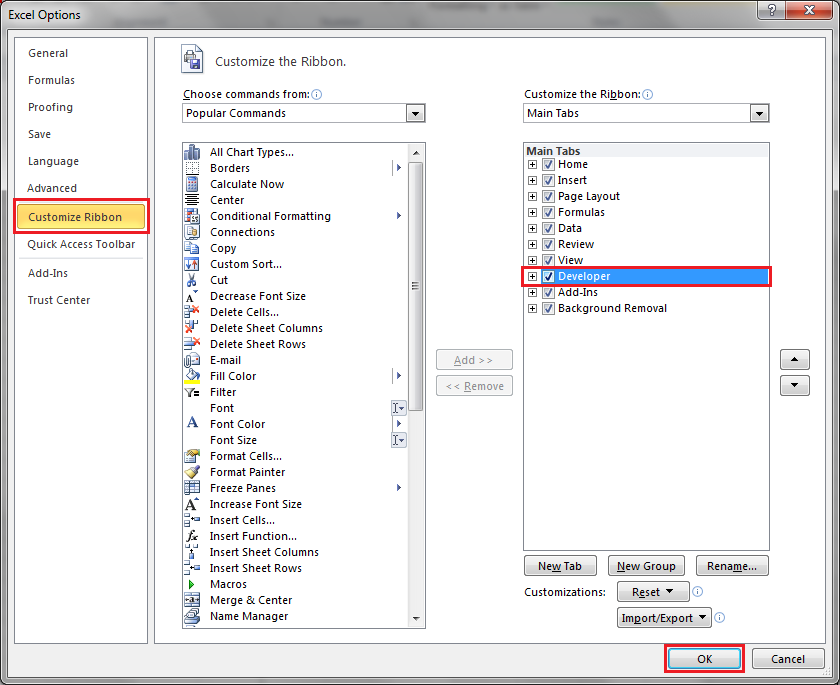
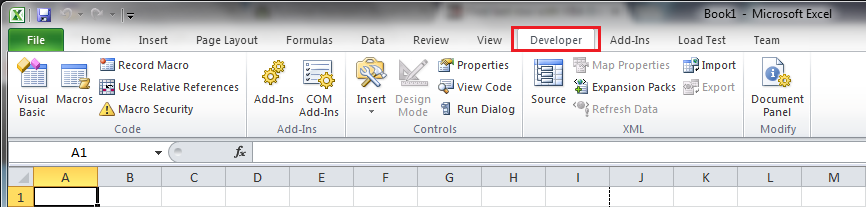
# INTRODUCTION

VBA (Visual Basic for Applications) is the programming language of Excel and other Office programs. With Excel VBA we can automate tasks in Excel by writing macros.

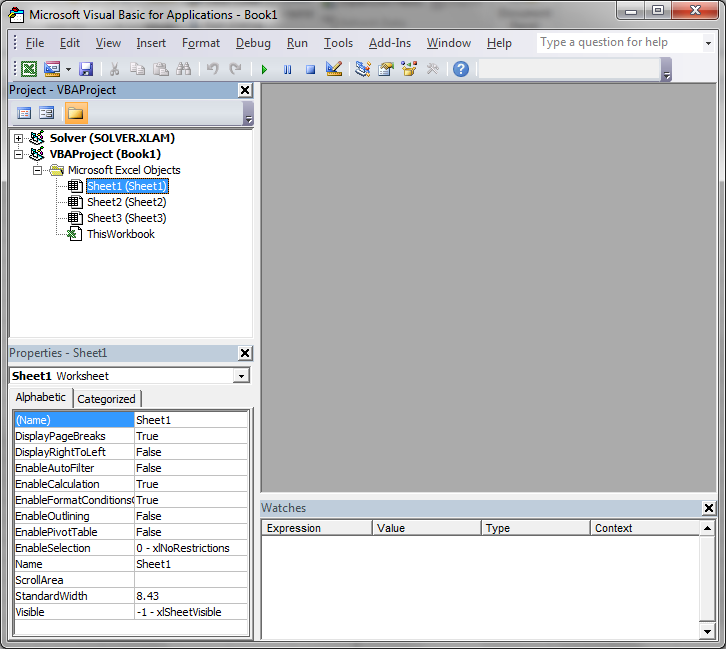
VBA is available in excel under Developer tab and same can be enabled as shown below.



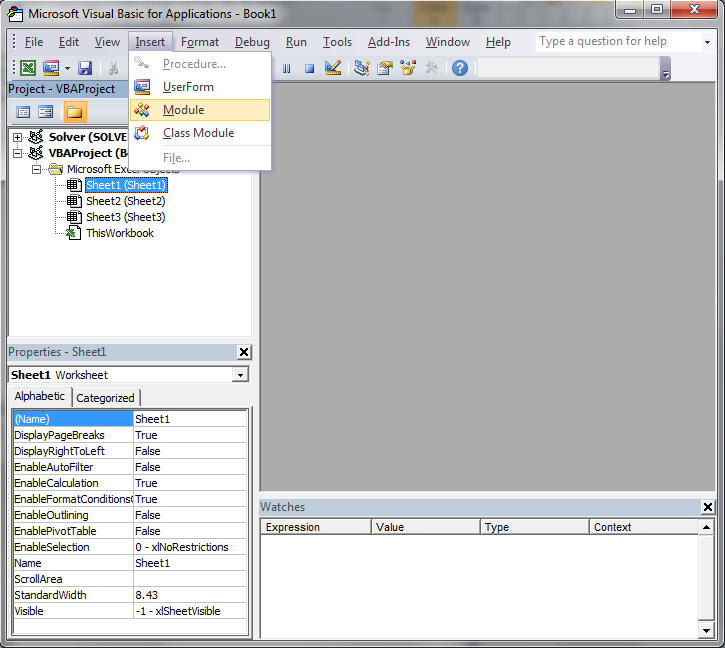
Once done we can start writing VBA code using Visual Basic or Record Marco option available under Developer tab.



From the Developer Tab, click Visual Basic on the left side of your window or use the keyboard shortcut Alt+F11.



To write a code which is available to all the sheets in the workbook, add a module

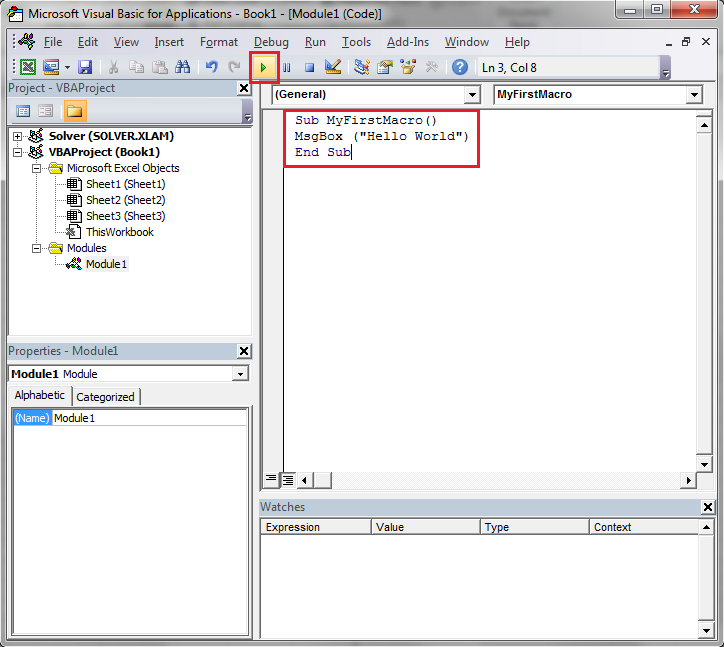


Module is used to store the vba code written by the developer so that it can be executed in the workbook to automate the tasks. Module contains user defined procedures and functions.

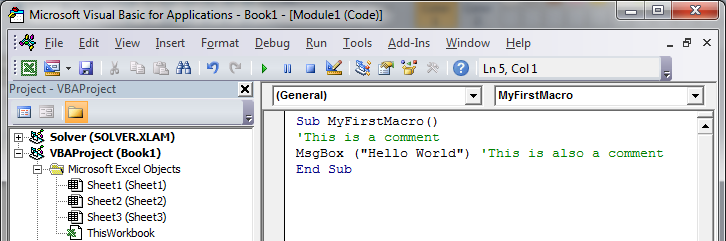
To store vba code in excel, file should be saved as macro enabled workbook with .xlsm extension.

VBA code is written on the vba editor window. Below in the screenshot we can also see Project Explorer and Properties window.

Press F5 to run the procedure and F8 to debug the procedure.



Any text appearing after an apostrophe will be ignored by the VBA interpreter.

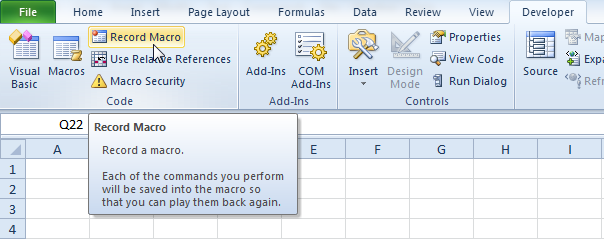


# MACRO RECORDER

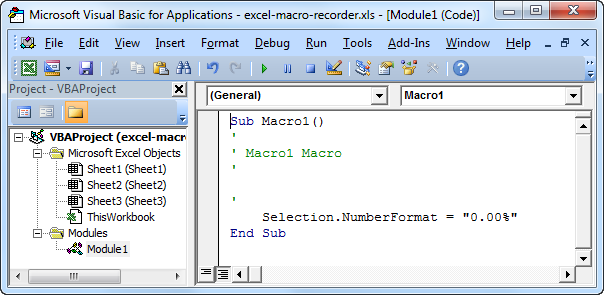
The Macro Recorder, a very useful tool included in Excel VBA, records every task you perform with Excel. All you have to do is record a specific task once. Next, you can execute the task over and over with the click of a button. The Macro Recorder is also a great help when you don't know how to program a specific task in Excel VBA. Simply open the Visual Basic Editor after recording the task to see how it can be programmed.

Unfortunately, there are a lot of things you cannot do with the Macro Recorder. For example, you cannot loop through a range of data with the Macro Recorder. Moreover, the Macro Recorder uses a lot more code than is required, which can slow your process down.

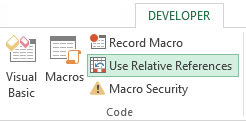
To record a macro, click o n the “Record Macro” button available under developer tab.



Once you are done with the macro recording, click stop recording and a new module containing the code will be available as shown below:



Use relative reference while recording macro if you want relative reference to your range instead of hard coded cell address



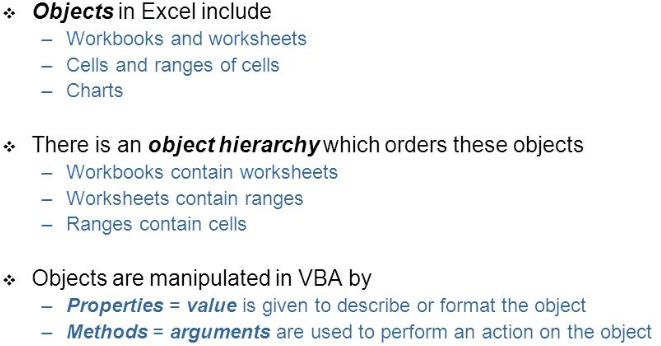
# WHEN THINGS GO WRONG WITH MACRO

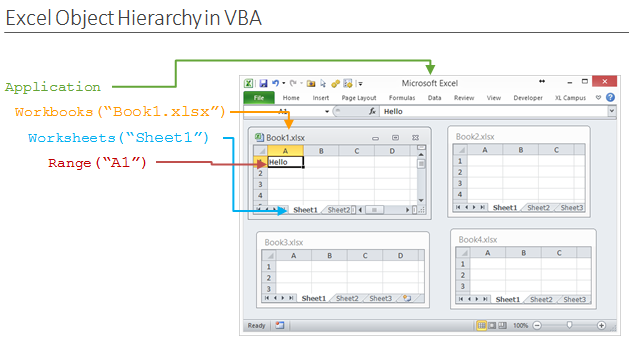
It is very important to know that all the changes done by macro code to the worksheet data cannot be undo as the undo button (CTRL + Z) does not work. Hence its is wise to always keep a backup of data.

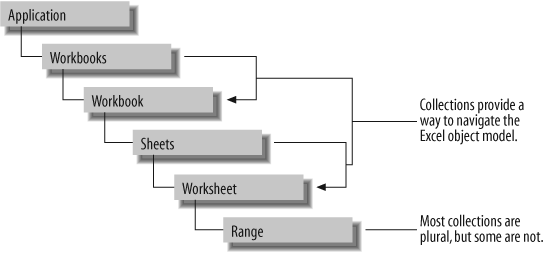
Also, if you have run a macro and deleted some data by mistake, instead of saving the changes if you close the workbook without saving the changes you will get your deleted data back.

# OBJECT HIERARCHY

The VBA objects are organized in a hierarchy, which makes them easier to reference. At the top of this hierarchy is the Excel Application. All the objects within Excel are members or sub-members of the Application object.







Here is a**fully qualified line of code**that starts at the application level.

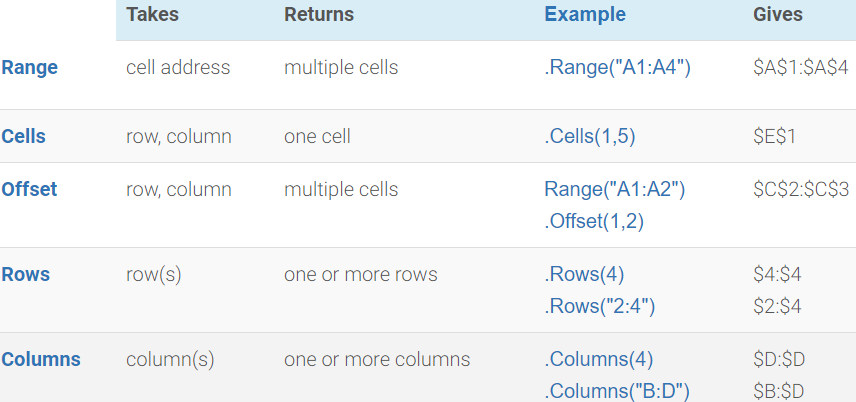


# RANGE OBJECT

The VBA Range Object represents a cell or multiple cells in your Excel worksheet. Properties and Methods of the Range Object are used to manipulate cell values, change formatting and return attributes.

A Property stores information about the object, such as its address ($A$1, for example), its column number, and even its font name. These object properties can be assigned to other variables or can be set to new values.

A Method, on the other hand, is an action the object can perform. The Range Object can perform many actions, including being selected, cleared, copied and sorted.



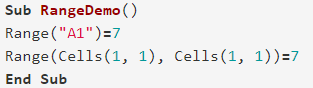
## RANGE ARGUMENTS

The VBA Range Object accepts two arguments: Cell1 and Cell2. The second argument is optional

VBA Range Object

You can pass an address to the Range Object with a string or with the Cells property. Standard address string notation is quite similar to how you would reference a cell in Excel. “B3” or “$B$3” would denote the third row, second column. “A1:B3” or “$A$1:$B$3” would represent the entire range from the first row, first column to the third row, second column.

The easiest way to learn is by playing around, so let’s start playing around. Open up the Visual Basic Editor and create a macro titled RangeDemo (). You’ll see the following pairs are equivalent:



Or as



You’ll notice the second item in each of the previous examples references an argument called Cells. The Cells property, or Range.Cells property, accepts two arguments: Row first, Column second.

VBA Cells Property

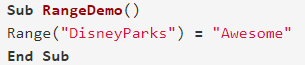
Each argument must be an integer (a whole number). Specifically, each argument must be an integer greater than 0 since Excel row numbers and column numbers start from 1.

If you’re using the Cells Property to define your range, you’ll need to enter both arguments in the Range Object (Cell1 and Cell2). If you’re using a string, you really only need one.

Although defining your range with the Cells Property is longer, you’ll find it useful when you start looping through ranges in excel (we can provide column no instead of alphabet which makes it useful).

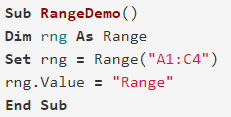
## NAMED RANGES

Let’s say we have a range named DisneyParks. You would reference the named range just like you would use string address notation.

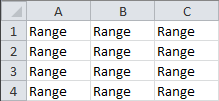


Declaring a Range

Whether for simplicity or convenience, sometimes you want to define a range in VBA.



Result:



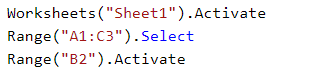
## RANGE METHODS

Below are some important methods for range object.

### ACTIVATE

VBA activate range macro code helps you to activate a range in an excel worksheet. You can use the Activate method of Range object to activate any range or one single cell in the worksheet.

It Activates a single cell, which must be inside the current selection. To select a range of cells, use the Select method.



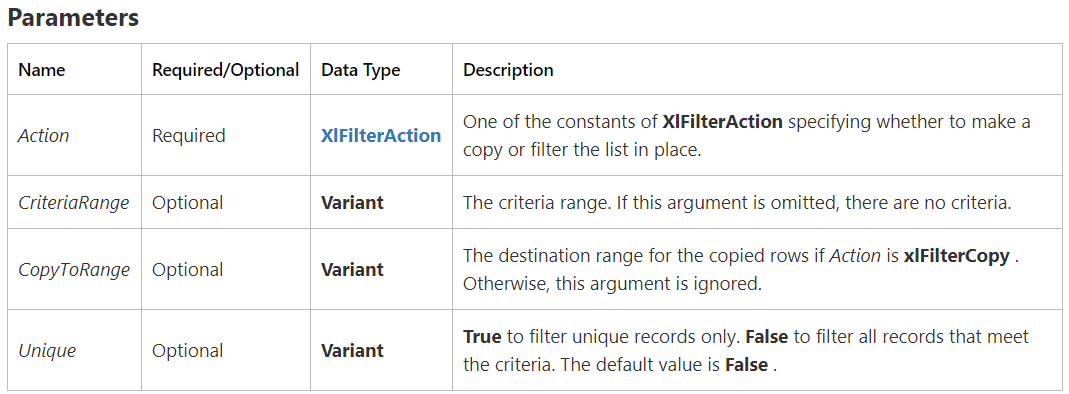
To select a cell or a range of cells use the Select method. To make a single cell the active cell with in a rabge use the Activate method.

### ADVANCEDFILTER

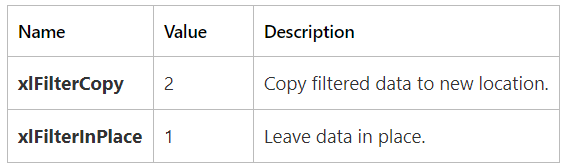
Filters or copies data from a list based on a criteria range. If the initial selection is a single cell, that cell's current region is used.

An Advanced Filter can show specific data from a list, by applying criteria. Instead of manually running an Advanced Filter, save time with a macro. Set up a criteria area and extract area, then run a macro to filter the results.

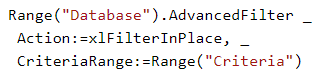




Specifies whether data is to be copied or left in place during a filter operation.



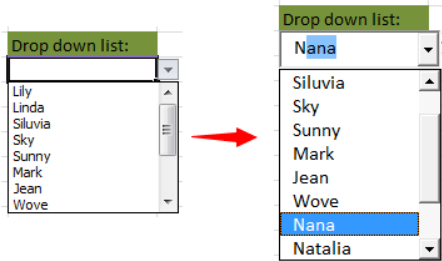
Example: This example filters a database (named "Database") based on a criteria range named "Criteria".



### AUTOCOMPLETE

Returns an AutoComplete match from the list. If there’s no AutoComplete match or if more than one entry in the list matches the string to complete, this method returns an empty string.

