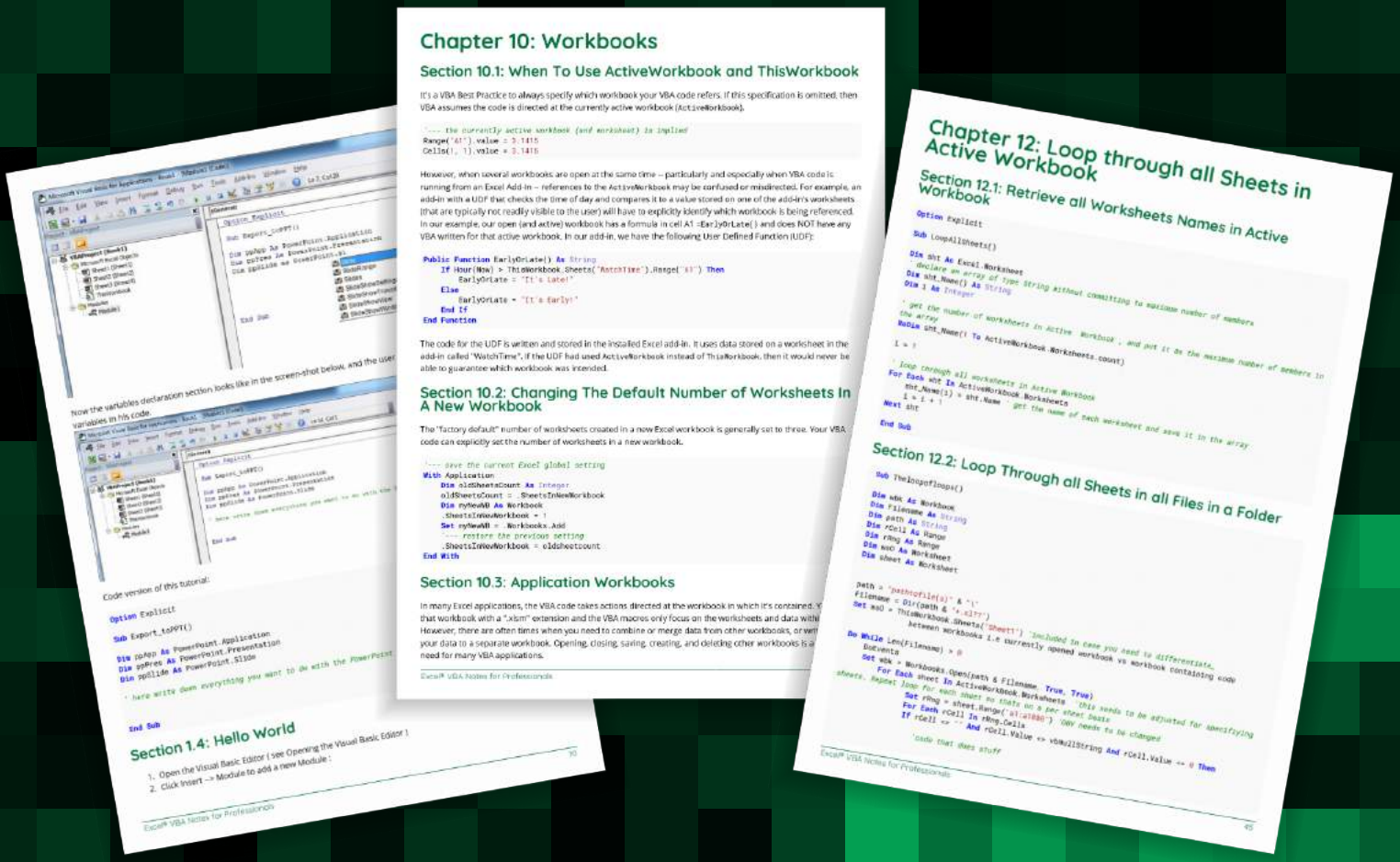
Excel® VBA

Notes for Professionals



100+ pages

of professional hints and tricks

Contents

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/ExcelVBABook

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with Excel VBA

Microsoft Excel includes a comprehensive macro programming language called VBA. This programming language provides you with at least three additional resources:

1. Automatically drive Excel from code using Macros. For the most part, anything that the user can do by manipulating Excel from the user interface can be done by writing code in Excel VBA.
2. Create new, custom worksheet functions.
3. Interact Excel with other applications such as Microsoft Word, PowerPoint, Internet Explorer, Notepad, etc.

