 1).Value =
    "GROSS" ActiveCell.Offset(1, 0).Select

    y = InputBox("How Many transactions?", , 5)
    For counter = 1 To y
        x = InputBox("Enter Net")
        ActiveCell.Value = x
        ActiveCell.NumberFormat = "#,##0.00"
        ActiveCell.Offset(0, 1).FormulaR1C1 = "=RC[-1]*1.175"
        ActiveCell.Offset(0, 1).NumberFormat = "£ 0.00"
        ActiveCell.Offset(1, 0).Select
    Next counter

    ActiveCell.FormulaR1C1 = "=SUM(R[-" & y & "]C:R[-1]C)"
    'Variable y concatenated to formula (Sum)
    ActiveCell.Offset(0, 1).FormulaR1C1 = "=SUM(R[-" & y & "]C:R[-1]C)"
    ActiveCell.Range("A1:B1").Select
    Selection.Font.Bold = True

    With Selection.Borders(xlEdgeTop)
        .LineStyle = xlContinuous
        .Weight = xlThin
        .ColorIndex = xlAutomatic
    End With

    With Selection.Borders(xlEdgeBottom)
        .LineStyle = xlDouble
        .Weight = xlThick
        .ColorIndex = xlAutomatic
    End With
End Sub
```

The above is A For Next Example with InputBox Function, With Block and Offset method

Back to top

Printing examples

To control orientation and defined name range - 1 copy.

```
Sub PrintReport1()
    Sheets(1).PageSetup.Orientation = xlLandscape
```

```
    Range("Report").PrintOut Copies:=1
End Sub
```

To print several ranges on the same sheet -1 copy

```
Sub PrintReport2()
    Range("HVIII_3A2").PrintOut
    Range("BVIII_3").PrintOut
    Range("BVIII_4A").PrintOut
    Range("HVIII_4A2").PrintOut
    Range("BVIII_5A").PrintOut
    Range("BVIII_5B2").PrintOut
    Range("HVIII_5A2").PrintOut
    Range("HVIII_5B2").PrintOut
End Sub
```

To print a defined area, centre horizontally, with 2 rows as titles, in portrait orientation and fitted to page wide and tall - 1 copy.

```
Sub PrintReport3()
    With Worksheets("Sheet1")
        .PageSetup
        .CenterHorizontally = True
        .PrintArea = "$A$3:$F$15"
        .PrintTitleRows = ("$A$1:$A$2")
        .Orientation = xlPortrait
        .FitToPagesWide = 1
        .FitToPagesTall = 1
    End With
    Worksheets("Sheet1").PrintOut
End Sub
```

To print preview, control the font and to pull second line of header (“A1”) from first worksheet.

```
Sub PrintHeaderPreview()
    ActiveSheet.PageSetup.CenterHeader = "&""Arial,Bold Italic""&14 _
        My Report" & Chr(13) & Sheets(1).Range("A1")
    ActiveWindow.SelectedSheets.PrintPreview
End Sub
```

"&""Arial,Bold Italic""&14 = fields used in page set-up of header/footer

Back to top

General application commands

Using the shortcut approach to assign a cell with an Excel function.

```
Sub GetSum()
    [A1].Value = Application.Sum([E1:E15])
End Sub
```

Can use an absolute reference: Range("A1") = Application.Sum([E1:E15])

Other functions - AVERAGE, MIN, MAX, COUNT, COUNTBLANK, COUNTA, VLOOUKP etc…

Enables the use of events if disabled (worksheet/workbook).

```
Sub EnableEventReset()
    Application.EnableEvents = True
End Sub
```

To display the full path and filename of the current workbook (Function)

```
Sub FormatHeader()
    With ThisWorkbook
        .Worksheets("MySheet").PageSetup.LeftHeader = .FullName
    End With
End Sub
```

Capture object (chart) into as separate file

```
Sub ExportToJPG()
    ActiveChart.Export FileName:="c:\Mychart.jpeg", FilterName:="JPG"
End Sub
```

Make sure chart is selected first

Add a custom button to the ‘Chart’ quick access toolbar.

Assign and un-assign a function key to a procedure

```
Sub Set_FKeys()
    Application.OnKey "{F3}", "MySub"
End Sub
```

```
Sub Restore_FKeys()
    Application.OnKey "{F3}"
End Sub
```

Can be assigned to the event of when a workbook opens a closes.

Cursors

```
Sub ShowHourGlass()
    Application.Cursor = xlWait
End Sub
```

```
Sub ResetCursor()
    Application.Cursor = xlNormal
End Sub
```

Can also be **xlNorthwestArrow** and **xlIBeam**.

Some more to finish off with...