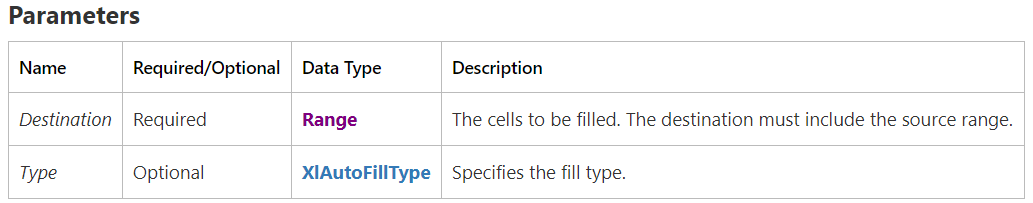
This method works even if the AutoComplete feature is disabled.



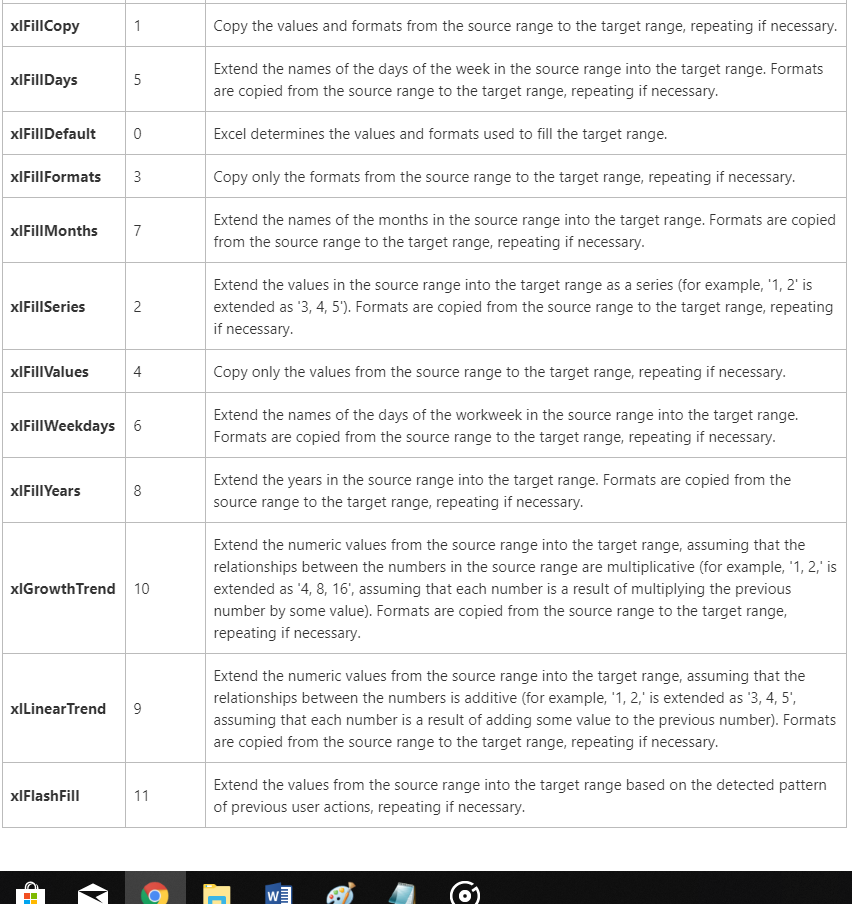
### AUTOFILL

Performs an autofill on the cells in the specified range.





Different values for XlAutoFillType are as below



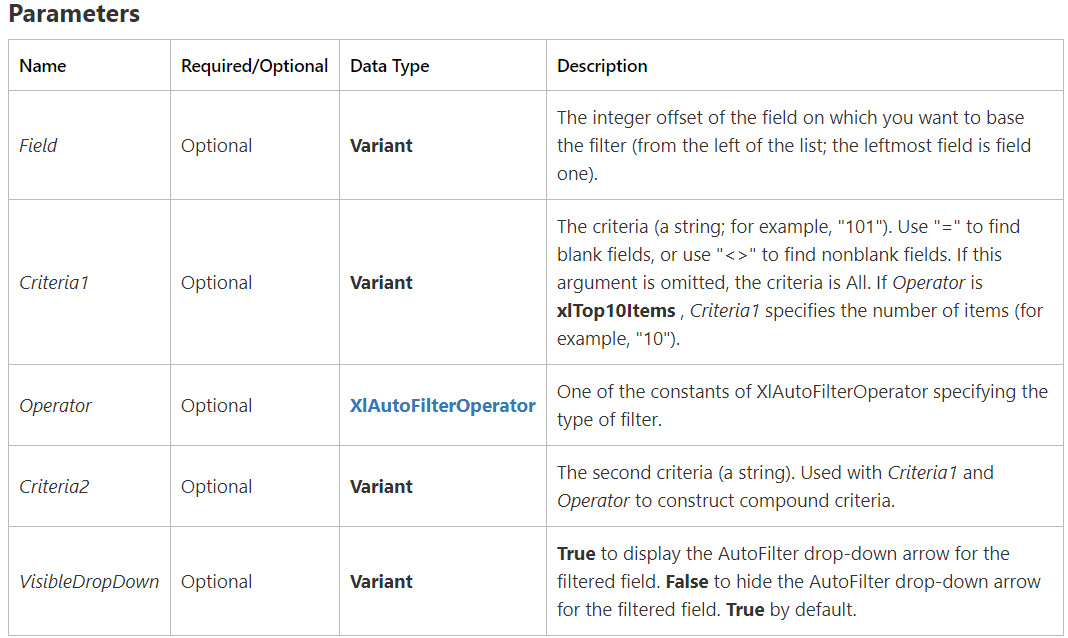
Below example performs an autofill on cells A1:A20 on Sheet1, based on the source range A1:A2 on Sheet1. Before running this example, type 1 in cell A1 and type 2 in cell A2.



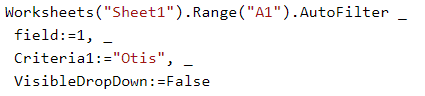
### AUTOFILTER

Filters a list using the AutoFilter.





This example filters a list starting in cell A1 on Sheet1 to display only the entries in which field one is equal to the string "Otis". The drop-down arrow for field one will be hidden.



### AUTOFIT

Changes the width of the columns in the range or the height of the rows in the range to achieve the best fit.

The Range object must be a row or a range of rows, or a column or a range of columns. Otherwise, this method generates an error.

This example changes the width of columns A through I on Sheet1 to achieve the best fit.



This example changes the width of columns A through E on Sheet1 to achieve the best fit, based only on the contents of cells A1:E1.



### CLEAR

Clears the entire object.

This example clears the formulas and formatting in cells A1:G37 on Sheet1.



### CLEARCONTENTS

Clears formulas and values from the range.

This example clears formulas and values from cells A1:G37 on Sheet1 but leaves the cell formatting and conditional formatting intact.



### CLEARFORMATS

Clears the formatting of the object.

This example clears all formatting from cells A1:G37 on Sheet1.

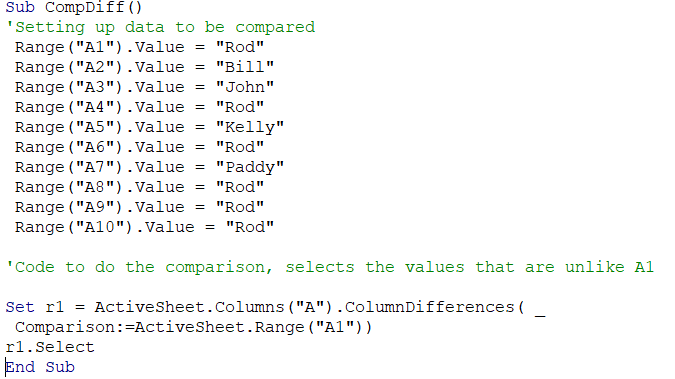


This example clears the formatting from embedded chart one on Sheet1.

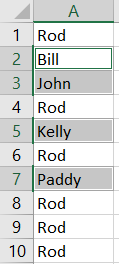


### COLUMNDIFFERENCES

Returns a Range object that represents all the cells whose contents are different from the comparison cell in each column.



Result:



### COPY

Copies the range to the specified range or to the Clipboard.



### DELETE

Deletes the object. Used only with Range objects. Specifies how to shift cells to replace deleted cells. Can be one of the following XlDeleteShiftDirection constants: xlShiftToLeft or xlShiftUp. If this argument is omitted, Microsoft Excel decides based on the shape of the range.

### FILLDOWN

Fills down from the top cell or cells in the specified range to the bottom of the range. The contents and formatting of the cell or cells in the top row of a range are copied into the rest of the rows in the range.

This example fills the range A1:A10 on Sheet1, based on the contents of cell A1.



Similarly, we have FillUp, FillLeft and Fill Right

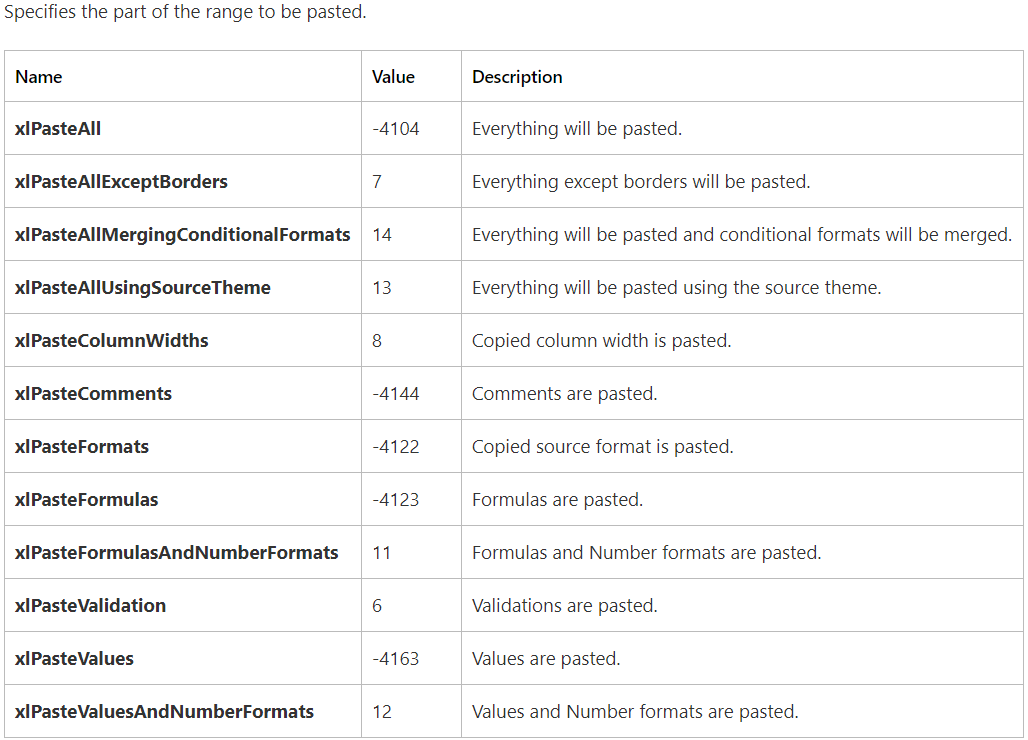
This example fills the range A1:A10 on Sheet1, based on the contents of cell A10.

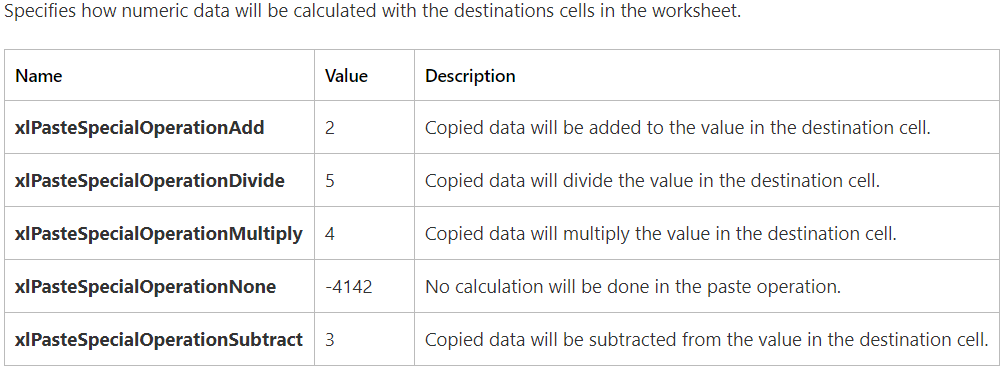


### PASTESPECIAL

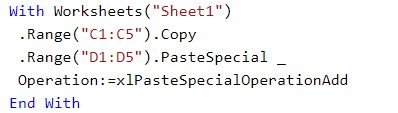
Pastes a Range that has been copied into the specified range.

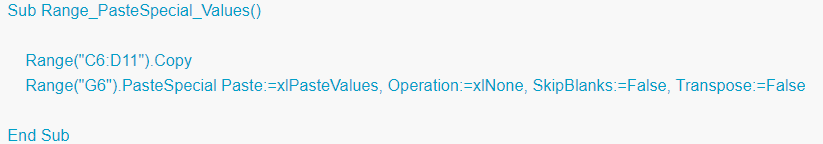






This example replaces the data in cells D1:D5 on Sheet1 with the sum of the existing contents and cells C1:C5 on Sheet1.





### REMOVEDUPLICATES

Removes duplicate values from a range of values.



The following code sample removes duplicates with the first 2 columns

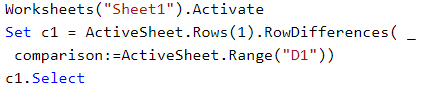


Replace

### ROWDIFFERENCES

Returns a Range object that represents all the cells whose contents are different from those of the comparison cell in each row

This example selects the cells in row one on Sheet1 whose contents are different from those of cell D1.



### SELECT

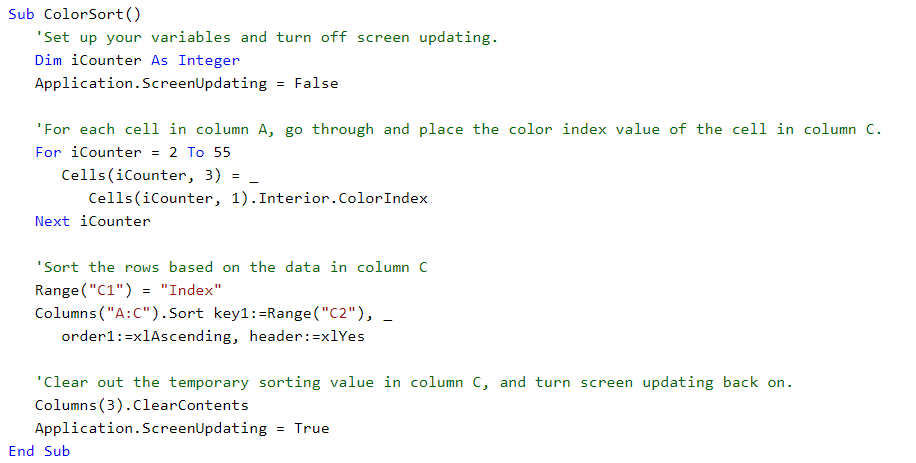
Selects the object. To select a cell or a range of cells, use the Select method. To make a single cell the active cell, use the Activate method.

### SORT

Sorts a range of values



Example:



## RANGE PROPERTIES

Below are some important properties of range object.

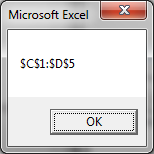
### ADDRESS

Returns a **String** value that represents the range reference in the language of the macro.

Below example show use of address property



Result:

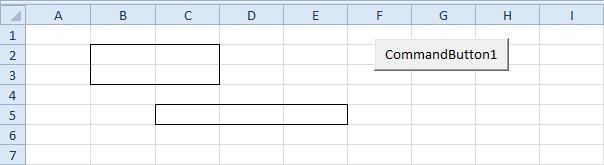


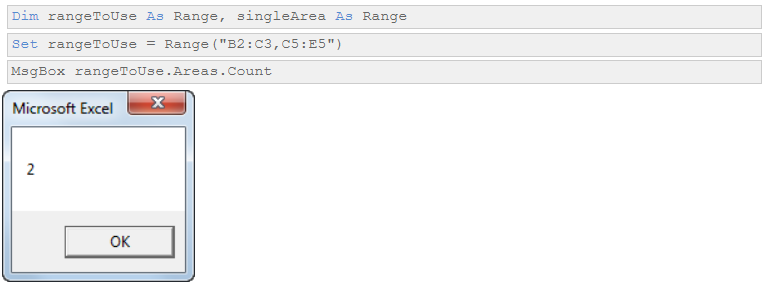
### AREAS

A collection of the areas, or contiguous blocks of cells, within a selection. Returns an Areas collection that represents all the ranges in a multiple-area selection. Read-only.

For a single selection, the Areas property returns a collection that contains one object — the original Range object itself. For a multiple-area selection, the Areas property returns a collection that contains one object for each selected area.

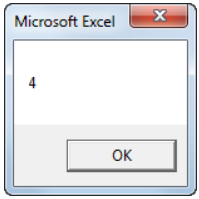
This example illustrates the Areas collection in Excel VBA. Below we have bordered Range("B2:C3,C5:E5"). This range has two areas. The comma separates the two areas.



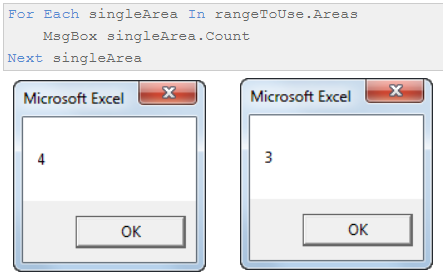


You can refer to the different areas of rangeToUse by using the index values. The following code line counts the numbers of cells of the first area.





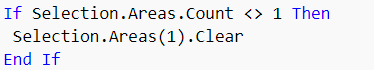
You can also loop through each area of rangeToUse and count the number of cells of each area. The macro below does the trick.



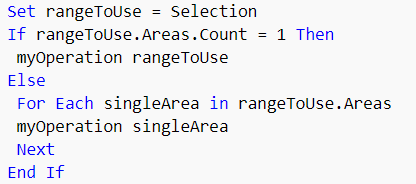
Use the Areas property to return the Areas collection. The following example clears the current selection if it contains more than one area.



Use Areas (index), where index is the area index number, to return a single Range object from the collection. The index numbers correspond to the order in which the areas were selected. The following example clears the first area in the current selection if the selection contains more than one area.



Some operations cannot be performed on more than one area in a selection at the same time; you must loop through the individual areas in the selection and perform the operations on each area separately. The following example performs the operation named "myOperation" on the selected range if the selection contains only one area; if the selection contains multiple areas, the example performs myOperation on each individual area in the selection.



### CELLS

Returns a Range object that represents the cells in the specified range.

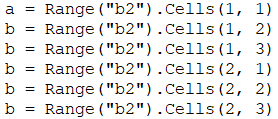
Because the Item property is the default property for the Range object, you can specify the row and column index immediately after the Cells keyword. For more information, see the Item property and the examples for this topic.

Using this property without an object qualifier returns a Range object that represents all the cells on the active worksheet.

This example sets the font style for cells A1:C5 on Sheet1 to italic.



This example shows how cells property can be used to access values considering B2 as reference point

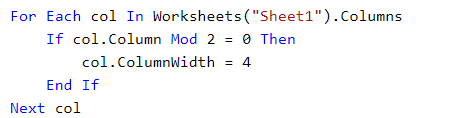


Cells (1, 1) returns the B2 cell value, Cells (1, 2) returns same row next col (i.e. C2) value. Similarly, other statements.

### COLUMN

Returns the number of the first column in the first area in the specified range. Read-only long. Column A returns 1, column B returns 2, and so on.

This example sets the column width of every other column on Sheet1 to 4 points.



### COLUMNS

Returns a Range object that represents the columns in the specified range. Using this property without an object qualifier is equivalent to using ActiveSheet.Columns.