VBA stands for Visual Basic for Applications. It is a custom version of the venerable Visual Basic programming language that has powered Microsoft Excel's macros since the mid-1990s.

IMPORTANT
Please ensure any examples or topics created within the excel-vba tag are **specific** and **relevant** to the use of VBA with Microsoft Excel. Any suggested topics or examples provided that are generic to the VBA language should be declined in order to prevent duplication of efforts.

- on-topic examples:

  ✓ *Creating and interacting with worksheet objects*
  ✓ *The `WorksheetFunction` class and respective methods*
  ✓ *Using the `xlDirection` enumeration to navigate a range*

- off-topic examples:

  ✗ *How to create a 'for each' loop*
  ✗ *`MsgBox` class and how to display a message*
  ✗ *Using WinAPI in VBA*

VB

Version	Release Date
VB6	1998-10-01
VB7	2001-06-06
WIN32	1998-10-01
WIN64	2001-06-06
MAC	1998-10-01

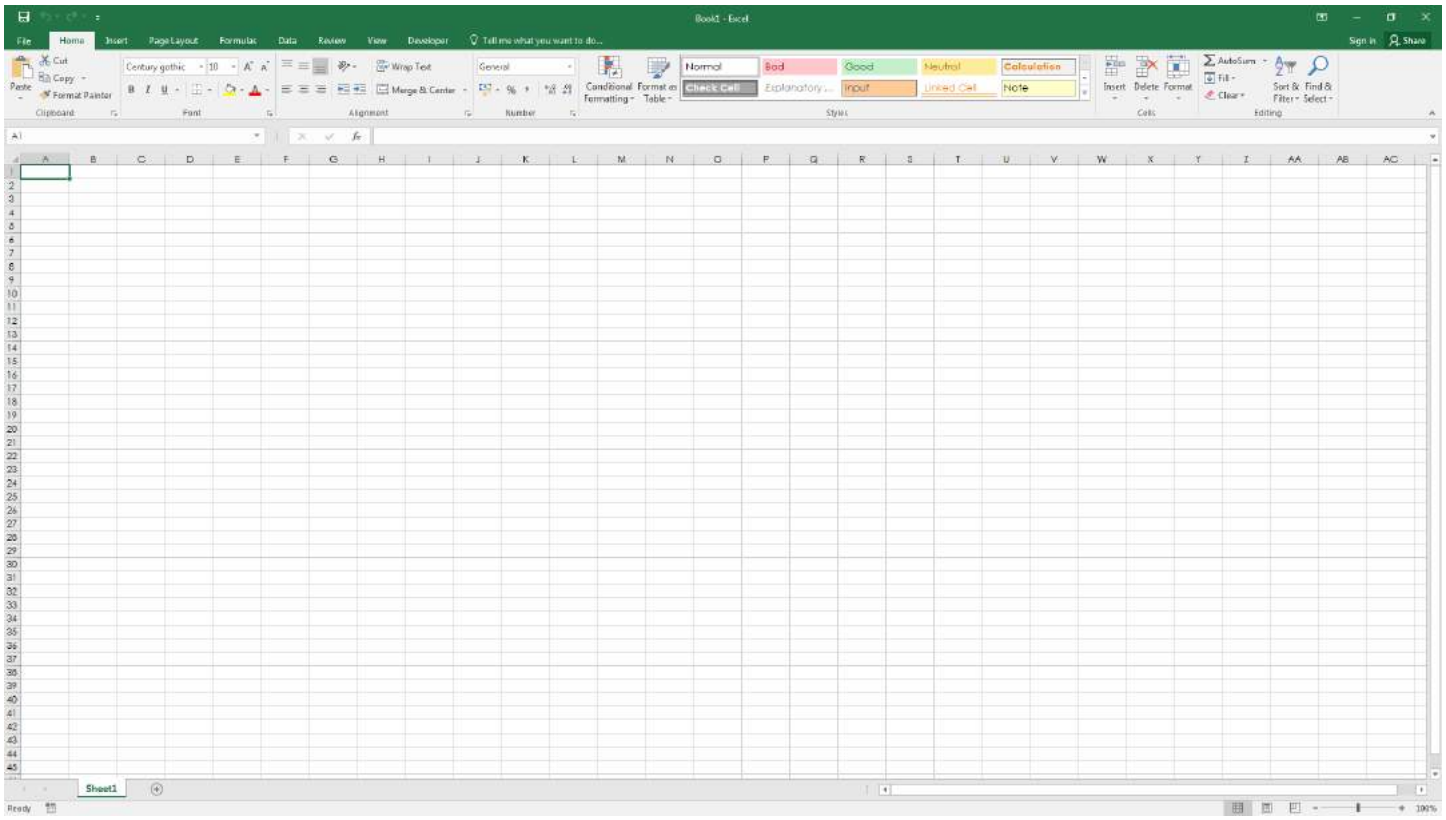
Excel

Version	Release Date
16	2016-01-01
15	2013-01-01
14	2010-01-01
12	2007-01-01
11	2003-01-01
10	2001-01-01
9	1999-01-01

8 1997-01-01
7 1995-01-01
5 1993-01-01
2 1987-01-01

Section 1.1: Opening the Visual Basic Editor (VBE)