```
With ActiveWindow
    .DisplayGridlines = Not .DisplayGridlines
    .DisplayHeadings = Not .DisplayHeadings
    .DisplayHorizontalScrollBar = Not .DisplayHorizontalScrollBar
    .DisplayVerticalScrollBar = Not .DisplayVerticalScrollBar
    .DisplayWorkbookTabs = Not .DisplayWorkbookTabs
End With
```

```
With ActiveWindow
    .DisplayFormulaBar = Not .DisplayFormulaBar
    .DisplayStatusBar = Not .DisplayStatusBar
End With
```

```
Selection.Clear 'clears all attributes
Selection.ClearFormats 'clears only formats
Selection.ClearContents 'clears only content (DEL)
```

Active cell moves I row, 1 column in for selection

```
Sub ActiveCellInRange()
    Range("A11:D15").Select
    Selection.Offset(1, 1).Activate
End Sub
```

Back to top

Ranges - various examples

To add a range name for known range

```
Sub AddName1()
    ActiveSheet.Names.Add Name:="MyRange1", RefersTo:="=$A$1:$B$10"
End Sub
```

To add a range name based on a selection.

```
Sub AddName2()
    ActiveSheet.Names.Add Name:="MyRange2", RefersTo:="=" & _
                                                Selection.Address()
End Sub
```

To add a range name based on a selection using a variable.

```
Sub AddName3()
    Dim rng As String

    rng = Selection.Address
    ActiveSheet.Names.Add Name:="MyRange3", RefersTo:="=" & rng
End Sub
```

To add a range name based on current selection.

```
Sub AddName4()
    Selection.Name = "MyRange4"
End Sub
```

Deletes all named ranges

```
Sub DeleteAllRanges()
    Dim rName As Name

    For Each rName In ActiveWorkbook.Names
        rName.Delete
    Next rName
End Sub
```

Scrolls the spreadsheet to where the active cell is.

```
Sub ScreeTopLeft()
    ActiveCell.Select
    With ActiveWindow
        .ScrollColumn = ActiveCell.Column
        .ScrollRow = ActiveCell.Row
    End With
End Sub
```

Function to return a range object.

```
Function LastCell(ws As Worksheet) As Range
    Dim LastRow As Long, LastCol As Long
    'Error-handling is here in case there is not any
    'data in the worksheet
    On Error Resume Next

    With ws
        'Find the last row
        LastRow = .Cells.Find(What:="*", _
             SearchDirection:=xlPrevious, _
             SearchOrder:=xlByRows).Row
        'Find the last column
        LastCol = .Cells.Find(What:="*", _
             SearchDirection:=xlPrevious, _
             SearchOrder:=xlByColumns).Column
    End With
    'Finally, initialize a Range object variable for
    'the last populated row.

    Set LastCell = ws.Cells(LastRow, LastCol)
End Function
```

Call procedure for above (not for a worksheet function call)

```
Sub ShowLastCell()
    MsgBox LastCell(Sheet1).Address(False, False)
End Sub
```

Try MsgBox LastCell(Sheet1).Row

Try MsgBox LastCell(Sheet1).Column

Check to see if active cell is in range A1:A10.

```
Sub CheckRange()
    Dim rng As Range

    Set rng = Application.Intersect(ActiveCell, Range("A1:A10"))
    If rng Is Nothing Then
        MsgBox "It is not in the range.", vbInformation
    Else
        MsgBox "It's in the range called 'A1:A10'!", vbCritical
    End If
End Sub
```

Current selected rows or cells in a column.

```
Sub MyCount()
    Dim myCount As Long
    myCount = Selection.Rows.Count
    MsgBox myCount
End Sub
```

Number of worksheets in a workbook.

```
Sub MySheetCount()
    Dim myCount As Long
    myCount = Application.Sheets.Count
    MsgBox myCount
End Sub
```

Copy and paste a range (A1:A3) to active cell in same worksheet.