When applied to a Range object that's a multiple-area selection, this property returns columns from only the first area of the range. For example, if the Range object has two areas — A1:B2 and C3:D4 — Selection.Columns.Count returns 2, not 4. To use this property on a range that may contain a multiple-area selection, test Areas.Count to determine whether the range contains more than one area. If it does, loop over each area in the range.

This example sets the value of every cell in column one in the range named "myRange" to 0 (zero).



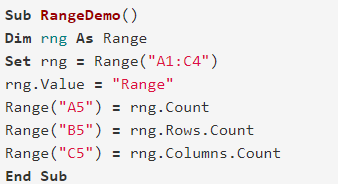
This example fills column H with a value of 0



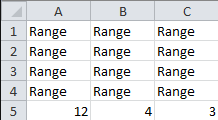
### COUNT

Returns a Long value that represents the number of objects in the collection.

Want to know how big your range is? Or more useful, how many rows and how many columns your range is?



Result:



The Range.Count Property returns the size of the complete range, 12. The Rows.Count Property returns the number of rows in the range, 4. The Columns.Count Property returns the number of columns in the range, 3.

### COUNTLARGE

Returns a value that represents the number of objects in the collection. Read-only Variant.

The CountLarge property is functionally the same as the Count property, except that the Count property will generate an overflow error if the specified range has more than 2,147,483,647 cells (one less than 2048 columns). The CountLarge property, however, can handle ranges up to the maximum size for a worksheet, which is 17,179,869,184 cells.

### CURRENTARRAY

If the specified cell is part of an array, returns a Range object that represents the entire array. Read-only.

This example assumes that cell A1 on Sheet1 is the active cell and that the active cell is part of an array that includes cells A1:A10. The example selects cells A1:A10 on Sheet1.



### CURRENTREGION

Returns a Range object that represents the current region. The current region is a range bounded by any combination of blank rows and blank columns. Read-only.

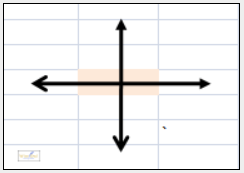
This property is useful for many operations that automatically expand the selection to include the entire current region, such as the AutoFormat method.



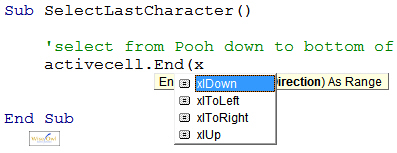
This property cannot be used on a protected worksheet.

### END PROPERTY

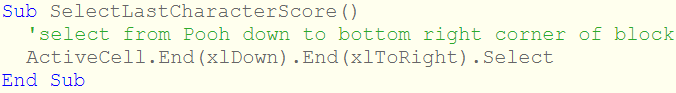
END property is used to move to end of range from current cell.

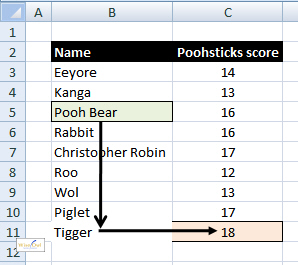


When you start typing in a command, Excel will suggest the 4 possible directions:



Below formula can be used to move to bottom right value from cell B5





Below example shows how can be break code to other line using \_ (space and an underscore)



### ENTIRECOLUMN

Returns a Range object that represents the entire column (or columns) that contains the specified range. Read-only.

This example sets the value of the first cell in the column that contains the active cell. The example must be run from a worksheet.



Below example sets the values in the ActiveCell column to “”.



### ENTIREROW

Returns a Range object that represents the entire row (or rows) that contains the specified range. Read-only.

This example sets the value of the first cell in the row that contains the active cell. The example must be run from a worksheet.



### FORMULA

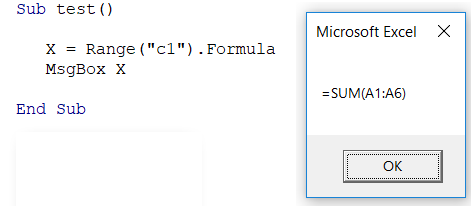
Returns or sets a Variant value that represents the object's formula in A1-style notation and in the macro language.

If the cell contains a constant, this property returns the constant. If the cell is empty, this property returns an empty string. If the cell contains a formula, the Formula property returns the formula as a string in the same format that would be displayed in the formula bar (including the equal sign (=)).

If the range is a one- or two-dimensional range, you can set the formula to a Visual Basic array of the same dimensions. Similarly, you can put the formula into a Visual Basic array.

Setting the formula for a multiple-cell range fills all cells in the range with the formula.

Example below:



The following code example sets the formula for cell A1 on Sheet1.



### FORMULAARRAY

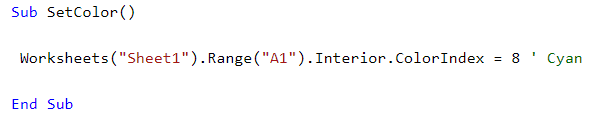
Returns or sets the array formula of a range. Returns (or can be set to) a single formula or a Visual Basic array. If the specified range doesn't contain an array formula, this property returns null.

This example enters the array formula =SUM (A1:C3) in cells E1:E3 on Sheet1.



### INTERIOR

Returns an Interior object that represents the interior of the specified object.



### ITEM

Returns a Range object that represents a range at an offset to the specified range.



The RowIndex and ColumnIndex arguments are relative offsets. In other words, specifying a RowIndex of 1 returns cells in the first row of the range, not the first row of the worksheet. For example, if the selection is cell C3, Selection.Cells (2, 2) returns cell D4 (you can use the Item property to index outside the original range).

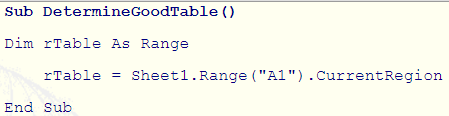
This example fills the range A1:A10 on Sheet1, based on the contents of cell A1.



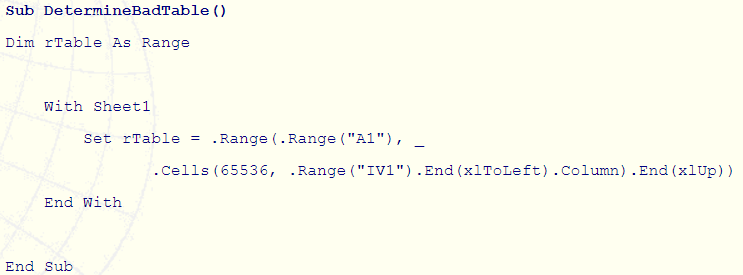
### LISTHEADERROWS

It's often that one needs to work on a range of data, or table, but NOT include any headings/headers. This can be done very easily with the ListHeaderRows Property of a Range Object.

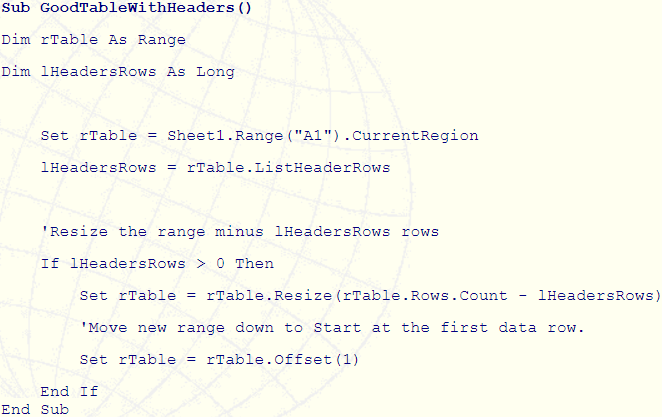
Before we can use the ListHeaderRows Property we must first determine the table range. If your table is set up correctly (no blank rows or columns) this can be done very easily via the CurrentRegion Property. That is,



However, if your table DOES include blank rows or columns, we must find the outer top left and bottom right cell. This can be done like below where we know A1 to be our top left cell of the table.

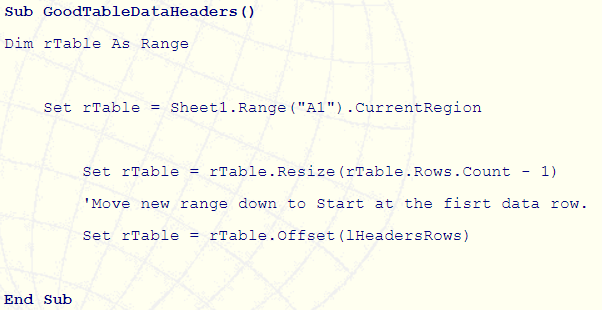


Ok, now we have determined and set a Range variable (rTable) to our table range its time to redefine the Range variable so no headers/headings are included. Here is how via the use of the ListHeaderRows Property.



If your table is numeric data and your headings are text (or vice versa), Excel will assume row 1 of the table as a header row. However, if your data AND headings are both numeric, or both text, Excel will consider your table as having NO headers. The way to overcome this is to make your headings different to that of the data. This can be done via bolding, font color/size etc.

Or, should you simply know for a fact that row 1 of the table IS a header row you can use the code below;



### LOCKED

Returns or sets a Variant value that indicates if the object is locked.

This property returns True if the object is locked, False if the object can be modified when the sheet is protected, or Null if the specified range contains both locked and unlocked cells.

This example unlocks cells A1:G37 on Sheet1 so that they can be modified when the sheet is protected.

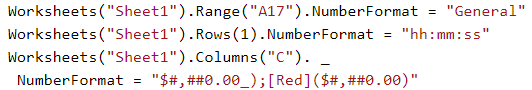


### NAME

Returns or sets a Variant value that represents the format code for the object. This property returns Null if all cells in the specified range don't have the same number format.

The format code is the same string as the Format Codes option in the Format Cells dialog box. The Format function uses different format code strings than do the NumberFormat and NumberFormatLocal properties.

These examples set the number format for cell A17, row one, and column C (respectively) on Sheet1.



### NEXT

On a protected sheet, this property returns the next unlocked cell. On an unprotected sheet, this property always returns the cell immediately to the right of the specified cell.

Returns a Range object that represents the next cell.

If the object is a range, this property emulates the TAB key, although the property returns the next cell without selecting it.

### OFFSET PROPERTY

This example assumes that Sheet1 contains a table that has a header row. The example selects the table, without selecting the header row. The active cell must be somewhere in the table before the example is run.

Returns a Range object that represents a range that’s offset from the specified range



### RANGE

When used without an object qualifier, this property is a shortcut for ActiveSheet.Range (it returns a range from the active sheet; if the active sheet isn’t a worksheet, the property fails).

When applied to a Range object, the property is relative to the Range object. For example, if the selection is cell C3, then Selection.Range("B1") returns cell D3 because it’s relative to the Range object returned by the Selection property. On the other hand, the code ActiveSheet.Range("B1") always returns cell B1.

Returns a Range object that represents a cell or a range of cells.

This example sets the value of cell A1 on Sheet1 to 3.14159.



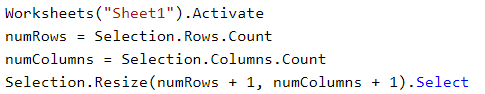
This example creates a formula in cell A1 on Sheet1.



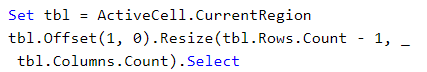
### RESIZE

Resizes the specified range. Returns a Range object that represents the resized range.

This example resizes the selection on Sheet1 to extend it by one row and one column.



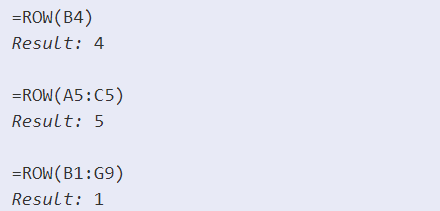
This example assumes that you have a table on Sheet1 that has a header row. The example selects the table, without selecting the header row. The active cell must be somewhere in the table before you will run the example.



### ROW

The Microsoft Excel ROW function returns the row number of a cell reference. Returns the number of the first row of the first area in the range.

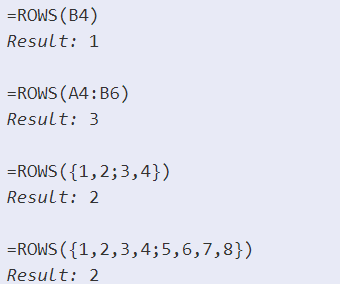
The ROW function is a built-in function in Excel that is categorized as a Lookup/Reference Function. It can be used as a worksheet function (WS) in Excel. As a worksheet function, the ROW function can be entered as part of a formula in a cell of a worksheet.



### ROWS

The Microsoft Excel ROWS function returns the number of rows in a cell reference. Returns a Range object that represents the rows in the specified range. Read-only Range object.

The ROWS function is a built-in function in Excel that is categorized as a Lookup/Reference Function. It can be used as a worksheet function (WS) in Excel. As a worksheet function, the ROWS function can be entered as part of a formula in a cell of a worksheet.



### TEXT

Returns or sets the text for the specified object. Read-only String.

### UNION PROPERTY

Use Union (range1, range2, ...) to return multiple-area ranges—that is, ranges composed of two or more contiguous blocks of cells. The following example creates an object defined as the union of ranges A1:B2 and C3:D4, and then selects the defined range.



### VALIDATION

Returns the Validation object that represents data validation for the specified range. Read-only.

This example causes data validation for cell E5 to allow blank values.

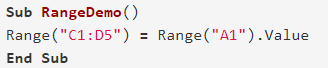


### VALUE

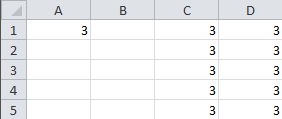
Returns or sets a Variant value that represents the value of the specified range.

Although it was optional, I used the Value Property in my Declaring a Range example, earlier. Let’s see another example. Say you have a value in one cell that you want to repeat in another range:

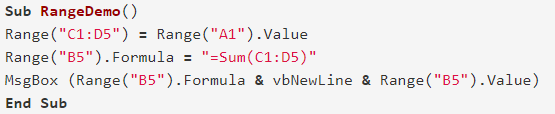
Open a worksheet and type a “3” in cell “A1.”



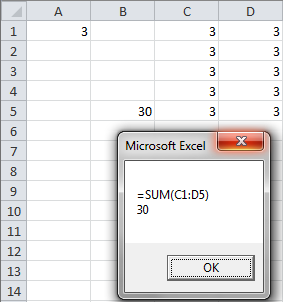
Result:



This is the default property for RANGE object. If no property or method is specified, then it takes value as default.



Result:



Or we can use Cells property as well to assign values to cell



### VALUE2

Returns or sets the cell value.

The only difference between this property and the Value property is that the Value2 property doesn't use the Currency and Date data types. You can return values formatted with these data types as floating-point numbers by using the Double data type.

### WORKSHEET

Returns a Worksheet object that represents the worksheet containing the specified range. Read-only.

This example displays the name of the worksheet that contains the active cell. The example must be run from a worksheet.



This example displays the name of the worksheet that contains the range named "testRange."



# APPLICATION OBJECT

Represents the entire Microsoft Excel application. The Application Object refers to the host application of Excel, and the entire Excel application is represented by it. If the host application is the Microsoft Word application, the Application Object refers to and represents the Word application.

The Application object is the Default Object, Excel assumes it even when it is not specified. The Application qualifier is mostly not required to be used in vba code, because the default application is Excel itself, unless you want to refer to other outside applications (like Microsoft Word or Access) in your code or you want to refer to Excel from another application like Microsoft Word. In your VBA code, both the expressions Application.ActiveWorkbook.Name and ActiveWorkbook.Name will have the same effect of returning the Active Workbook's name.

## APPLICATION PROPERTIES

Below are some important properties of application object

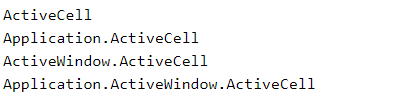
### ACTIVECELL

Returns a Range object that represents the active cell in the active window (the window on top) or in the specified window. If the window isn't displaying a worksheet, this property fails. Read-only.

If you don't specify an object qualifier, this property returns the active cell in the active window.

Be careful to distinguish between the active cell and the selection. The active cell is a single cell inside the current selection. The selection may contain more than one cell, but only one is the active cell.

The following expressions all return the active cell and are all equivalent.



This example uses a message box to display the value in the active cell. Because the ActiveCell property fails if the active sheet isn't a worksheet, the example activates Sheet1 before using the ActiveCell property.



### ACTIVECHART

Returns a Chart object that represents the active chart (either an embedded chart or a chart sheet). An embedded chart is considered active when it's either selected or activated. When no chart is active, this property returns Nothing.

This example turns on the legend for the active chart.



### ACTIVESHEET

Returns an object that represents the active sheet (the sheet on top) in the active workbook or in the specified window or workbook. Returns Nothing if no sheet is active.

If you do not specify an object qualifier, this property returns the active sheet in the active workbook. If a workbook appears in more than one window, the ActiveSheet property may be different in different windows.

This example displays the name of the active sheet.



### ACTIVEWORKBOOK

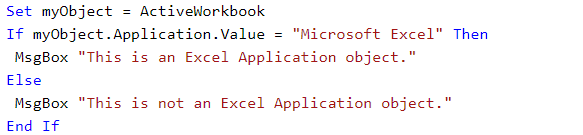
Returns a Workbook object that represents the workbook in the active window (the window on top). Read-only. Returns Nothing if there are no windows open or if either the Info window or the Clipboard window is the active window.

This example displays the name of the active workbook.



### APPLICATION

When used without an object qualifier, this property returns an Application object that represents the Microsoft Excel application. When used with an object qualifier, this property returns an Application object that represents the creator of the specified object (you can use this property with an OLE Automation object to return the application of that object). Read-only.



This example displays a message about the application that created myObject.

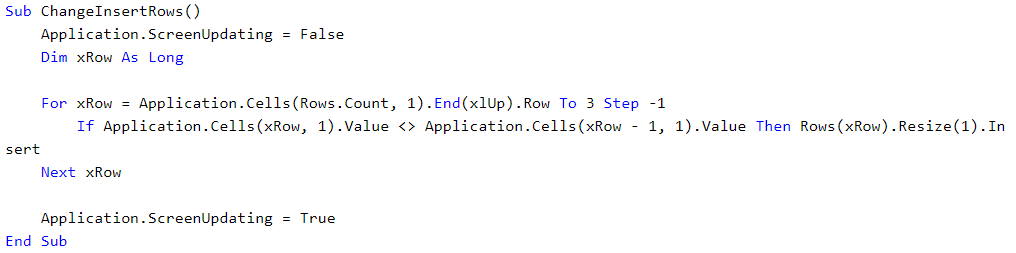
### CELLS

Returns a Range object that represents all the cells on the active worksheet. If the active document is not a worksheet, this property fails.

Because the Item property is the default property for the Range object, you can specify the row and column index immediately after the Cells keyword. For more information, see the Item property and the examples for this topic.

Using this property without an object qualifier returns a Range object that represents all the cells on the active worksheet.

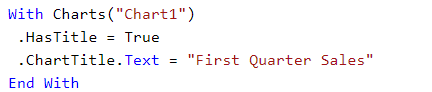
This example looks at data in each row and inserts a blank row each time the value in column A changes.



### CHARTS

Returns a Sheets collection that represents all the chart sheets in the active workbook.

This example sets the text for the title of Chart1.

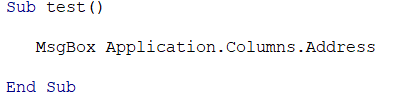


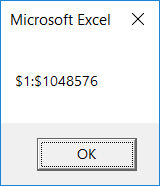
This example hides Chart1, Chart3, and Chart5.



### COLUMNS

Returns a Range object that represents all the columns on the active worksheet. If the active document isn't a worksheet, the Columns property fails.





Using this property without an object qualifier is equivalent to using ActiveSheet.Columns.

### DISPLAYALERTS

True if Microsoft Excel displays certain alerts and messages while a macro is running. Read/write Boolean.

The default value is True. Set this property to False to suppress prompts and alert messages while a macro is running; when a message requires a response, Microsoft Excel chooses the default response.

If you set this property to False, Microsoft Excel sets this property to True when the code is finished, unless you are running cross-process code.