Step 1: Open a Workbook

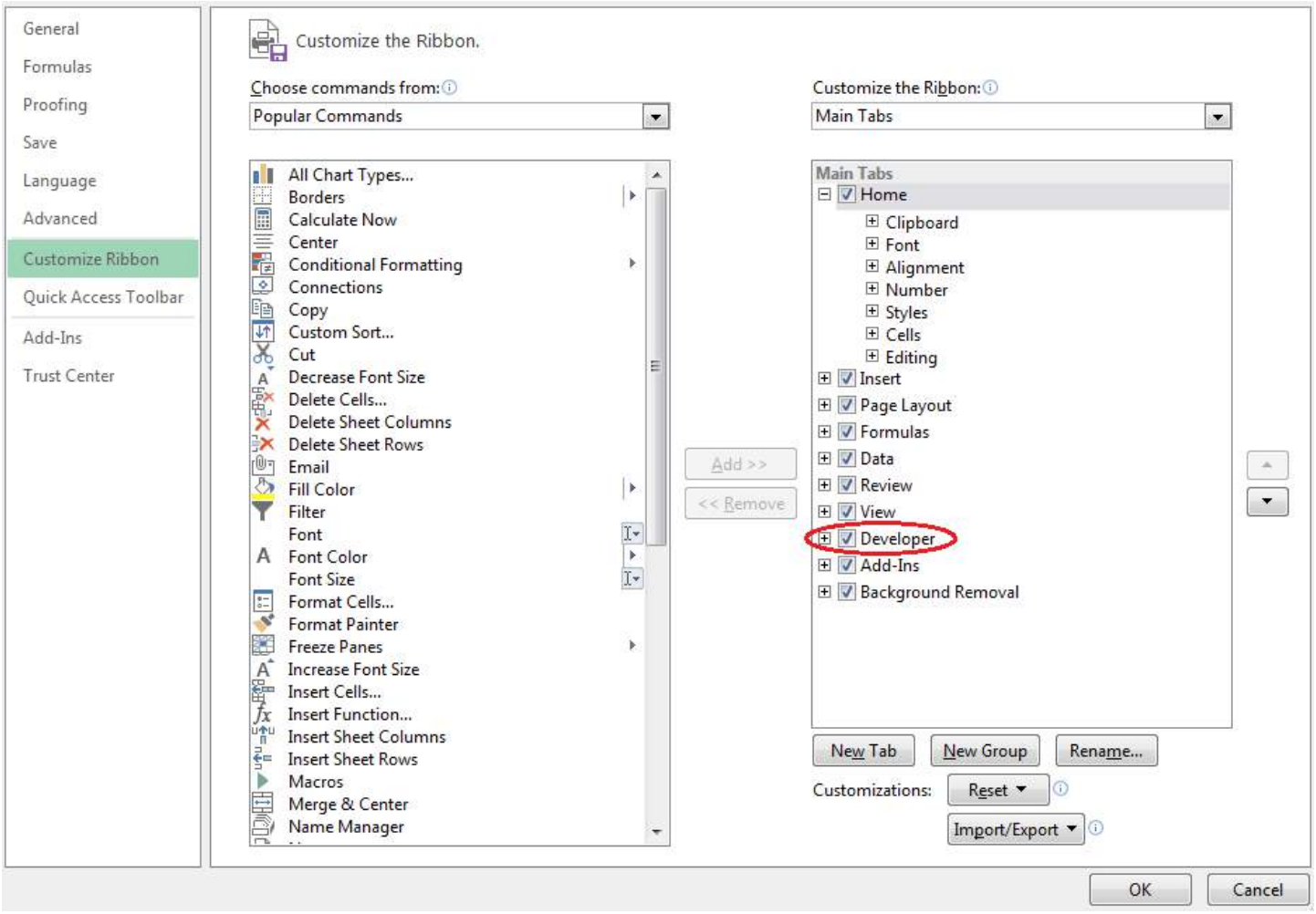


Step 2 Option A: Press `Alt` + `F11`

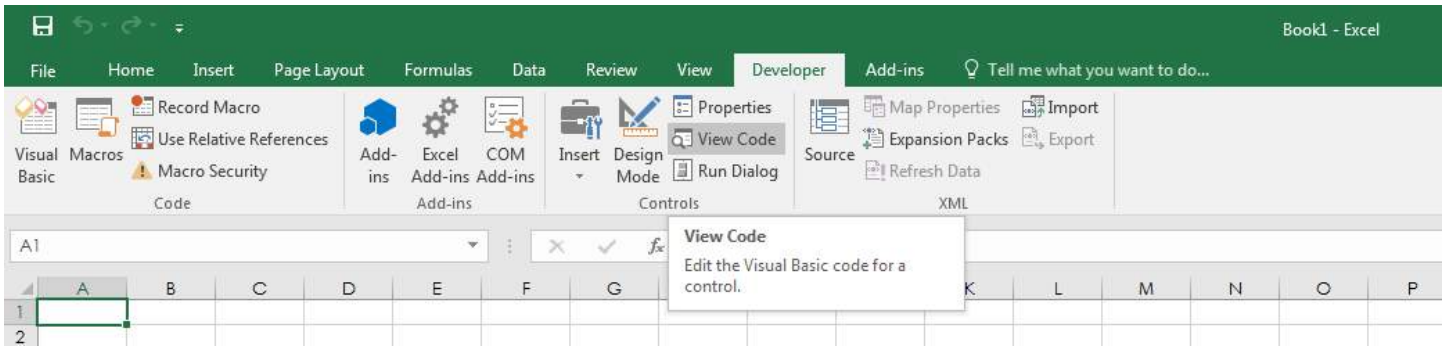
This is the standard shortcut to open the VBE.

Step 2 Option B: Developer Tab --> View Code

First, the Developer Tab must be added to the ribbon. Go to File -> Options -> Customize Ribbon, then check the box for developer.