```
Sub CopyRange1()
    Range("A1:A3").Copy Destination:=ActiveCell
End Sub
```

Copy and paste a range (A1:A3) to active cell from 'Sheet3'.

```
Sub CopyRange2()
    Sheets("sheet3").Range("A1:A3").Copy Destination:=ActiveCell
End Sub
```

Show current active cell position (address) – co-ordinate

```
Sub MyPosition()
    Dim myRpw, myCol
    myRow = ActiveCell.Row
    myCol = ActiveCell.Column
    MsgBox myRow & "," & myCol
End Sub
```

Specific Range references

`Range("A1")`	Cell A1
`Range("A1:E10")`	Range A1 to E10
`[A1]`	Cell A1
`[A1:E10]`	Range A1 to E10

`ActiveCell.Range("A2")`	The cell below the active cell
`Cell(1)`	Cell A1
`Range(Cells(1,1),Cell(10,5))`	Range A1 to E10
`Range("A:A")`	Column A
`[A:A]`	Column A
`Range("5:5")`	Row 5
`[5:5]`	Row 5
`Sheets("Sheet1")`	Sheet called Sheet1
`Worksheets("Sheet1")`	Worksheets called Sheet1
`Sheets(2)`	Second worksheet in workbook
`Worksheets(3)`	Third worksheet in workbook
`Worksheets("Sheet1").Range("A1")`	Cell A1 in Sheet1
`[Sheet1].[A1]`	Cell A1 in Sheet1
`ActiveSheet.Next`	The sheet after the active sheet
`Workbook("Test")`	Workbook file called Test.xls

Back to top

Navigation in a worksheet using Offset

```
Sub MoveDown()
    ActiveCell.Offset(1, 0).Select
End Sub
```

```
Sub MoveUp()
    ActiveCell.Offset(-1, 0).Select
End Sub
```

```
Sub MoveRight()
    ActiveCell.Offset(0, 1).Select
End Sub
```

```
Sub DownLeft()
    ActiveCell.Offset(0, -1).Select
End Sub
```

```
Sub LastCellInRange()
    Range(ActiveCell.Address).End(xlDown).Select
    Range(ActiveCell.Address).End(xlToRight).Select
End Sub
```

Back to top

Read Window documents

Calling sub procedure passing a string argument.

Use the Private keyword, which is local and invisible via Excel application.

```
Private Sub ReadFiles(Path As String)
    Dim FileName As String

    'Initialize a string variable for the first file
    'in a specified directory. This sets the Dir( )
    'function to that directory.

    Select Case Right(Path, 1)
        Case "\":    FileName = Dir(Path)
        Case Else:   FileName = Dir(Path & "\")
    End Select

    'Loop through the specified directory until the
    'Dir( ) function returns an empty string, indicating
    'there are not any more contents to be evaluated.

    Do While Len(FileName) > 0
        'Print each file name to the immediate (debug) window
        Debug.Print FileName
        'Re-initialize the string variable to the next
        'file in the directory
        FileName = Dir()
    Loop
End Sub
```

Call the above in a separate procedure

```
Sub ListFiles()
    ReadFiles "c:\winnt"
End Sub
```

Back to top

General function examples

Displays the period quarter.

```
Function Qtr(dtOrig As Date) As String
    Dim qtrNo As Integer
    Dim sQtr As String

    Select Case Format(dtOrig, "q")
        Case Is = 1
            sQtr = "1st Qtr"
        Case Is = 2
            sQtr = "2nd Qtr"
        Case Is = 3
            sQtr = "3rd Qtr"
        Case Is = 4
            sQtr = "4th Qtr"
        Case Else 'assume 1
            sQtr = "1st Qtr"
    End Select
    Qtr = sQtr
End Function
```

In a worksheet, enter the formula: =Qtr("01/01/2010")

Show full path and file name in a worksheet.

```
Function FileName()
    FileName = Application.Caller.Parent.Parent.FullName
End Function
```

In a worksheet, enter the formula: **=FileName()**

Return the difference in percentage terms of two values (increase/decrease).