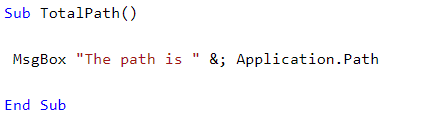
This example closes the Workbook Book1.xls and does not prompt the user to save changes. Changes to Book1.xls are not saved.



### PATH

Returns a String value that represents the complete path to the application, excluding the final separator and name of the application.

This example displays the complete path to Microsoft Excel.



### PATHSEPARATOR

Returns the path separator character ("\"). Read-only String.

This example displays the current path separator.



### ROWS

Returns a Range object that represents all the rows on the active worksheet. If the active document isn't a worksheet, the Rows property fails. Read-only Range object.

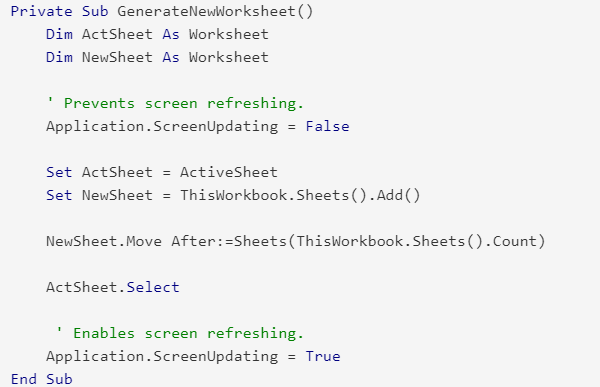
Using this property without an object qualifier is equivalent to using ActiveSheet.Rows



### SCREENUPDATING

True if screen updating is turned on. Read/write Boolean.

Turn screen updating off to speed up your macro code. You won't be able to see what the macro is doing, but it will run faster. Remember to set the ScreenUpdating property back to True when your macro ends.



### SELECTION

Returns the currently selected object in the active worksheet for an Application object. Returns Nothing if no objects are selected. Use the Select method to set the selection and use TypeName to discover the kind of object that is selected.

The returned object type depends on the current selection (for example, if a cell is selected, this property returns a Range object). The Selection property returns Nothing if nothing is selected.

Using this property with no object qualifier is equivalent to using Application.Selection

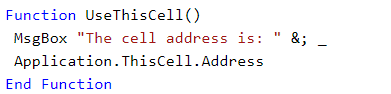


### THISCELL

Returns the cell in which the user-defined function is being called from as a Range object.

Users should not access properties or methods on the Range object when inside the user-defined function. Users can cache the Range object for later use and perform additional actions when the recalculation is finished.

In this example, a function called "UseThisCell" contains the ThisCell property to notify the user of the cell address.



### THISWORKBOOK

Returns a Workbook object that represents the workbook where the current macro code is running. Read-only.

Use this property to refer to the workbook that contains your macro code. ThisWorkbook is the only way to refer to an add-in workbook from inside the add-in itself. The ActiveWorkbook property doesn't return the add-in workbook; it returns the workbook that's calling the add-in.

The Workbooks property may fail, as the workbook name probably changed when you created the add-in. ThisWorkbook always returns the workbook in which the code is running.

For example, use code such as the following to activate a dialog sheet stored in your add-in workbook.



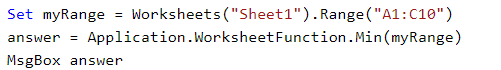
This example closes the workbook that contains the example code. Changes to the workbook, if any, aren't saved.



### WORKSHEETFUNCTION

Returns the WorksheetFunction object.

This example displays the result of applying the Min worksheet function to the range A1:A10.



# WORKBOOKS OBJECT

A collection of all the Workbook objects that are currently open in the Microsoft Excel application.

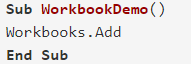
The Workbook Object appears next below the Application Object in Excel VBA object hierarchy and represents a single workbook within the Excel application. The Workbooks Object refers to a collection of all currently open Workbooks (i.e. all currently open Workbook Objects) in Excel.

## WORKBOOKS METHODS

Below are some important methods for workbooks object.

### ADD

Creates a new workbook. The new workbook becomes the active workbook. The Workbooks.Add Method creates a new workbook and sets it to the ActiveWorkbook.



Because the new Excel File is the ActiveWorkbook, you can manipulate the new file with the ActiveWorkbook Property.

### CLOSE

Closes the object.

### OPEN

Opens a workbook.

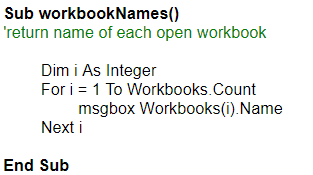
By default, macros are enabled when opening files programmatically. Use the AutomationSecurity property to set the macro security mode used when opening files programmatically.

## WORKBOOKS PROPERTIES

Below are some important properties of workbooks objects.

### COUNT

The Count property of the Workbooks object - WorkbooksObject.Count Property - returns the number of workbooks in the collection.



### ITEM

The Item Property of the Workbooks object (Workbooks.Item property) refers to a single workbook in a collection.

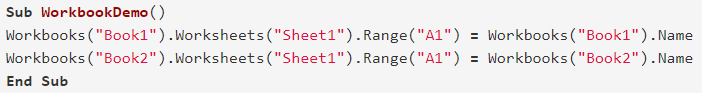
# WORKBOOK

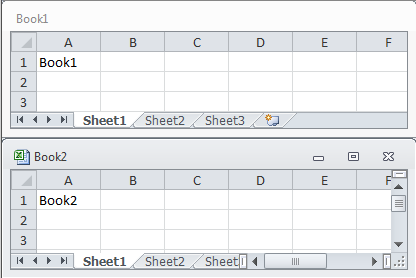
The Workbook object is a member of the Workbooks collection. The Workbooks collection contains all the Workbook objects currently open in Microsoft Excel.



The Workbook Object is fancy way of referring to your Excel file. The Workbook Object is a part of the Workbooks Collection. Because it’s a part of a collection, you’ll see it written plural in VBA.

Let’s see an example. Open two new workbooks, Book1 and Book2, in Excel and run the following macro.





This macro writes the workbook name to “Sheet1” and cell “A1” of each open workbook. The ability to call different workbooks by name allows you to interact with multiple workbooks (Excel files) from a single VBA macro.

The ThisWorkbook property returns the workbook where the Visual Basic code is running. In most cases, this is the same as the active workbook. However, if the Visual Basic code is part of an add-in, the ThisWorkbook property won't return the active workbook. In this case, the active workbook is the workbook calling the add-in, whereas the ThisWorkbook property returns the add-in workbook.

Use Workbooks (index), where index is the workbook name or index number, to return a single Workbook object. The following example activates workbook one.



The index number denotes the order in which the workbooks were opened or created. Workbooks (1) is the first workbook created, and Workbooks (Workbooks.Count) is the last one created. Activating a workbook doesn't change its index number. All workbooks are included in the index count, even if they're hidden.

The Name property returns the workbook name. You cannot set the name by using this property; if you need to change the name, use the SaveAs method to save the workbook under a different name. The following example activates Sheet1 in the workbook named Cogs.xls (the workbook must already be open in Microsoft Excel).



The ActiveWorkbook property returns the workbook that's currently active. The following example sets the name of the author for the active workbook.



## WORKBOOK EVENTS

An Event occurs when certain actions are performed by the user or VBA. For example, when you open a workbook the Workbook\_Open event is triggered. When you (or your VBA) saves a workbook, the Workbook\_BeforeSave event is triggered.

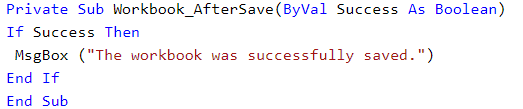
These events are built into Excel VBA and are executed automatically by Excel. You can write your own code which is also executed when these events occur. This code is known as an Event Handler.

### ACTIVATE

Occurs when a workbook, worksheet, chart sheet, or embedded chart is activated.

### AFTERSAVE

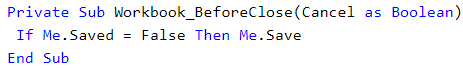
Occurs after the workbook is saved.



### BEFORECLOSE

Occurs before the workbook closes. If the workbook has been changed, this event occurs before the user is asked to save changes.

This example always saves the workbook if it's been changed.



### NEWCHART

Occurs when a new chart is created in the workbook.

The NewChart event occurs whenever a new chart is inserted or pasted in a worksheet, a chart sheet, or other sheet types. If multiple charts are inserted or pasted, the event will occur for each chart in the order they are inserted or pasted. If a chart object or chart sheet is moved from one location to another, the event will not occur. However, if the chart is moved between a chart object and a chart sheet, the event will occur because a new chart must be created.

The NewChart event will not occur in the following scenarios: copying or pasting a chart sheet, changing a chart type, changing a chart data source, undoing or redoing inserting or pasting a chart, and loading a workbook that contains a chart.

### NEWSHEET

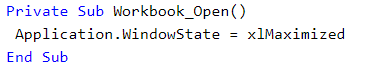
Occurs when a new sheet is created in the workbook.



This example moves new sheets to the end of the workbook.

### OPEN

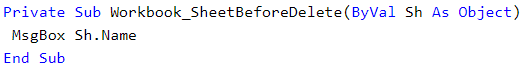
Occurs when the workbook is opened.



This example maximizes Microsoft Excel whenever the workbook is opened.

### SHEETBEFOREDELETE

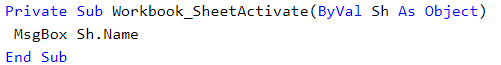
Occurs when any sheet is deleted.



This example displays the name of each deactivated sheet.

### SHEETACTIVATE

Occurs when any sheet is activated.



This example displays the name of each activated sheet.

## WORKBOOK METHODS

Below are some important methods for workbook object.

### ACTIVATE

Activates the first window associated with the workbook.

This example activates Book4.xls. If Book4.xls has multiple windows, the example activates the first window, Book4.xls:1.



### CLOSE

Closes the object.





This example closes Book1.xls and discards any changes that have been made to it.

### PROTECT

Protects a workbook so that it cannot be modified.

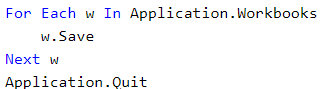
### SAVE

Saves changes to the specified workbook.

This example saves the active workbook.



This example saves all open workbooks and then closes Microsoft Excel.

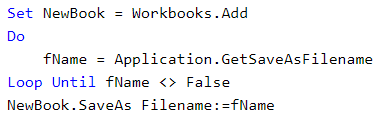


### SAVEAS

Saves changes to the workbook in a different file or for saving a newly created workbook. The first time you save a workbook, use the SaveAs method to specify a name for the file.



This example creates a new workbook, prompts the user for a file name, and then saves the workbook.



### SAVECOPYAS

Saves a copy of the workbook to a file but doesn't modify the open workbook in memory.



This example saves a copy of the active workbook.



### UNPROTECT

Removes protection from a sheet or workbook. This method has no effect if the sheet or workbook isn't protected.



If you forget the password, you cannot unprotect the sheet or workbook. It's a good idea to keep a list of your passwords and their corresponding document names in a safe place.

## WORKBOOK PROPERTIES

Below are some important properties of workbook object.

### ACTIVESHEET

Returns an object that represents the active sheet (the sheet on top) in the active workbook or in the specified window or workbook. Returns Nothing if no sheet is active.

If you don't specify an object qualifier, this property returns the active sheet in the active workbook.

If a workbook appears in more than one window, the ActiveSheet property may be different in different windows.



This example displays the name of the active sheet.

### CHART

Returns a Sheets collection that represents all the chart sheets in the specified workbook.



This example deletes every chart sheet in the active workbook.

### NAME

Returns a **String** value that represents the name of the object.

### NAMES

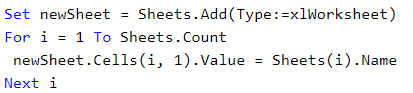
Returns a [Names](https://msdn.microsoft.com/en-us/VBA/Excel-VBA/articles/names-object-excel) collection that represents all the names in the specified workbook (including all worksheet-specific names). Read-only Names object.

Using this property without an object qualifier is equivalent to using ActiveWorkbook.Names

### SHEETS

Returns a Sheets collection that represents all the sheets in the specified workbook. Read-only Sheets object.

Using this property without an object qualifier is equivalent to using ActiveWorkbook.Sheets

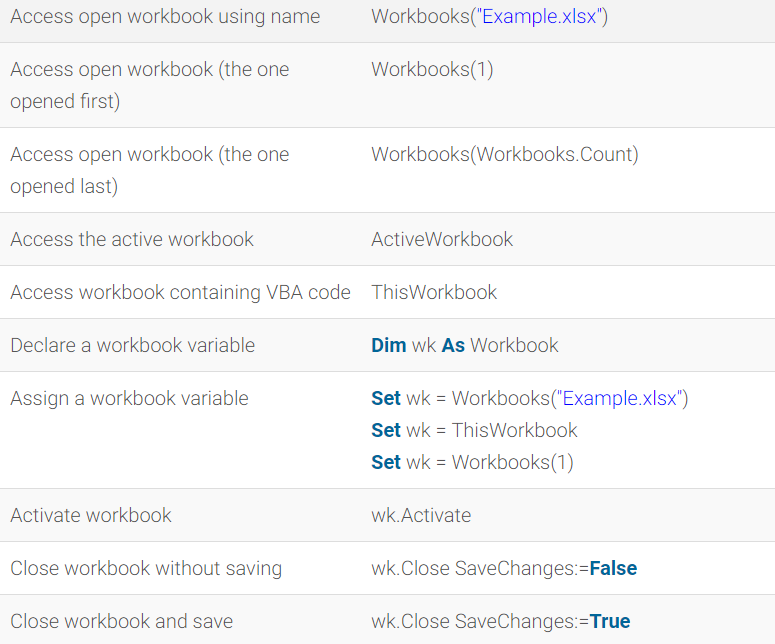


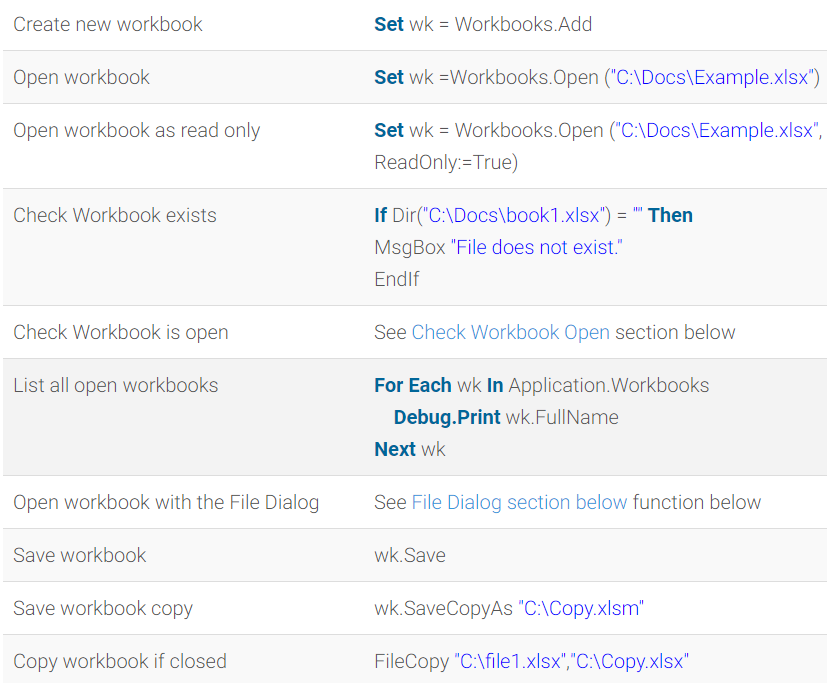
### WORKSHEETS

Returns a Sheets collection that represents all the worksheets in the specified workbook. Read-only Sheets object.

Using this property without an object qualifier returns all the worksheets in the active workbook.

This property doesn't return macro sheets; use the Excel4MacroSheets property or the Excel4IntlMacroSheets property to return those sheets.





# SHEETS VS WORKSHEETS

The SHEETS object is a parent object for:

* Worksheets
* Chart Sheets

If a workbook has 3 worksheets and 1 chart sheet, in VBA:

* Sheets.Count will include both types. 4
* Worksheets.Count will include only worksheets. 3

# SHEETS

A collection of all the sheets in the specified or active workbook. The Sheets collection can contain Chart or Worksheet objects.

The Sheets collection is useful when you want to return sheets of any type. If you need to work with sheets of only one type, see the object topic for that sheet type.

Use the Sheets property to return the Sheets collection. The following example prints all sheets in the active workbook.



Use the Add method to create a new sheet and add it to the collection. The following example adds two chart sheets to the active workbook, placing them after sheet two in the workbook.



Use Sheets (array) to specify more than one sheet. The following example moves the sheets named "Sheet4" and "Sheet5" to the beginning of the workbook.



Worksheets objects are a collection of all the Worksheet objects in the specified or active workbook. Each Worksheet object represents a worksheet.

The methods and properties for Sheets and Worksheets are same. Here we will discuss for sheets.

## SHEETS METHODS

Below are some important methods for sheet object.

### ADD

Creates a new worksheet, chart, or macro sheet. The new worksheet becomes the active sheet.

If Before and After are both omitted, the new sheet is inserted before the active sheet.



This example inserts a new worksheet before the last worksheet in the active workbook.



Sheets properties

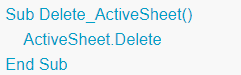
### ADD2

This method is only implemented for the **Charts** collection object and will produce a run time error if used on the **Sheets** and **Worksheets** objects.

### DELETE

Deletes the object.

Delete Worksheet method in VBA is used to delete the sheet from the Excel workbook. When we delete a worksheet, delete worksheet method displays a dialog box that reminds the user to confirm the deletion by default. If we click on Delete button on the dialog box then it deletes the worksheet from a workbook. It has Boolean value. That means dialog box will appear with two options. That is either Cancel (False) or Delete (True).





### MOVE

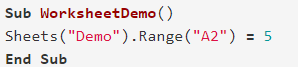
Moves the sheet to another location in the workbook.

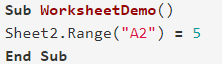
If you don't specify either Before or After, Microsoft Excel creates a new workbook that contains the moved sheet.



### SELECT

Selects the object.





## SHEETS PROPERTIES

Below are some important properties of sheets object.

### COUNT

Returns a Long value that represents the number of objects in the collection.

### ITEM

Returns a single object from a collection.



### ACTIVATE

Makes the current sheet the active sheet. Calling this method is equivalent to clicking the sheet's tab.

This example activates Sheet1.



### CHARTOBJECTS

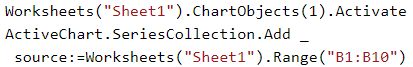
Returns an object that represents either a single embedded chart (a ChartObject object) or a collection of all the embedded charts (a ChartObject object) on the sheet.

This method isn't equivalent to the Charts property. This method returns embedded charts; the Charts property returns chart sheets. Use the Chart property to return the Chart object for an embedded chart.

This example adds a title to embedded chart one on Sheet1.



This example creates a new series in embedded chart one on Sheet1. The data source for the new series is the range B1:B10 on Sheet1.



This example clears the formatting of embedded chart one on Sheet1.



# WORKSHEET

The Worksheet object is a member of the Worksheets collection. The Worksheets collection contains all the Worksheet objects in a workbook.

The Worksheet object is also a member of the Sheets collection. The Sheets collection contains all the sheets in the workbook (both chart sheets and worksheets).

Use Worksheets (index), where index is the worksheet index number or name, to return a single Worksheet object. The following example hides worksheet one in the active workbook.

The worksheet index number denotes the position of the worksheet on the workbook's tab bar. Worksheets (1) is the first (leftmost) worksheet in the workbook, and Worksheets (Worksheets.Count) is the last one. All worksheets are included in the index count, even if they're hidden.

### WORKSHEET METHODS

Some important methods for worksheet object

### ACTIVATE

Makes the current sheet the active sheet.