```
Function PChange(OrigVal As Double, NewVal As Double) As Single
    If OrigVal = 0 Then
        PChange = ""
    Else
        PChange = ((NewVal - OrigVal) / Abs(OrigVal))
    End If
End Function
```

In a worksheet, enter the formula: **=PChange(100,150) = 50%**
(0.5 for unformatted)

Gross Price (inc)

```
Function TotalValue(Qty As Double, UPrice As Double) As Double
    TotalValue = Format((Qty * UPrice * 1.175), "£#,##0.00")
End Function
```

Age (simple)

```
Function Age2(DOB)
    Age2 = Int((Now() - DOB) / 365.25) & " Years old"
End Function
```

Age (alternative)

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

Returns the cell in range which is underline (single style) or the word "unknown"

```
Public Function GetUnderlinedCell(CellRef As Range) As String
    Dim c As Integer
    Dim sResult As String

    'Force Running when Recalculating Since Formatting Only
    Application.Volatile True

    'Assume Unknown
    sResult = "Unknown"

    'Loop Thru Each Column and Test for Underline
    For c = 1 To CellRef.Columns.Count
        If CellRef.Columns(c).Font.Underline = xlUnderlineStyleSingle Then
            sResult = CellRef.Columns(c).Value
        End If
    Next c

    'Return Results
    GetUnderlinedCell = sResult
End Function
```

Visual Basic Functions - **Choose** (Lookup).

```
Sub LookupExample()
    Dim strMonth As String
    Dim bytCurMonth As Byte

    bytCurMonth = Month(Date)
    strMonth = Choose(bytCurMonth, "Jan", "Feb", "Mar", "Apr", _
                   "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")
    MsgBox "Current month is " & strMonth
End Sub
```

Also, take a look at the **`Switch()`** function using VBA Help.

Back to top

Creates a new word document

Creates a new word document and populates the contents of cell "*B1*" along with some basic formatting.

You need create a reference to the Word Object Library (8.0/9.0/10.0/11.0) in the VB Editor

```
Sub CreateMSWordDoc()
    On Error GoTo errorHandler

    Dim wdApp As Word.Application
    Dim myDoc As Word.Document
    Dim mywdRange As Word.Range

    Set wdApp = New Word.Application

    With wdApp
        .Visible = True
        .WindowState = wdWindowStateMaximize
    End With

    Set myDoc = wdApp.Documents.Add
    Set mywdRange = myDoc.Words(1) 'index range?

    With mywdRange
        .Text = Range("B1") & vbNewLine & "This above text is _
                                                stored in cell 'B1'."
        .Font.Name = "Comic Sans MS"
        .Font.Size = 12
        .Font.ColorIndex = wdGreen
        .Bold = True
    End With

errorHandler:
    Set wdApp = Nothing
    Set myDoc = Nothing
    Set mywdRange = Nothing
End Sub
```

Back to top

Creates an Outlook message

Creates an Outlook message (new) populating the 'To', 'subject' and 'Body' properties with the content stored in cell "A1".

You need create a reference to the Outlook Object Library (8.0/9.0/10.0/11.0) in the VB Editor

```
Sub SendMessage()
    Dim objOL As New Outlook.Application
    Dim objMail As MailItem

    Set objOL = New Outlook.Application
    Set objMail = objOL.CreateItem(olMailItem)

    With objMail
        .To = "name@domain.com"
        .Subject = "Excel VBA to Outlook Message Example"
        .Body = "This is an automated message from Excel. " & _
               vbNewLine & "The content of cell reference 'A1' is: " & _
                                                    Range("A1").Value
        .Display
    End With

    Set objMail = Nothing
    Set objOL = Nothing
End Sub
```

Back to top

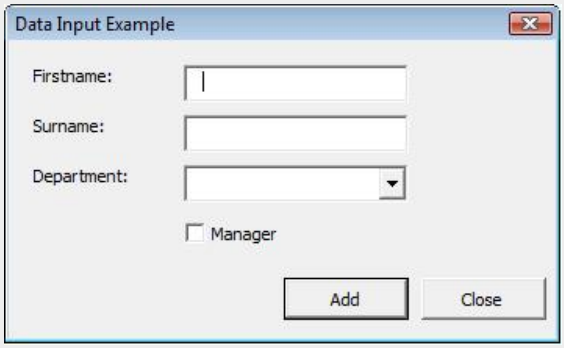
User Form - Input Example

This article steps you through a simple user form input which adds record information into a worksheet.

The form is a basic design with the emphasis on how to build the form and code it to respond to the functionality we are after.

Here's what we are are going to achieve:

1. A user form will load from a button on a worksheet.



2. Users must complete **Firstname**, **Surname** and choose a **Department** (which will be coded as mandatory fields).
3. When choosing the **Add** button it will append to the worksheet (called *Data*) and always find the next available blank row to populate.
4. The form will remain open clearing the values ready for the next record input until the **Close** button is clicked.
5. A private macro (from the standard module) calls the user form (via its worksheet button).

There are properties and code for the form, two and buttons and drop-down combo box which we will need to add the form's private module.

The order in creating such a feature should loosely follow these steps:

1. Create the user form canvas.
2. Add the controls to the form and set various basic properties (including names).
3. Add code to the form's controls
4. Code the interaction to the worksheet (& prepare the worksheet layout too).
5. Add a macro to call the form and attach to a button on the worksheet.
6. Test the process!

Create the user form canvas

Add a new blank user form the VBA Project.

In the VBE Editor, select **Insert**, **UserForm.**

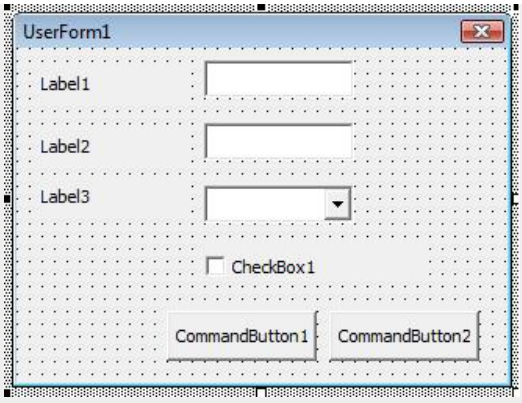
Add the controls to the form

You need to add the following controls:

1. Two Command Buttons
2. Three Labels
3. Two TextBoxes
4. One ComboBox
5. One CheckBox

Place the controls roughly where you would like to use these control and resize the form.

Don't worry about the exact position for now:



Setting the properties to each control

The following controls can be set using the **Properties** Window (**F4** function key).

First single click to select the control (so it has the focus) and then from the properties changes their settings.

Here's the table for the above controls (and user form itself):

Control	***Property***	***Value***
CommandButton1	Name	cmdAdd
	Caption	Add
	Default	True
	Height	20
	Width	60
	Left	132
	Top	114
	TabIndex	4
CommandButton2	Name	cmdClose
	Caption	Close
	Cancel	True
	Height	20
	Width	60
	Left	198
	Top	114
	TabIndex	5
Label1	Caption	Firstname:
	Height	18
	Left	12
	Top	12
	Width	72
Label2	Caption	Surname:
	Height	18
	Left	12
	Top	36
	Width	72
Label3	Caption	Department:
	Height	18
	Left	12
	Top	60
	Width	72
TextBox1	Name	txtFName
	Height	18
	Left	84
	Top	12
	Width	108
	TabIndex	0
TextBox2	Name	txtSName
	Height	18
	Left	84
	Top	36
	Width	108
	TabIndex	1

ComboBox1	Name	cboDept
	Height	18
	Left	84
	Top	60
	Width	108
	TabIndex	2
CheckBox1	Name	chkManager
	Caption	Manager
	Height	18
	Left	84
	Top	84
	Width	108
	TabIndex	3
UserForm1	Name	frmDataInput
	Caption	Data Input Example
	Height	162.75
	Width	267

You can change some of these properties to taste - this is what I'm using in this example.

Adding code to controls

The next step is start coding the form and it's important that you have at least named the controls you wish code as it will generate its own event signature.

Starting with the **Close** button which will simply close and end the user form.

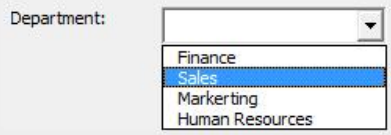
```
Private Sub cmdClose_Click()
    'close the form (itself)
    Unload Me
End Sub
```

`Unload Me` refers to itself which is quick and easy. To explicitly close a user form, you refer to actual name of the form. Therefore, using **`Unload frmDataInput`** will be the same outcome.

Next, lets add code (run time) to populate the ComboBox control (**cboDept**) which will dynamically create four fixed options to choose from.