Calling this method is equivalent to clicking the sheet's tab.

This example activates Sheet1.



### COPY

Copies the sheet to another location in the current workbook or a new workbook.

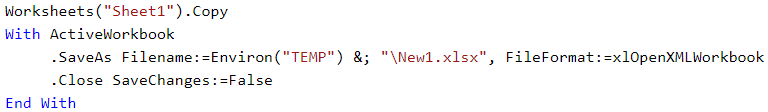
If you don't specify either Before or After, Microsoft Excel creates a new workbook that contains the copied sheet object that contains the copied Worksheet object. The newly created workbook holds the Application.ActiveWorkbook Property (Excel) property and contains a single worksheet. The single worksheet retains the Worksheet.Name Property (Excel) and Worksheet.CodeName Property (Excel) properties of the source worksheet. If the copied worksheet held a worksheet code sheet in a VBA project, that is also carried into the new workbook.

An array selection of multiple worksheets can be copied to a new blank Workbook Object (Excel) object in a similar manner.

This example copies Sheet1, placing the copy after Sheet3.



This example first copies Sheet1 to a new blank workbook, then saves and closes the new workbook.



### DELETE

Deletes the object.

When you delete a Worksheet, this method displays a dialog box that prompts the user to confirm the deletion. This dialog box is displayed by default. When called on the Worksheet object, the Delete method returns a Boolean value that is False if the user clicked Cancel on the dialog box or True if the user clicked Delete.

### MOVE

Moves the sheet to another location in the workbook.

If you don't specify either Before or After, Microsoft Excel creates a new workbook that contains the moved sheet.



### SAVEAS

Saves changes to the chart or worksheet in a different file.

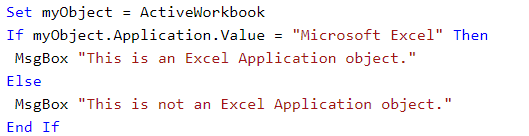
## WORKBOOK PROPERTIES

Some important properties of workbook object.

### APPLICATION

When used without an object qualifier, this property returns an Application object that represents the Microsoft Excel application. When used with an object qualifier, this property returns an Application object that represents the creator of the specified object (you can use this property with an OLE Automation object to return the application of that object). Read-only.

This example displays a message about the application that created myObject.



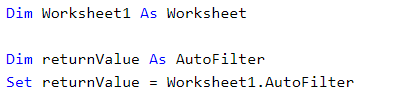
### AUTOFILTER

Returns an AutoFilter object if filtering is on. Read-only.

The property returns Nothing if filtering is off.

To create an AutoFilter object for a worksheet, you must turn autofiltering on for a range on the worksheet either manually or using the AutoFilter method of the Range object.

The following example returns autofilter for the current worksheet.

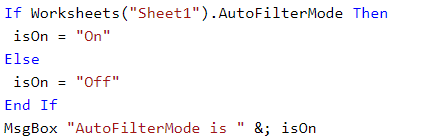


### AUTOFILTERMODE

True if the AutoFilter drop-down arrows are currently displayed on the sheet. This property is independent of the FilterMode property. Read/write Boolean.

This property returns True if the drop-down arrows are currently displayed. You can set this property to False to remove the arrows, but you cannot set it to True. Use the AutoFilter method to filter a list and display the drop-down arrows.

This example displays the current status of the AutoFilterMode property on Sheet1.



### CELLS

Returns a Range object that represents all the cells on the worksheet (not just the cells that are currently in use).

This example sets the font size for cell C5 on Sheet1 to 14 points.



This example clears the formula in cell one on Sheet1.



### COLUMNS

Returns a Range object that represents all the columns on the active worksheet. If the active document isn't a worksheet, the Columns property fails.

Using this property without an object qualifier is equivalent to using ActiveSheet.Columns.

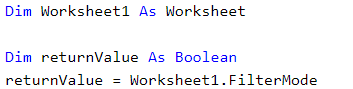
This example formats the font of column one (column A) on Sheet1 as bold.



### FILTERMODE

True if the worksheet is in the filter mode. Read-only Boolean.

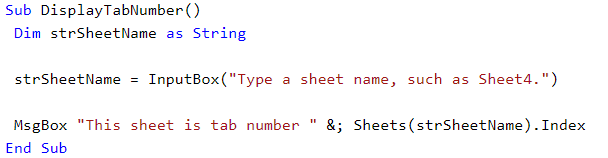
In the following example, the code returns True if the worksheet is in the filter mode.



### INDEX

Returns a Long value that represents the index number of the object within the collection of similar objects.

This example displays the tab number of the sheet specified by the name that you type. For example, if Sheet4 is the third tab in the active workbook, the example displays "3" in a message box.



### NAME

Returns or sets a **String** value that represents the object name.

### NAMES

Returns a Names collection that represents all the worksheet-specific names (names defined with the "WorksheetName!" prefix).

### RANGE

Returns a Range object that represents a cell or a range of cells.

### ROWS

Returns a Range object that represents all the rows on the specified worksheet. Read-only Range object.

### SORT

Returns a Sort object. Read-only.

### USEDRANGE

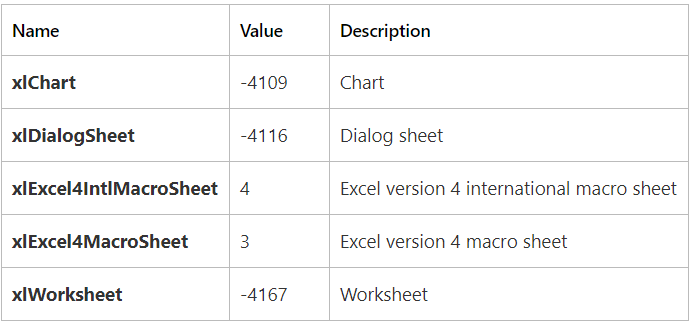
Returns a Range object that represents the used range on the specified worksheet. Read-only.

This example selects the used range on Sheet1.



### TYPE

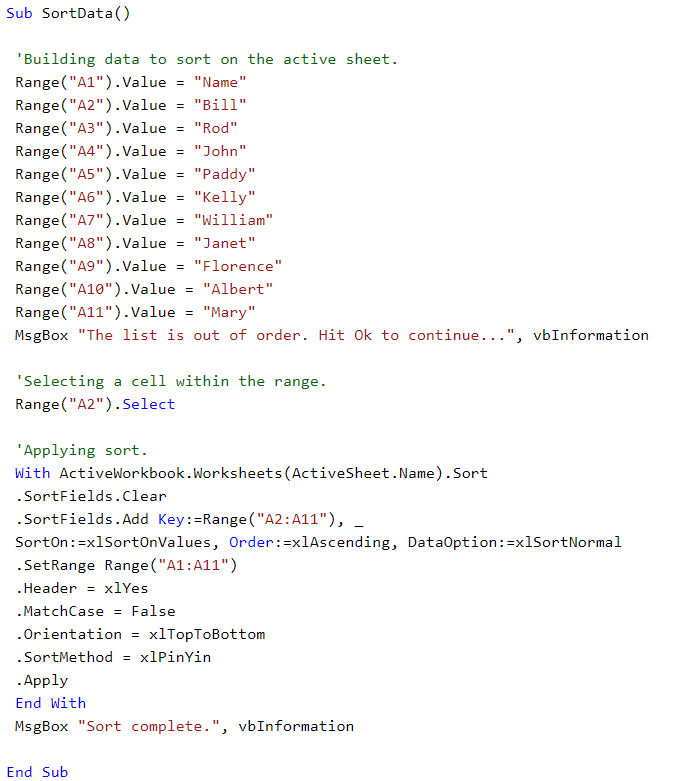
Returns an XlSheetType value that represents the worksheet type. Specifies the worksheet type.



# SORT OBJECT

Represents a sort of a range of data.

The following procedure builds and sorts data in a range in the active worksheet.



## METHODS

Some important methods.

### APPLY

Sorts the range based on the currently applied sort states.

### SETRANGE

Sets the range over which the sort occurs.

Note: SetRange can only be used when applying a sort to a sheet range and cannot be used if the range is within a table.

## PROPERTIES

Some important properties.

### HEADER

Specifies whether the first row contains header information.

xlNo is the default value. You can specify xlGuess if you want Excel to attempt to determine the header. XlYes if the range has a header.

### MATCHCASE

Set to True to perform a case-sensitive sort or set to False to perform non-case sensitive sort. Read/write.

### RNG

Return the range of values on which the sort is performed. Read-only.

### SORTFIELD

Returns the SortFields object that represents the collection of sort fields associated with the Sort object. Read-only.

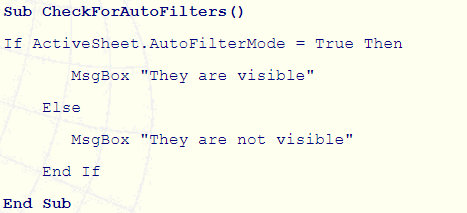
# FILTERING VBA

When using filtering in VBA

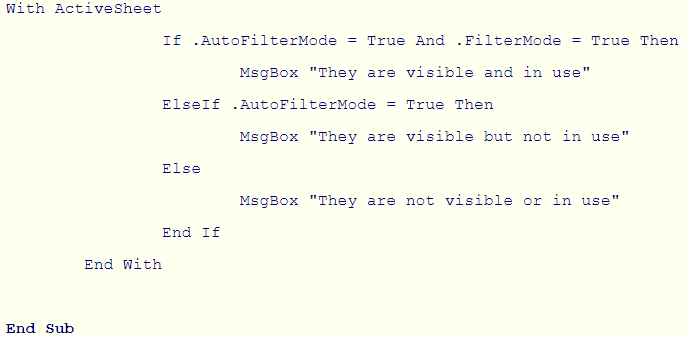
* Autofilter property is used to filter data
* AutoFilterMode tells whether filter arrows (drop down lists) are visible on the selected range
* FilterMode tells us whether filter has been applied onto the range or not using criteria(s).

AutoFilter provides us with a MUCH faster alternative to loops of all kinds. In majority of the cases it's faster and more efficient to use one of Excel's built in features as opposed to re-inventing the wheel with VBA code.

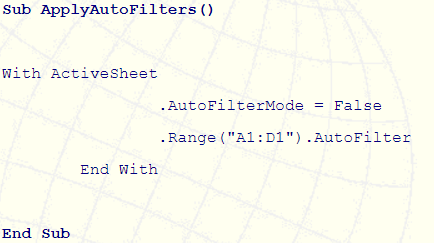
Ok, the first thing we need to know is how to apply AutoFilter to a range. When we do apply AutoFilter via VBA one SHOULD always turn the off any current filters and remove them completely. Why not check if the AutoFilter is already in place and go from there? The answer is simple, while we can determine if AutoFilter has/is on a specific Worksheet, we cannot guarantee (with extra checking) that it is in use on the range we need! For example, we could use the code below to check.



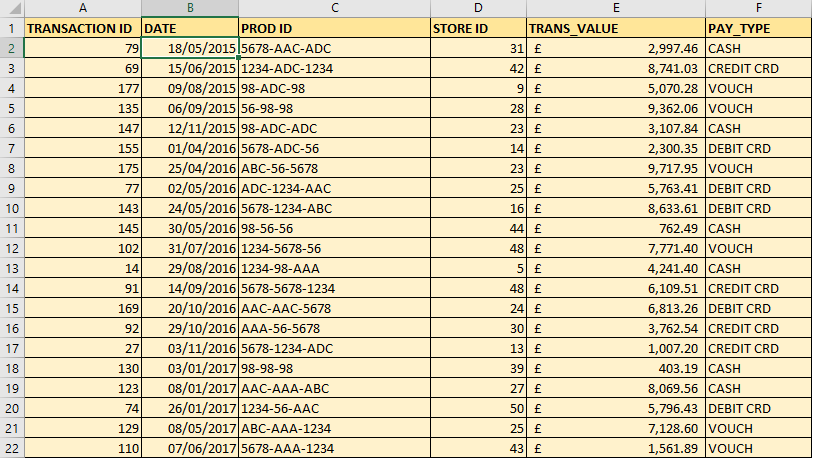
From the code above, we will know if AutoFilters are visible, but not necessarily in Filter mode (more on that soon). However, we cannot tell if the AutoFilterMode is applied to the correct range. Let's now see how we can determine if the AutoFilters and in use and are being used to filter down.



As you can see, we have used the FilterMode Property of the Worksheet to determine whether the AutoFilters are filtering data down. So, in summary, AutoFilterMode tells us if the AutoFilter arrows are visible and FilterMode tells us if they are in use. However, as I mentioned above this does not tell us which range has had AutoFilter applied. So with this in mind we are better off simply removing any existing Autofilter and then applying them to our required range. Here is how, assuming we want A1:D1 to have the AutoFilters.



Example:



## FILTER ON A SINGLE CRITERIA

This code would filter the data so that only cash payments were displayed.



## FILTER ON TWO “OR” CRITERIA

Use the xlOr operator to perform “Or” criteria.



## FILTER ON MORE THAN TWO “OR” CRITERIA USING ARRAY



You must use the xlFilterValues operator when using an Array as your criteria.

## FILTER ON TWO “AND” CRITERIA

Use the xlAnd operator to perform “And” criteria.



## FILTER ON TOP/BOTTOM X VALUES



Use the Criteria parameter to specify the number of records to return.

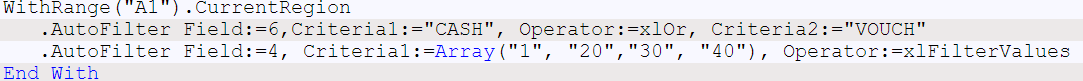
## Perform Dynamic Date Filters



## PERFORM WILDCARD FILTERS ON TEXT FIELDS



You can use the \* and? wildcard characters in the usual way.



# PUBLIC VS PRIVATE

Private and Public are mostly used to either declare the scope of a variable or a subroutine (sub). You may also see the word “Dim” used to declare a variable. You can think of Dim as another way of stating Private however, there is a time and a place to use each one. I will touch on how to determine which word to use in the following sections.

## PRIVATE

Private Sub sets the scope so that subs in outside modules cannot call that particular subroutine. This means that a sub in Module 1 could not use the Call method to initiate a Private Sub in Module 2. (Note: If you start at the Application level, you can use Run to override this rule and access a Private Sub)

Private [insert variable name] means that the variable cannot be accessed or used by subroutines in other modules. To use, these variables must be declared outside of a subroutine (usually at the very top of your module). You can use this type of variable when you have one subroutine generating a value and you want to pass that value on to another subroutine in the same module.

Dim [insert variable name] is used to state the scope inside of a subroutine (you cannot use Private in its place). Dim can be used either inside a subroutine or outside a subroutine (using it outside a subroutine would be the same as using Private).

## PUBLIC

Public Sub means that your subroutine can be called or triggered by other subs in different modules. Public is the default scope for all subs so you do not need to add it before the word “sub”. However, it does provide further clarity to others who may be reading your code. As a personal preference I do not type Public Sub unless I am creating an intricate program that has a bunch of subroutines with varying scopes (i.e. I have a mix of Public & Private subs).

Public [insert variable name] means that the variable can be accessed or used by subroutines in outside modules. These variables must be declared outside of a subroutine (usually at the very top of your module). You can use this type of variable when you have one subroutine generating a value and you want to pass that value on to another subroutine stored in a separate module.

# VARIABLES

A variable is a placeholder which reserves a spot in your computer’s memory, so you can use it later.

When you declare a variable, you are reserving a spot in your computer memory for a specific data type with a specific name. VBA variable declarations can technically be placed anywhere within your macro, but convention states they should be placed immediately after your opening. Generically, variable declarations follow the format:

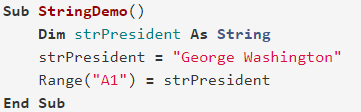


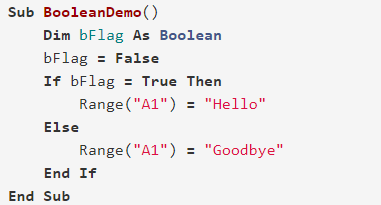
## NAMING CONVENTIONS

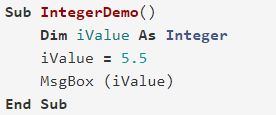
There is no “correct” naming convention for VBA variables. You can make up your own convention but be sure to use it consistently. A solid naming convention makes it easier for you and others to read your macros. You don’t have to keep scrolling around to try to find out what data type your variable is supposed to be. Here’s my naming convention for the most common data types:

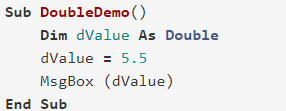
* String variable names begin with str (strEmployee)
* Boolean variable names begin with b (bFlag)
* Integer variable names begin with i (iCount)
* Double variable names begin with d (dSalary)
* Long variable names begin with l (lCount)
* Range object begins with rng (rngName)
* Workbook object begins with wbk (wkName)

Examples:





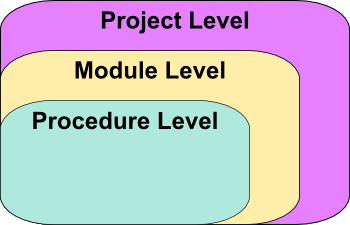




## SCOPE

For a given workbook, there are 3 levels of scope, defined below from least accessible to most accessible.

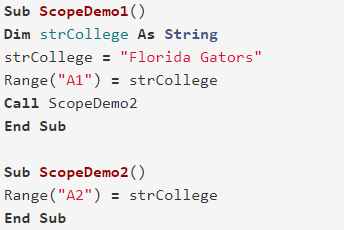
* Procedure Level
* Module Level
* Project Level



### PROCEDURE LEVEL

If you’re writing quick scripts, you’ll most likely use Procedure Level variables. As a matter of fact, Procedure Level variables are the only type of variable we have used thus far in our Excel tutorials.

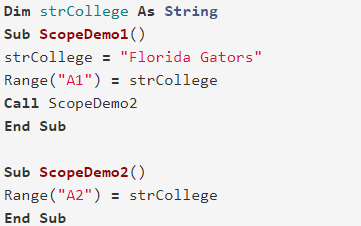
A Procedure Level variable is only visible (i.e., you can only use it) within the procedure it’s declared in. Recall, a procedure can be a Sub or a Function.



VBA Procedure Level

### MODULE LEVEL

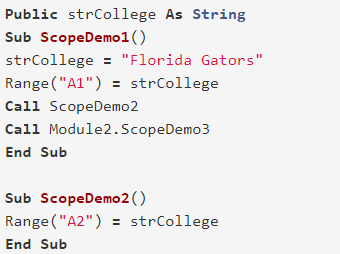
A Module Level variable, also called a Private Module Level variable, can be used by any procedure within the same module. It must be declared outside of your procedures, at the top of your module.

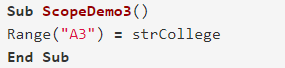


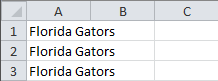
VBA Module Level

### PROJECT LEVEL

Project Level is sometimes referred to as Public Module Level. Variables declared as Public are accessible by all the modules in the vba project.