```
Private Sub UserForm_Initialize()
    Me.cboDept.AddItem "Finance"
    Me.cboDept.AddItem "Sales"
    Me.cboDept.AddItem "Markerting"
    Me.cboDept.AddItem "Human Resources"

    Me.txtFName.SetFocus 'position the cursor in this control

End Sub
```

As the form loads (initialises), it adds four items to the **cboDept** control and then positions the cursor in **txtFName** ready for the user to start keying in data.



You could of course set this in the properties (**RowSource**) for **cboDept** instead which refers to range of cells in a worksheet.

The final piece of code is attached the **cmdAdd** button control so when users click this event, it will add the details to the worksheet (*Data*).

```
Private Sub cmdAdd_Click()
    Dim i As Integer

    'position cursor in the correct cell A2.
    Range("A2").Select
    i = 1 'set as the first ID

    'validate first three controls have been entered...
    If Me.txtFName.Text = Empty Then 'Firstname
        MsgBox "Please enter firstname.", vbExclamation
        Me.txtFName.SetFocus 'position cursor to try again
        Exit Sub 'terminate here - why continue?
    End If

    If Me.txtSName.Text = Empty Then 'Surname
        MsgBox "Please enter surname.", vbExclamation
        Me.txtSName.SetFocus 'position cursor to try again
        Exit Sub 'terminate here - why continue?
    End If
```

```
    If Me.cboDept.Text = Empty Then 'Department
        MsgBox "Please choose a department.", vbExclamation
        Me.cboDept.SetFocus 'position cursor to try again
        Exit Sub 'terminate here - why continue?
    End If

    'if all the above are false (OK) then carry on.
    'check to see the next available blank row start at cell A2...
    Do Until ActiveCell.Value = Empty
        ActiveCell.Offset(1, 0).Select 'move down 1 row
        i = i + 1 'keep a count of the ID for later use
    Loop

    'Populate the new data values into the 'Data' worksheet.
    ActiveCell.Value = i 'Next ID number
    ActiveCell.Offset(0, 1).Value = Me.txtFName.Text 'set col B
    ActiveCell.Offset(0, 2).Value = Me.txtSName.Text 'set col C
    ActiveCell.Offset(0, 3).Value = Me.cboDept.Text 'set col D

    'Is this person the manager?
    If Me.chkManager.Value = True Then 'yes
        ActiveCell.Offset(0, 4).Value = "Yes" 'Col E
    Else
        ActiveCell.Offset(0, 4).Value = "No" 'Col E
    End If

    'Clear down the values ready for the next record entry...
    Me.txtFName.Text = Empty
    Me.txtSName.Text = Empty
    Me.cboDept.Text = Empty
    Me.chkManager.Value = False

    Me.txtFName.SetFocus 'positions the cursor for next record entry

End Sub
```

The above should be easy to follow (look at the comments).

We don't have to tell the system which worksheet to be in as it is going to be called from a control (worksheet button) where the data is held in the same worksheet and then hide this procedure from the Macros dialog box stopping any other way for this form to be called.

Create a worksheet button

In the worksheet in Excel, click the **Developer** tab from the *Ribbon Bar* and the **Insert** icon to drop-down a list of controls.

Choose the **Button** control icon from the Forms (*section*) and draw a button where you wish to place it (top row, frozen pane area).

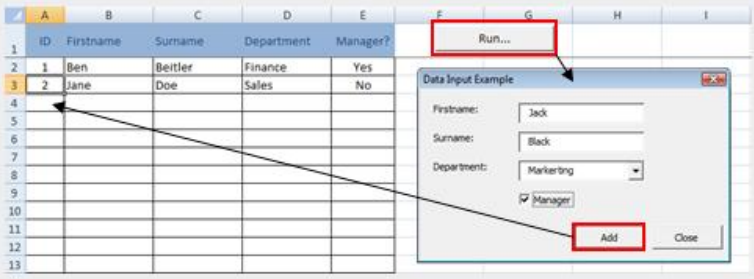
In the assigning macro pop-up dialog box, click the **New...** button to create a module and signature and add the following code:

```
Sub Button1_Click()
    'load the form
    frmDataInput.Show
End Sub
```

Add the keyword **Private** before the **Sub** keyword to hide this from the macro dialog box.

TEST IT OUT!

This is how the form looks as it is called from the worksheet button control from the *Data* worksheet.



Data Input User Form – populates the next available row (**Data** worksheet)

Want to teach yourself Access? Free online guide at About Access Databases