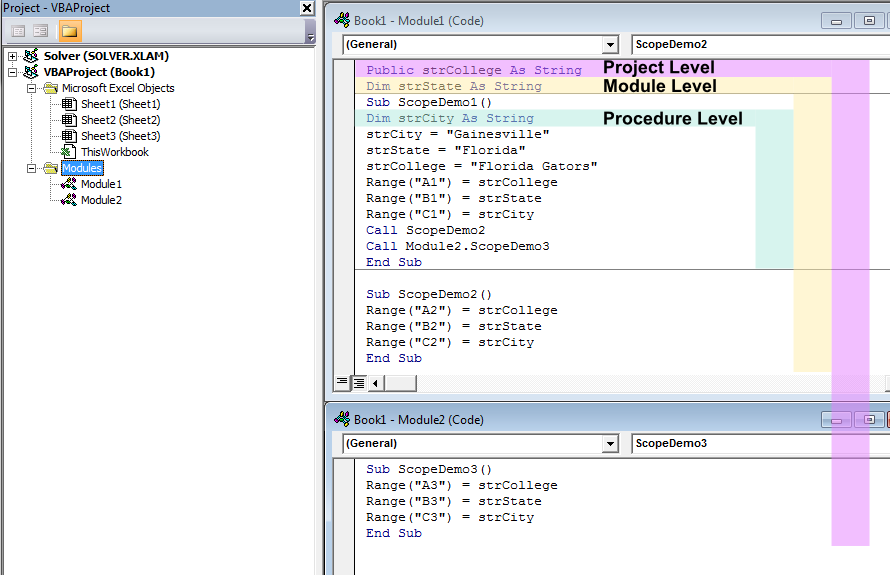


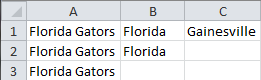




The only difference between the Module Level and the Project Level code is that the keyword Dim in the Variable Declaration is replaced with the keyword Public. That’s why this type of variable is often called a Public Module Level variable. The resulting variable is available to all procedures across all modules in the workbook.

The following color-coded graphic puts it all together:

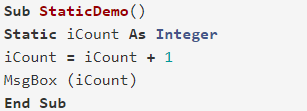




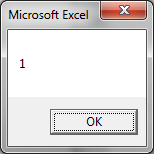
## LIFETIME OF A VARIABLE

The Lifetime of a variable is how long a variable can retain its value. A variable only retains its value for as long as it has scope. You can extend the scope, and thus the lifetime, by declaring your variable with the Static keyword.

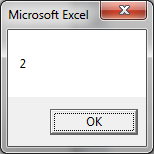
If you use the Static keyword to declare a variable, the variable will retain its value even after all macros are finished! This is extremely handy for many applications, including codes requiring initial user setup or custom undo/redo buttons. To try a Static declaration, copy the example below and run your macro twice.



Result after 1st run



Result after 2nd run

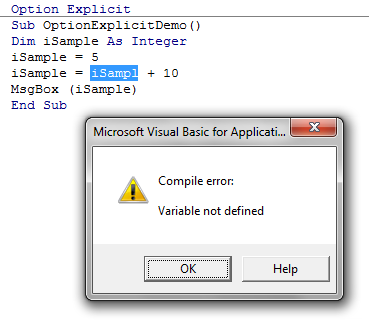


The Static Variable iCount retains its value after each execution, so each time you run your macro, iCount will increase by 1. If you were to declare iCount with the Dim statement, the result would always be a message box with a printed value of “1.” Go ahead, test it out.

## OPTION EXPLICIT

When Option Explicit is enabled, you are required to declare all your variables. If we don’t enable OPTION Explicit, then any variable used but not declared is declared by the vba as variant data type.

To enable Option Explicit, simply type Option Explicit at the very top of your Visual Basic Editor. It must be placed above your first procedure (Sub or Function).



Enabling Option Explicit has many benefits:

* Catches mistyped variable names. If you try using a variable that you never defined, as would happen if you mistyped a name, you would receive an error. I’ll demonstrate this later.
* Saves memory. If you don’t declare your variables and you don’t use Option Explicit, your VBA compiler implicitly assumes your variables are of data type Variant. This reserves a whopping 16 bytes of data for each variable!
* Saves time. Because your VBA compiler must choose the underlying data type for all your variant variables, your macros will run slower.

# OBJECT VARIABLES

An object variable is a variable that serves as a substitute for the actual object.

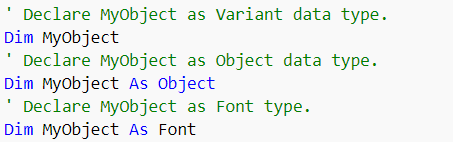
You can treat an object variable the same as the object to which it refers. You can set or return the properties of the object or use any of its methods.

To create an object variable:

* Declare the object variable.
* Assign the object variable to an object.

## DECLARATION

Use the Dim statement or one of the other declaration statements (Public, Private, or Static) to declare an object variable. A variable that refers to an object must be a Variant, an Object, or a specific type of object. For example, the following declarations are valid:



Note If you use an object variable without declaring it first, the data type of the object variable is Variant by default.

You can declare an object variable with the Object data type when the specific object type is not known until the procedure runs. Use the Object data type to create a generic reference to any object.

If you know the specific object type, you should declare the object variable as that object type. For example, if the application contains a Sample object type, you can declare an object variable for that object using either of these statements:



Declaring specific object types provides automatic type checking, faster code, and improved readability.

## ASSIGNMENT

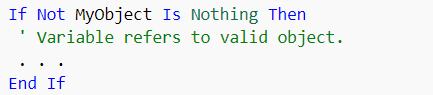
Use the Set statement to assign an object to an object variable. You can assign an object expression or Nothing. For example, the following object variable assignments are valid:



You can combine declaring an object variable with assigning an object to it by using the New keyword with the Set statement. For example:



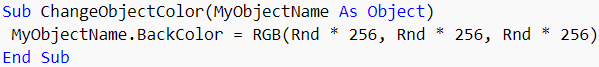
Setting an object variable equal to Nothing discontinues the association of the object variable with any specific object. This prevents you from accidentally changing the object by changing the variable. An object variable is always set to Nothing after closing the associated object, so you can test whether the object variable points to a valid object. For example:



Of course, this test can never determine with absolute certainty whether a user has closed the application containing the object to which the object variable refers.

## REFERRING

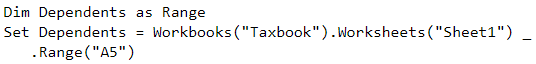
Use the Me keyword to refer to the current instance of the object where the code is running. All procedures associated with the current object have access to the object referred to as Me. Using Me is particularly useful for passing information about the current instance of an object to a procedure in another module. For example, suppose you have the following procedure in a module:



You can call the procedure and pass the current instance of the object as an argument using the following statement:



Example:



# AUTOFILTER vs ADVANCEFILTER

Below are some basic rules for filtering data:

* Headings. This is a must! They should always be in the row directly above your data. Never have a blank row between your data and the headings. Make them distinct in some way eg; bold them.
* Leave at least three blank rows above your headings. These can be used for formulas, critical data etc. You can hide the rows if you wish.
* If you have more than one List, use a different worksheet for each List. This way you can use Filters on any List without it interfering with another List.
* Organise your data so that related data is close. This will make reading from a Filtered list much easier.
* Filters will only hide Rows not Columns.
* If you have data that needs to be seen all the time don't place it to the left or right of your data as it will be hidden once you apply Filters.
* If you do have more than one List or Table on the same Worksheet leave at least one blank column between your List and Table. This will aid Excel in recognising them as separate.
* Avoid blank cells within your data.
* Sort your List or Data. This in not so much for Filters, but for the person reading the data.

If we follow these guidelines as close as possible, using Filters will be a relatively easy task.

While in most cases the AutoFilter is quite enough for most tables or lists, there are times when you need to extract data that cannot be extracted via Excel's AutoFilter. Probably the three biggest differences between the Advanced Filter and Excel's AutoFilter are:

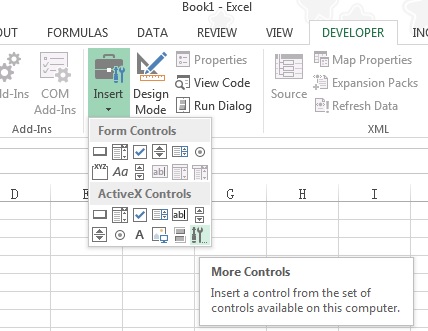
* Advanced Filter will allow us to nominate where we would like our filtered data to be placed. The choices are: Filter the list, in place or Copy to another location.
* Advanced Filter has a built-in function that will allow us to filter by unique records.
* Advanced Filter allows us to use a formula as our criteria.

There are a couple of rules that must be adhered to when using the Advanced Filter in certain ways.

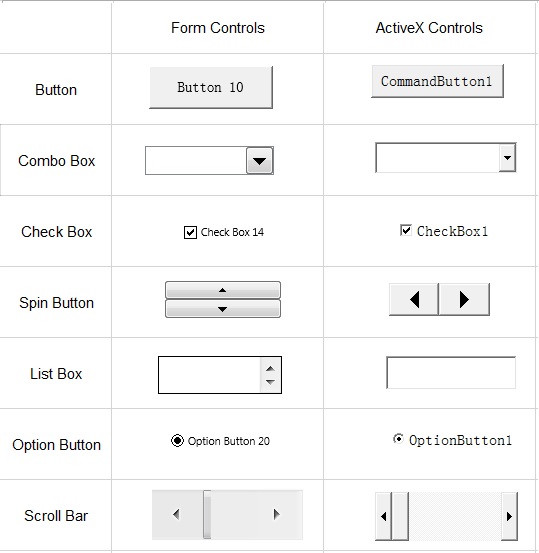
* The list must be filtered to a nominated range on the same Worksheet as the list or table resides. This means simply if our list or table is on Sheet 1, Excel will not allow us to nominate the Copy to another location option and copy to another Worksheet, we must copy to the same Worksheet that our list or table resides on. If you do wish to copy to another Worksheet simply opt to Copy to another location on the same Worksheet, then cut and paste the results to another Worksheet.
* When we use a formula as our criteria, the formula itself must evaluate to either True or False. We will be using formulas later, and you will see what I mean when I show you the example.

# FORM vs ACTIVEX

There are two different kinds of Controls in Excel spreadsheet both are very similar, but we need to know the difference between them. UserForm also has its Control but it is not discussed in this topic.

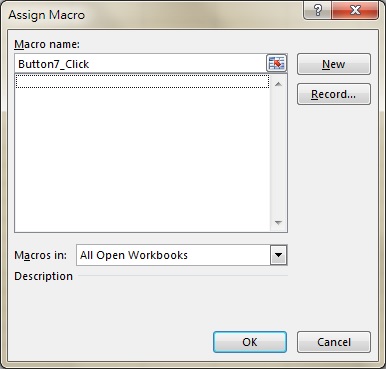


As seen in the picture above, both Form Controls and ActiveX Controls have common Controls in different order, but the appearance is slightly different.

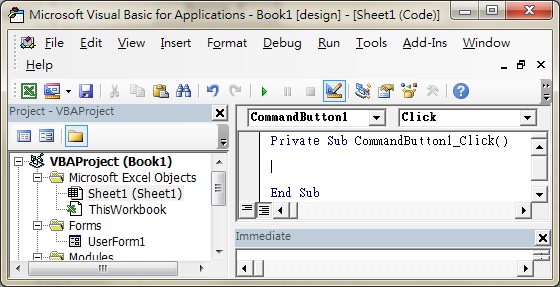


While ActiveX is a product of Microsoft hence it cannot be run in Mac computer. If you want to run on Mac, make sure you use Form Controls which do not contain any VBA.

When you drag a Button from Form Controls, there is a pop-up message asking you to assign a Macro to the Button. You need to create a Macro in Module in advance for selection.



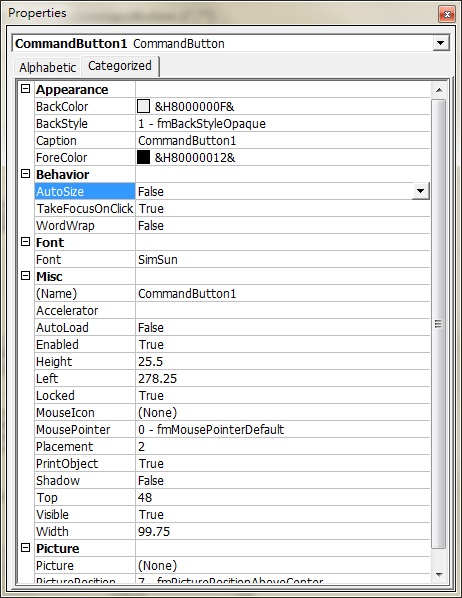
When you drag a Button from ActiveX Controls, you don’t need to assign a Macro. Instead, you can double click on the button and you will enter VBE where a button click event is created under the current worksheet. The below example shows the Click Event of a command button, you can also select other Events available.



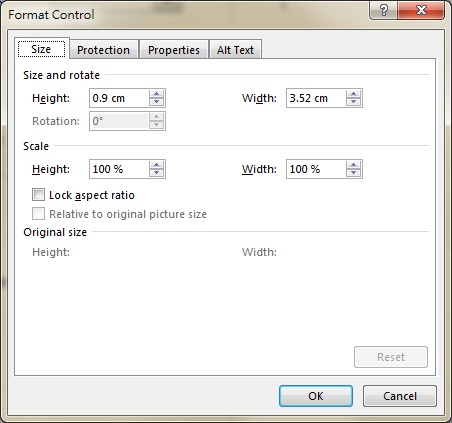
Other than Button, all other Form Controls refer to the Macro that you create in Module, while ActiveX Controls have their own Properties and Event.

ActiveX Controls have Properties setting but Form Controls don’t.

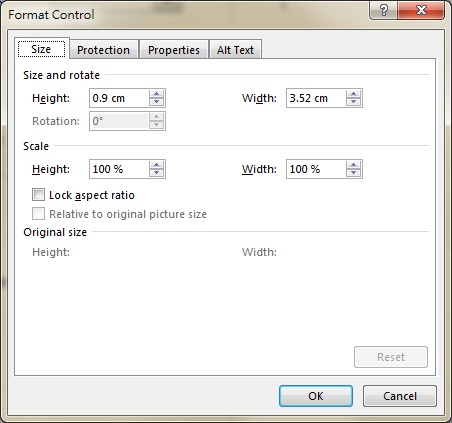
In Design Mode, right click on an ActiveX Control and select “Properties”. Below is the Properties of a Button.



You can also right click on an ActiveX control and then select Format Control for basic formatting.



To format a Form Control, you can only select Format Control. Although ActiveX Controls do not have Properties setting, some of the settings are present in Format Control instead.



# LOOPS

A loop statement allows us to execute a statement or group of statements multiple times. Following is the general form of a loop statement in VBA.

VBA provides the following types of loops to handle looping requirements. Click the following links to check their detail.

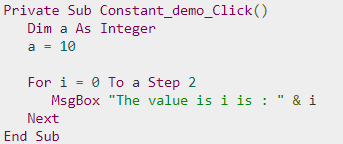
1. For loop
2. For each loop
3. While wend loop
4. Do while loops
5. Do until loops

## FOR LOOP

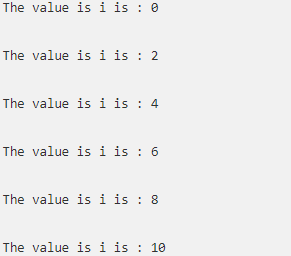
Following is the flow of control in a For Loop −

* The For step is executed first. This step allows you to initialize any loop control variables and increment the step counter variable.
* Secondly, the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement, just after the For Loop.
* After the body of the For loop executes, the flow of control jumps to the next statement. This statement allows you to update any loop control variables. It is updated based on the step counter value.
* The condition is now evaluated again. If it is true, the loop executes, and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the For Loop terminates.

Example:



Result:

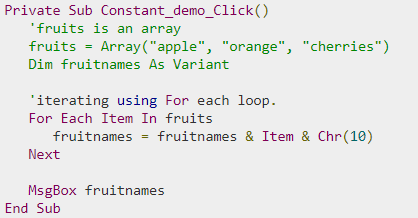


## FOR EACH

A For-Each loop is used to execute a statement or a group of statements for each element in an array or collection.

A For-Each loop is like For Loop; however, the loop is executed for each element in an array or group. Hence, the step counter won't exist in this type of loop. It is mostly used with arrays or used in context of the File system objects to operate recursively.

Example: Here Chr (10) is used for line feed (new linefeed)



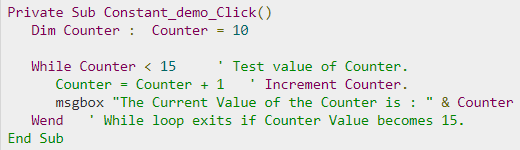
Result:



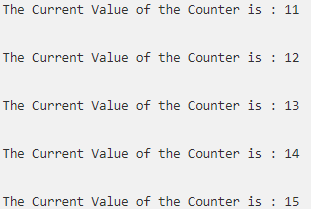
## WHILE WEND

In a While…Wend loop, if the condition is True, all the statements are executed until the Wend keyword is encountered.

If the condition is false, the loop is exited and the control jumps to the very next statement after the Wend keyword.

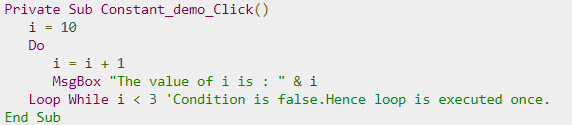


Result:



## DO WHILE

A Do…While loop is used when we want to repeat a set of statements as long as the condition is true. The condition may be checked at the beginning of the loop or at the end of the loop.

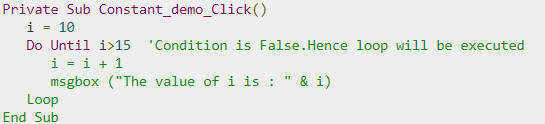


Result:

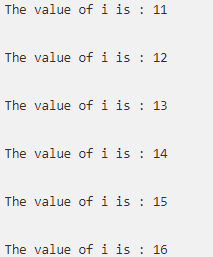


## DO UNTIL

A Do…Until loop is used when we want to repeat a set of statements as long as the condition is false. The condition may be checked at the beginning of the loop or at the end of loop.



Result:



## LOOP CONTROL STATEMENTS

Loop control statements change execution from its normal sequence. When execution leaves a scope, all the remaining statements in the loop are NOT executed.

VBA supports the following control statements. Click the following links to check their detail.

* Exit For statement

Terminates the For-Loop statement and transfers the execution to the statement immediately following the loop

* Exit Do statement

Terminates the Do While statement and transfers the execution to the statement immediately following the loop.

# ERROR HANDLING

There are three types of errors in programming: (a) Syntax Errors, (b) Runtime Errors, and (c) Logical Errors.

Assume if we have a runtime error, then the execution stops by displaying the error message. As a developer, if we want to capture the error, then Error Object is used.

VBA enables an error-handling routine and can also be used to disable an error-handling routine. Without an On Error statement, any run-time error that occurs is fatal: an error message is displayed, and the execution stops abruptly.

* GoTo line

Enables the error-handling routine that starts at the line specified in the required line argument. The specified line must be in the same procedure as the On-Error statement, or a compile-time error will occur.

* GoTo 0

Disables the enabled error handler in the current procedure and resets it to Nothing.

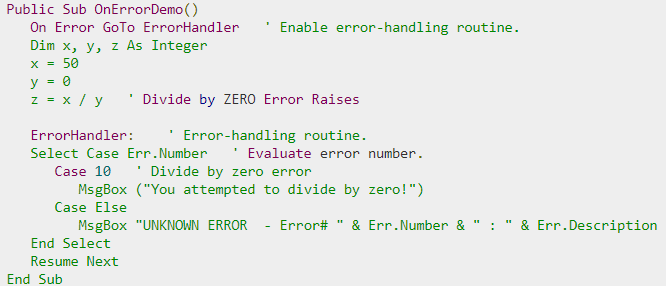
* GoTo -1

Disables the enabled exception in the current procedure and resets it to Nothing.

* Resume Next

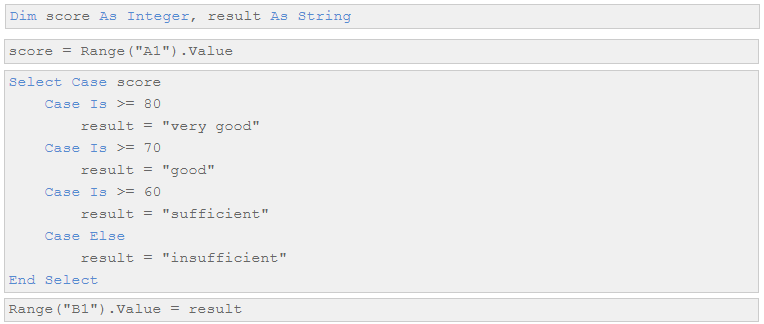
Specifies that when a run-time error occurs, the control goes to the statement immediately following the statement where the error occurred, and the execution continues from that point.

Example:

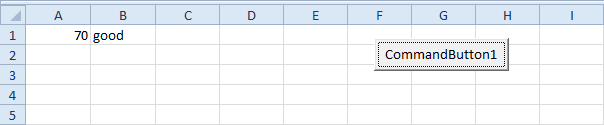


# SELECT CASE

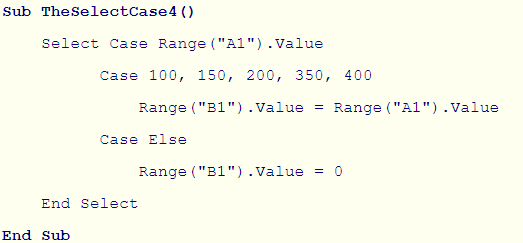
Instead of multiple If Then statements in **Excel VBA**, you can use the **Select Case structure**.



Result:



Example2:



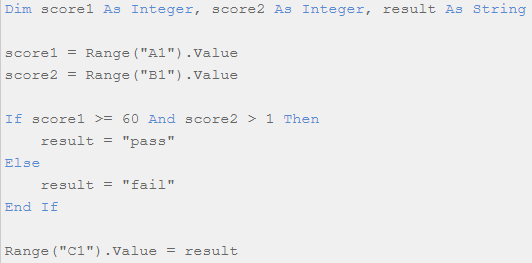
# OPERATORS

Below are some important logical operators in VBA.



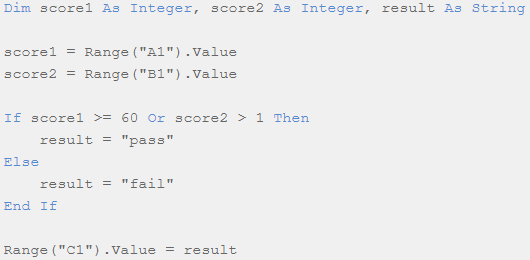
## AND

Example below:



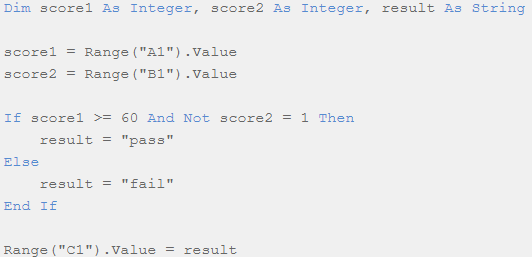
## OR

Example below:



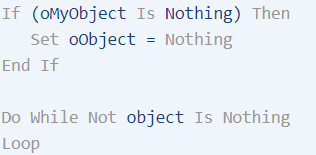
## NOT

Example below:



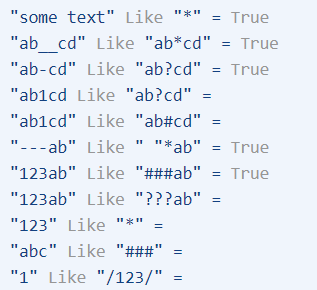
## IS

Example below:



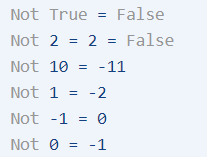
## LIKE

Example below:



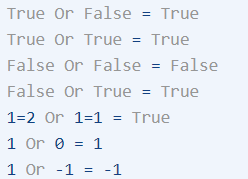
## NOT

Example below:



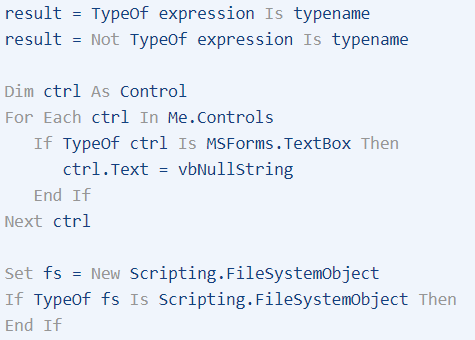
## XOR

Example below:



## TYPEOF

Example below:



# DIR

Description Returns True if a specified key exists in the Dictionary object; False if it does not. Syntax object. Exists(key) The Exists method syntax has these parts:

DIR is a very special function in VBA, its job is to return a string representing the name of a file, directory, or an archive that matches a specified pattern. DIR function only returns the first file name or folder name from a location that matches the specified attributes.

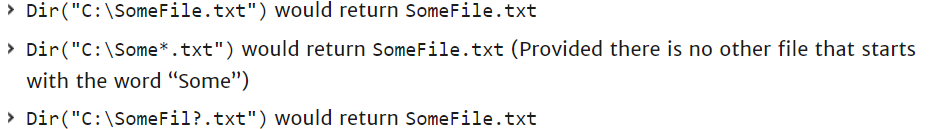
To fetch other file names or folder names from that location which match previously specified attributes, you need to call the DIR Function again with no arguments. This property of DIR Function can be quite useful in iterating or enlisting files or directories present inside a location.

Please note that DIR function can just give you the name of the file. But if you also need attributes of a file (like: last modified date, size, etc.) then consider using FileObjectSystem.

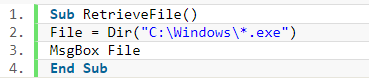
Few Important points about VBA DIR Function:

* Both the arguments in DIR Function are optional.
* You can use wildcard character (like: ‘?’ or ‘\*’) with DIR to specify multiple files.
  + ‘\*’ allows you to match any string of any length (including zero length)
  + ‘?’ allows you to match any single character.
* You must call the DIR Function along with the ‘pathname’ parameter for the first time. Subsequent calls to DIR Function can be made with no arguments to retrieve the next item.

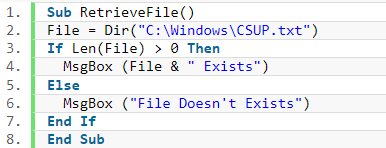
Example1:



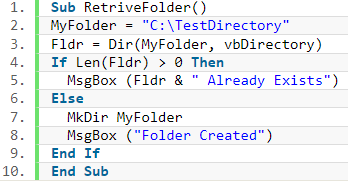
Example2: Display name of .exe if any at the location.



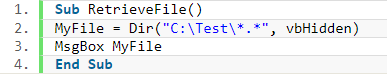
Example3: To check if the file exists

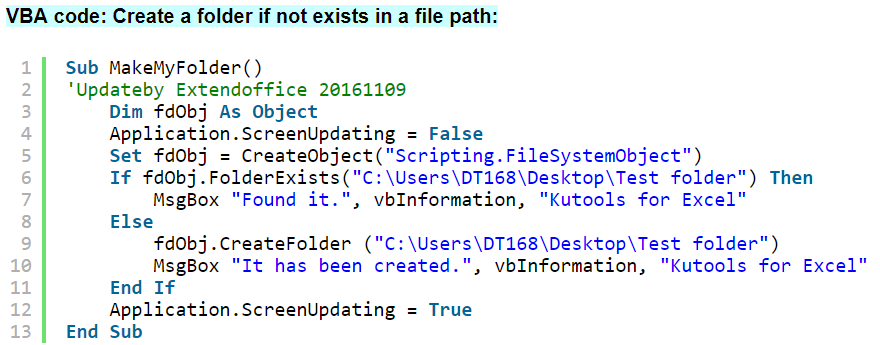


Example4: Create file if does not exists



Example5: Fetch name of hidden files

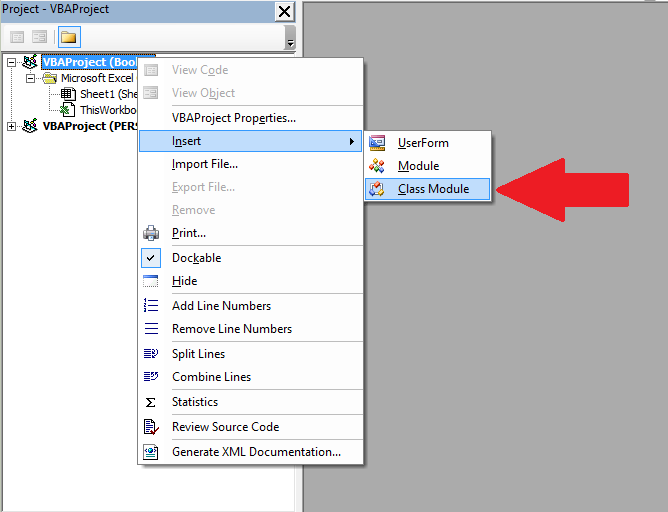




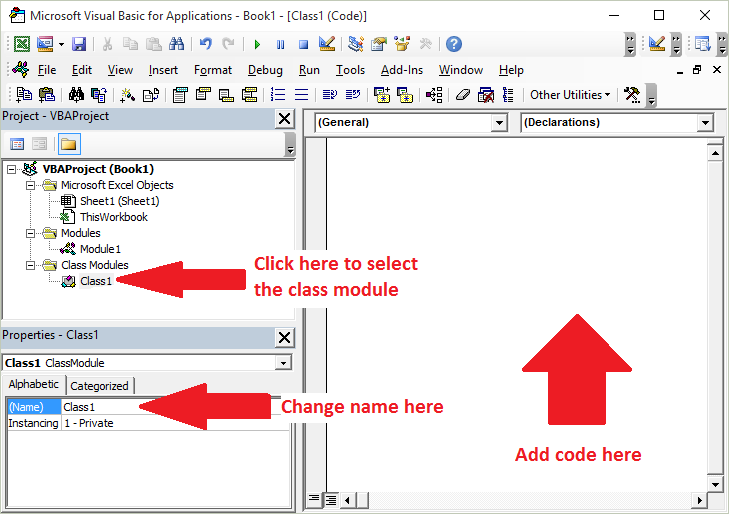
# CLASS MODULE

In VBA we have built-in objects such as the Collection, Workbook, Worksheet and so on. The purpose of VBA Class Modules is to allow us to custom build our own objects.

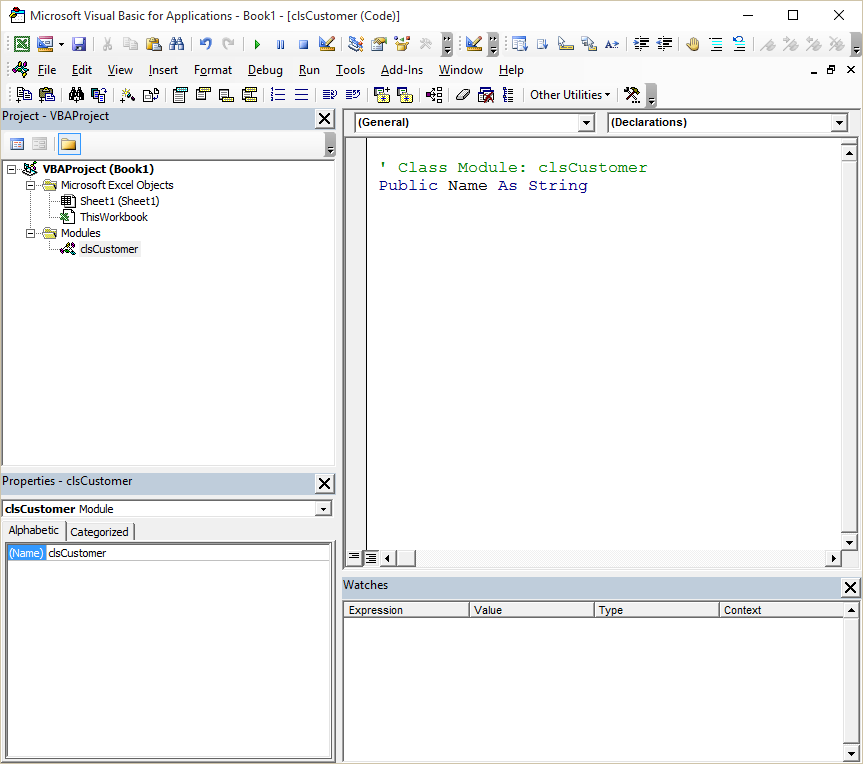
Let’s look at a very simple example of creating a class module and using it in our code. To create a class module, we right-click in the Project window and then select Insert and Class Module.



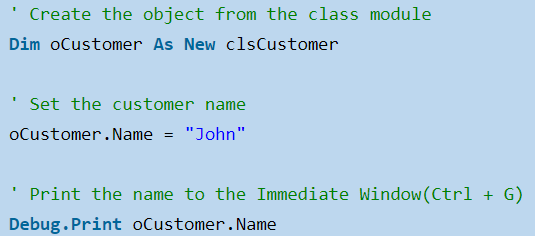
Our new class is called Class1. We can change the name in the Properties window as the following screenshot shows



Let’s change the name of the class module to clsCustomer. Then we will add a variable to the class module like this



We can use now use this class module in any module (standard or class) in our workbook. For example

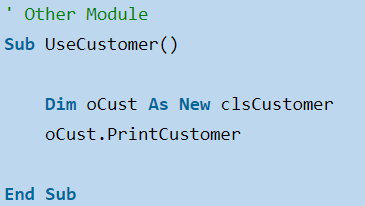


There are four different items in a class module. These are

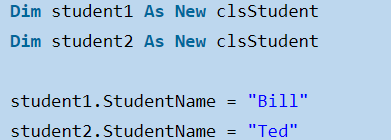
* Methods – functions/subs.
* Member variables – variables.
* Properties– types of functions/subs that behave like variables.
* Events – subs that are triggered by an event.

## MODULE VS CLASS MODULE

If you want to use a sub/function etc. from a class module you must create the object first.



When you create a variable in a normal module there is only one copy of it. For a class module, there is one copy of the variable for each object you create.



## CLASS MODULE PROPERTIES

Get – returns an object or value from the class

Let – sets a value in the class

Set – sets an object in the class

## REASONS FOR NOT USING A CLASS MODULE

* No intellisense - if passed or declared as Objects or Variants (commonly done/lazy)
* Harder/impossible to make changes at run-time. You always must reset the project
* Must be slower - prove this with an example
* Must use more resources - prove this with an example
* The Watch window only displays 256 items in a collection. Classes are always used in conjunction with collections
* The Watch window displays duplicates, properties and private variables - SS
* If properties and private variables have the same name then infinite loops can occur, resulting in "Out of Stack" run-time errors.

## REASONS FOR USING A CLASS MODULE

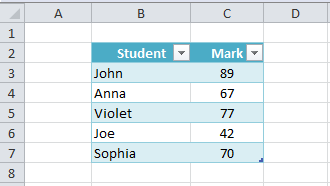
There are a few occasions when a class module is necessary:

* Responding to events (application or chart)
* Creating a single event for multiple controls. Using a Controls Collection
* Encapsulating your VBA procedures and functions (or even Windows API calls) so they are easier to use

# ARRAY

A VBA array is a type of variable. It is used to store lists of data of the same type. An example would be storing a list of countries or a list of weekly totals.

In VBA a normal variable can store only one value at a time. Suppose we wish to store the marks of another student then we need to create a second variable. In the following example we have the marks of five students.



We can use arrays in this case to store multiple values in a single variable. The fact arrays use an index (also called a subscript) to access each item is important. It means we can easily access all the items in an array using a For Loop.

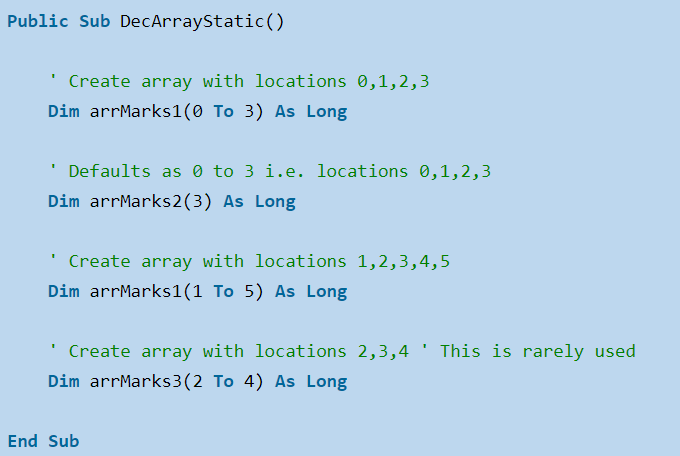
There are two types of arrays in VBA

* Static – an array of fixed size.
* Dynamic – an array where the size is set at run time.

The difference between these arrays mainly in how they are created. Accessing values in both array types is the same.

## STATIC ARRAY

A static array is declared as shown below:

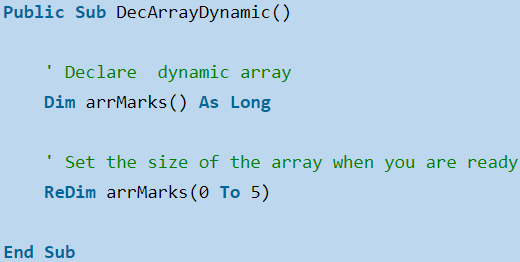


As you can see the size is specified when you declare a static array. The problem with this is that you can never be sure in advance the size you need. Each time you run the Macro you may have different size requirements.

If you do not use all the array indexes, then the resources are being wasted. If you need more indexes to store more elements, then you can use ReDim but this is essentially creating a new static array.

## DYNAMIC ARRAY

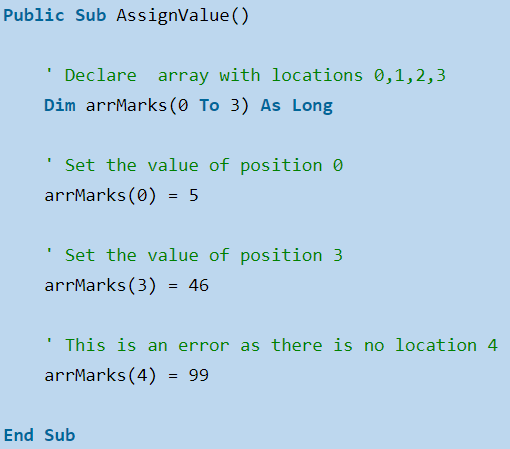
The dynamic array does not have such problems. You do not specify the size when you declare it. Therefore, you can then grow and shrink as required.



The dynamic array is not allocated until you use the ReDim statement. The advantage is you can wait until you know the number of items before setting the array size. With a static array you must give the size up front.

## ASSIGNING VALUES

To assign values to an array you use the number of the location. You assign value for both array types the same way.



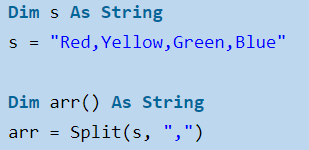
Result:

VBA Array 2

The number of the location is called the subscript or index.

The array created by the Array Function will start at index zero unless you use Option Base 1 at the top of your module. Then it will start at index one. In programming it is generally considered poor practice to have your actual data in the code. However sometimes it is useful when you need to test some code quickly. The Split function is used to split a string into an array based on a delimiter. A delimiter is a character such as a comma or space that separates the items.

The following code will split the string into an array of three elements.

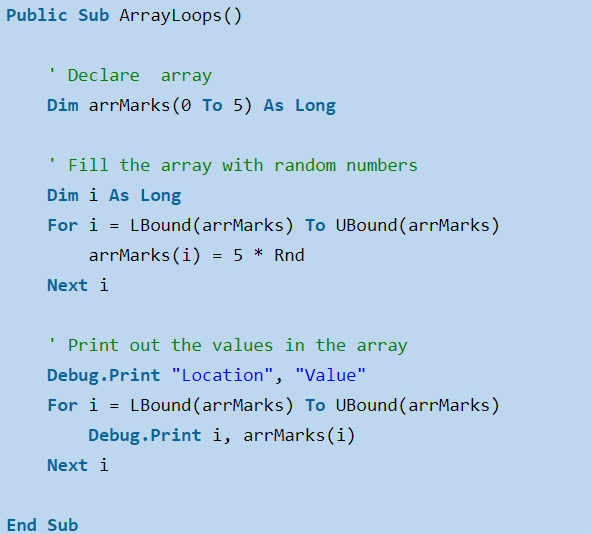


Result:

Arrays VBA

## LBOUND & UBOUND

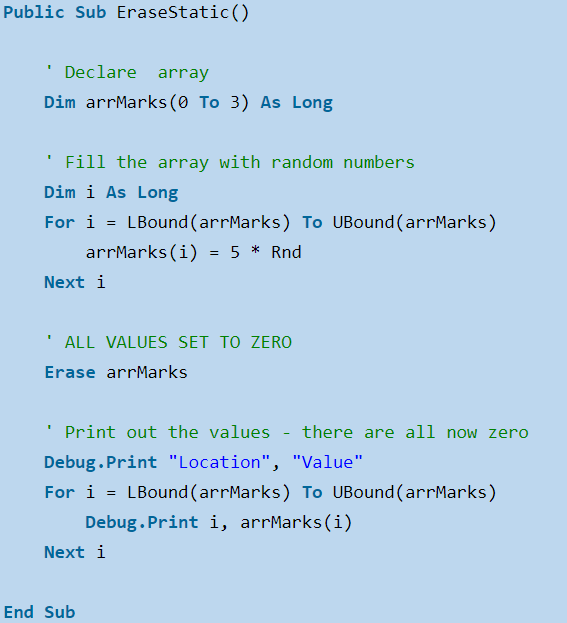
The functions LBound and Unbound are very useful. Using them means our loops will work correctly with any array size. The real benefit is that if the size of the array changes we do not have to change the code for printing the values. A loop will work for an array of any size as long as you use these functions.



## ERASE

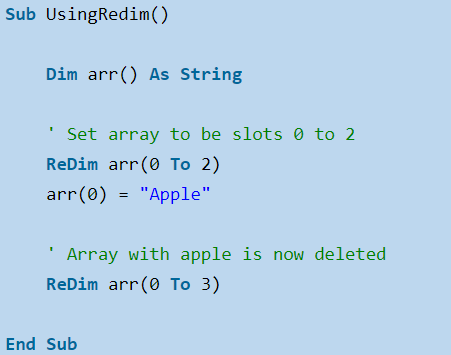
The Erase function can be used on arrays but performs differently depending on the array type.

* For a static Array the Erase function resets all the values to the default. If the array is of integers, then all the values are set to zero. If the array is of strings, then all the strings are set to “” and so on.
* For a Dynamic Array the Erase function Deallocates memory. That is, it deletes the array. If you want to use it again you must use ReDim to Allocate memory.



## REDIM WITH PRESERVE

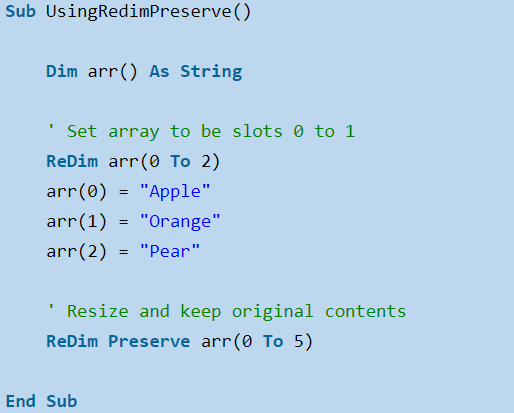
If we use ReDim on an existing array, then the array and its contents will be deleted. In the following example, the second ReDim statement will create a completely new array. The original array and its contents will be deleted.



If we want to extend the size of an array without losing the contents, we can use the Preserve keyword. When we use Redim Preserve the new array must be bigger and start at the same dimension e.g.

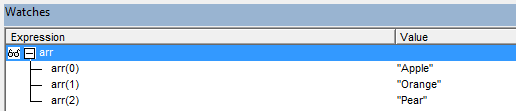


In the following code we create an array using ReDim and then fill the array with types of fruit. We then use Preserve to extend the size of the array, so we don’t lose the original contents.



You can see from the screenshots below, that the original contents of the array have been “Preserved”.

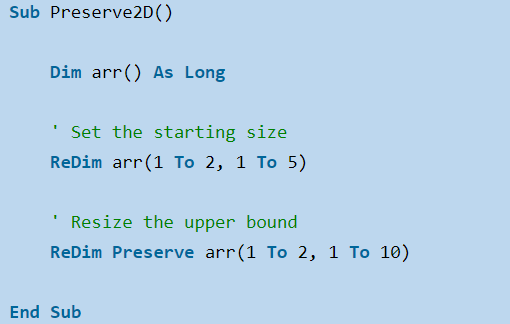
Before ReDim Preserve



After ReDim Preserve

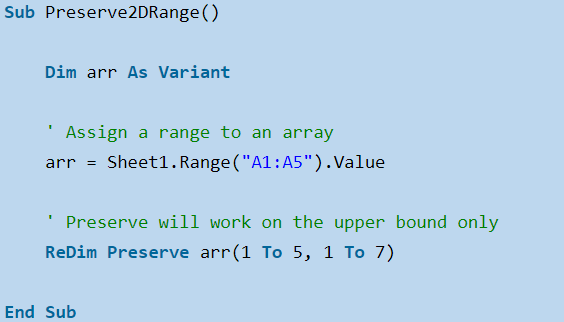


Using Preserve with two-dimensional array.

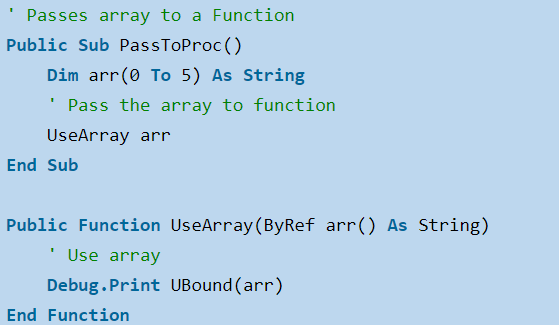


If we try to use Preserve on a lower bound, we will get the “Subscript out of range” error.

When we read from a range to an array, it automatically creates a two-dimensional array, even if we have only one column. The same Preserve rules apply. We can only use Preserve on the upper bound as this example shows:



Passing array to function

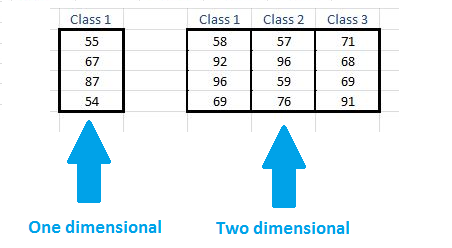


## TWO-DIMENSIONAL ARRAY

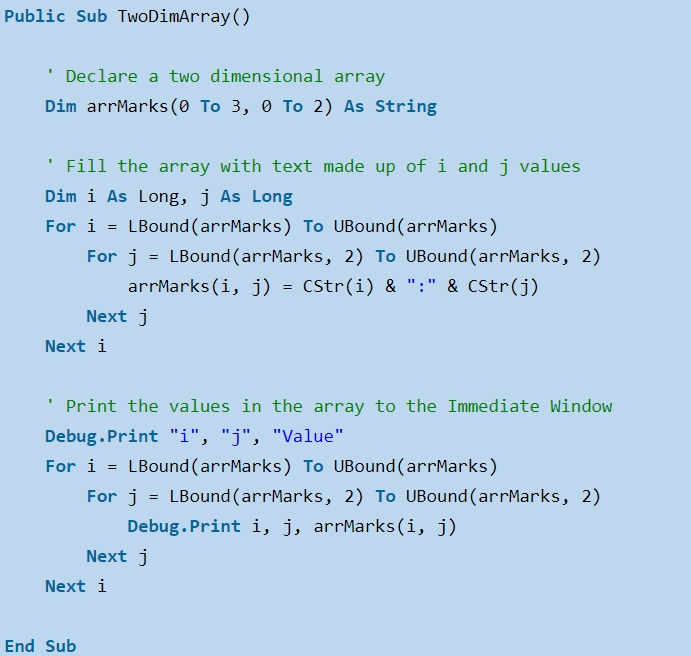
The arrays we have been looking at so far have been one dimensional array. This means the arrays are one list of items.

A two-dimensional array is essentially a list of lists. If you think of a single spreadsheet column as a single dimension, then more than one column is two dimensional. In fact, a spreadsheet is the equivalent of a 2-dimensional array. It has two dimensions – rows and columns.

The following image shows two groups of data. The first is a one-dimensional layout and the second is two dimensional.



LBound and UBound have a second argument of 2. This specifies that it is the upper or lower bound of the second dimension. That is the start and end location for j. The default value 1 which is why we do not need to specify it for the i loop.



## ARRAY METHODS

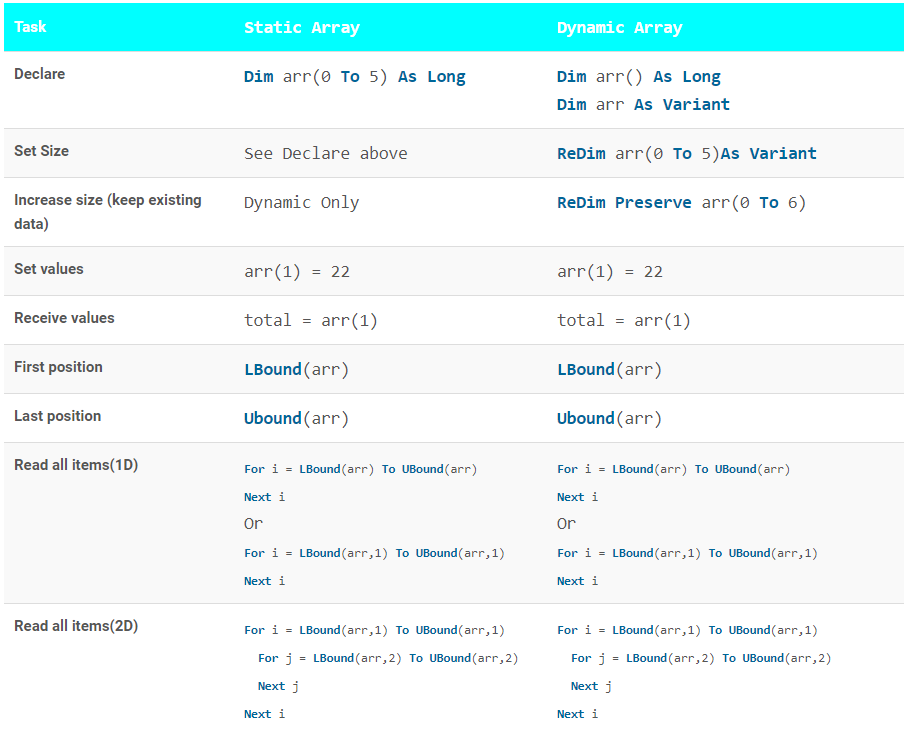
There are various inbuilt functions within VBScript which help the developers to handle arrays effectively. All the methods that are used in conjunction with arrays are listed below. Please click on the method name to know about it in detail.

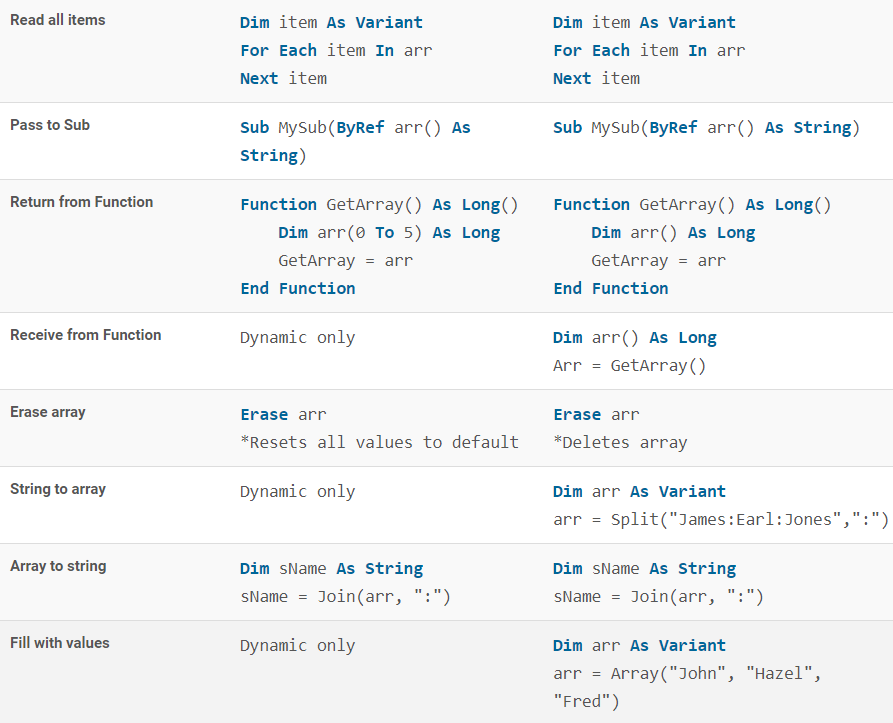
* LBound: A Function, which returns an integer that corresponds to the smallest subscript of the given arrays.
* UBound: A Function, which returns an integer that corresponds to the largest subscript of the given arrays.
* Split: A Function, which returns an array that contains a specified number of values. Split based on a delimiter.
* Join: A Function, which returns a string that contains a specified number of substrings in an array. This is an exact opposite function of Split Method.
* Filter: A Function, which returns a zero-based array that contains a subset of a string array based on a specific filter criteria.
* IsArray: A Function, which returns a boolean value that indicates whether the input variable is an array.
* Erase: A Function, which recovers the allocated memory for the array variables.

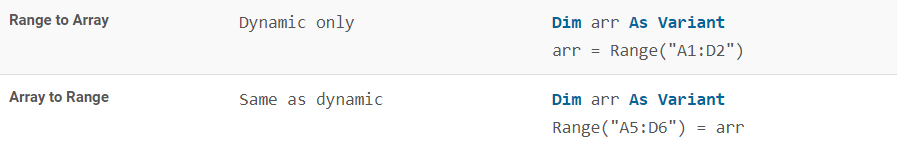
## CONCLUSION

The following is a summary of the main points:

1. Arrays are an efficient way of storing a list of items of the same type.
2. You can access an array item directly using the number of the location which is known as the subscript or index.
3. The common error “Subscript out of Range” is caused by accessing a location that does not exist.
4. There are two types of arrays: Static and Dynamic.
5. Static is used when the size of the array is always the same.
6. Dynamic arrays allow you to determine the size of an array at run time.
7. LBound and UBound provide a safe way of find the smallest and largest subscripts of the array.
8. The basic array is one dimensional. You can also have multi-dimensional arrays.
9. You can only pass an array to a procedure using ByRef. You do this like this: ByRef arr () as long.
10. You can return an array from a function but the array, it is assigned to, must not be currently allocated.
11. A worksheet with its rows and columns is essentially a two-dimensional array.
12. You can read directly from a worksheet range into a two-dimensional array in just one line of code.
13. You can also write from a two-dimensional array to a range in just one line of code.



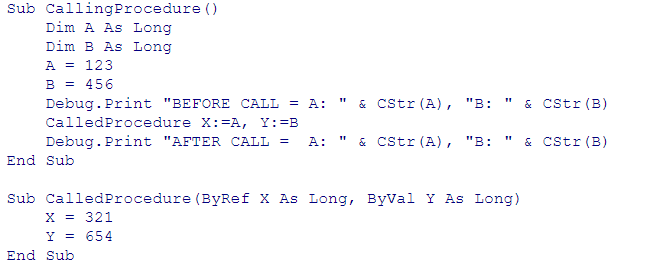




# BYVAL VS BYREF

Passing ByRef or ByVal indicates whether the actual value of an argument is passed to the CalledProcedure by the CallingProcedure, or whether a reference (called a pointer in some other languages) is passed to the CalledProcedure. If an argument is passed ByRef, the memory address of the argument is passed to the CalledProcedure and any modification to that parameter by the CalledProcedure is made to the value in the CallingProcedure. If an argument is passed ByVal, the actual value, not a reference to the variable, is passed to the CalledProcedure.

A simple example will illustrate this clearly:



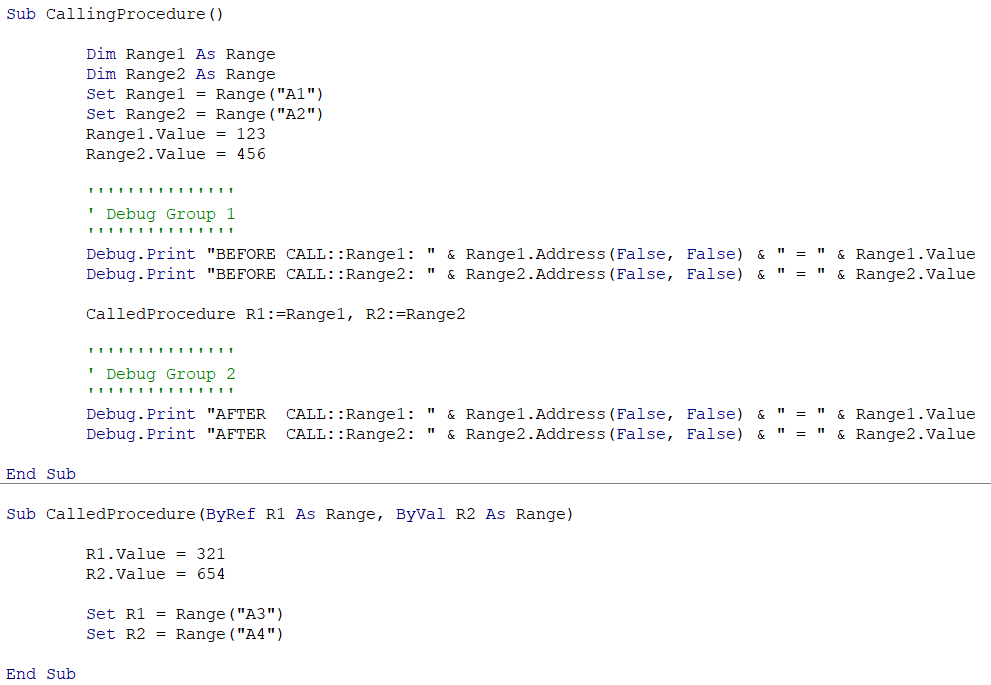
In the CallingProcedure the variables A and B are assigned the values 123 and 456, respectively. Then, the CalledProcedure is called passing arguments A and B. Within CalledProcedure the parameters X and Y are assigned the values 321 and 654 respectively, and control is returned to the procedure CallingProcedure.

Since the parameter X was declared with ByRef, a reference or pointer to A was passed to CalledProcedure and any modification to the X parameter in CalledProcedure affects the variable A in CallingProcedure. The parameter Y was declared with ByVal, so only the actual value of B was passed to CalledProcedure. Changes made to the parameter Y are not made to the variable B. This shows that A was modified by CalledProcedure, but that B was not changed.

## PASSING OBJECTS BYREF AND BYVAL

Objects are always passed by reference. The ByRef and ByVal modifiers indicate how the reference is passed to the called procedure. When you pass an object type variable to a procedure, the reference or address to the object is passed -- you never really pass the object itself. When you pass an object ByRef, the reference is passed by reference and the called procedure can change the object to which that reference refers to. When an object is passed ByVal an copy of the reference (address) of the object is passed.

As is so often the case, an example will serve well to illustrate this:



In the CallingProcedure, the variable Range1 is set to Range("A1") and the variable Range2 is set to Range("A2"). Then CalledProcedure is called, passing ByRef R1 and ByVal R2. The CalledProcedure sets the values of these ranges to new values and then changes the ranges to which R1 and R2 refer. Since R1 was passed ByRef, the CalledProcedure can change the cell to which Range1 in CallingProcedure refers to. As is confirmed by the second group of Debug.Print statements, the variable Range1 now refers to (points to) Range("A3"), not Range("A1"). However, since R2 was passed ByVal, the CalledProcedure cannot change the range to which Range2 refers to in CallingProcedure.