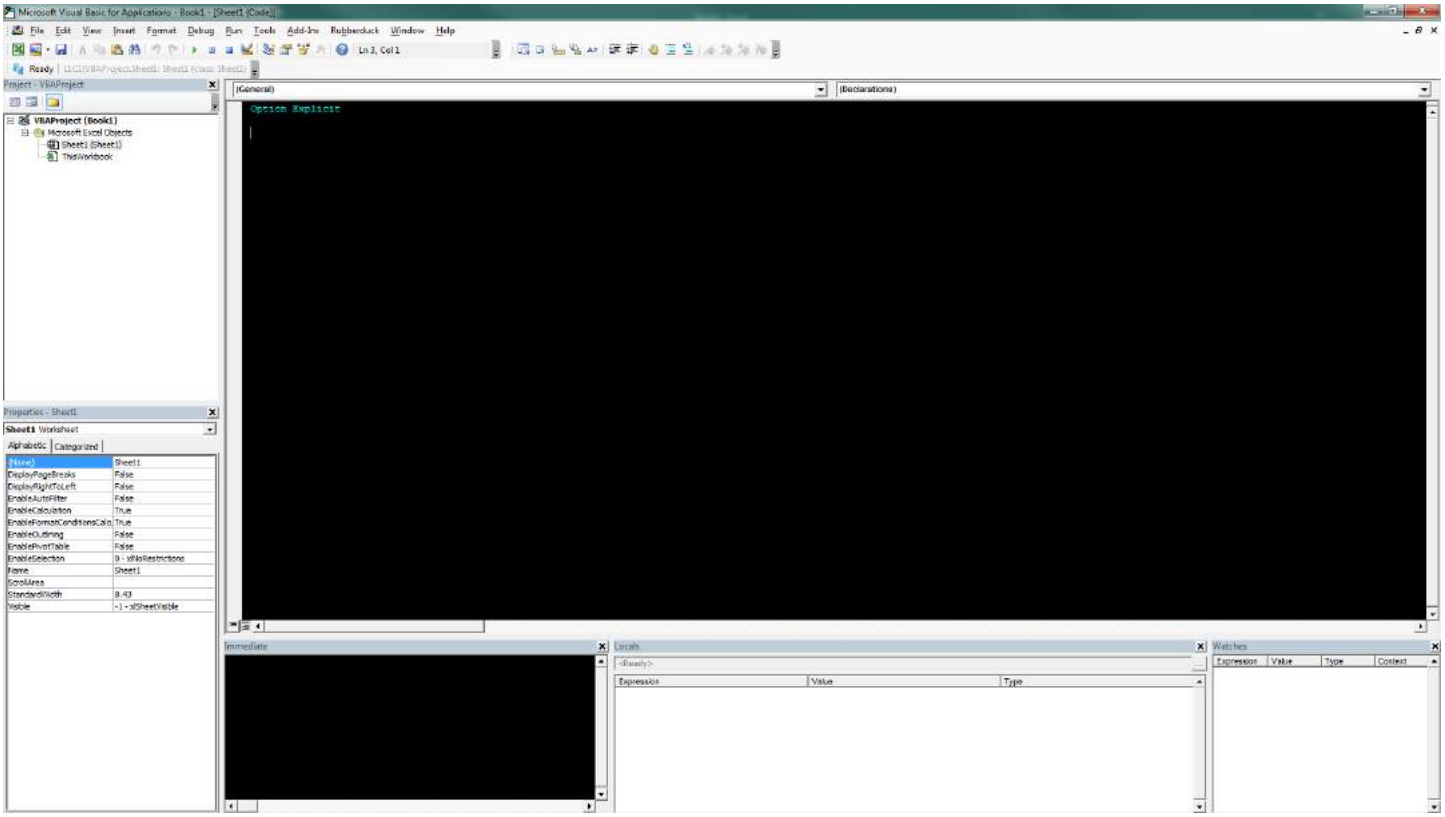


Then, go to the developer tab and click "View Code" or "Visual Basic"



Step 2 Option C: View tab > Macros > Click Edit button to open an Existing Macro

All three of these options will open the Visual Basic Editor (VBE):



Section 1.2: Declaring Variables

To explicitly declare variables in VBA, use the `Dim` statement, followed by the variable name and type. If a variable is used without being declared, or if no type is specified, it will be assigned the type `Variant`.

Use the `Option Explicit` statement on first line of a module to force all variables to be declared before usage (see ALWAYS Use "Option Explicit").

Always using `Option Explicit` is highly recommended because it helps prevent typo/spelling errors and ensures variables/objects will stay their intended type.

```
Option Explicit

Sub Example()
    Dim a As Integer
    a = 2
    Debug.Print a
    'Outputs: 2

    Dim b As Long
    b = a + 2
    Debug.Print b
    'Outputs: 4

    Dim c As String
    c = "Hello, world!"
    Debug.Print c
    'Outputs: Hello, world!
End Sub
```

Multiple variables can be declared on a single line using commas as delimiters, but **each type must be declared individually**, or they will default to the `Variant` type.

```
Dim Str As String, IntOne, IntTwo As Integer, Lng As Long
```

```
Debug.Print TypeName(Str)    'Output: String
Debug.Print TypeName(IntOne) 'Output: Variant <--- !!!
Debug.Print TypeName(IntTwo) 'Output: Integer
Debug.Print TypeName(Lng)    'Output: Long
```

Variables can also be declared using Data Type Character suffixes ($ % & ! # @), however using these are increasingly discouraged.

```
Dim this$  'String
Dim this%  'Integer
Dim this&  'Long
Dim this!  'Single
Dim this#  'Double
Dim this@  'Currency
```

Other ways of declaring variables are:

- `Static` like: `Static CounterVariable as Integer`

> When you use the Static statement instead of a Dim statement, the declared variable will retain its value between calls.

- `Public` like: `Public CounterVariable as Integer`

> Public variables can be used in any procedures in the project. If a public variable is declared in a standard module or a class module, it can also be used in any projects that reference the project where the public variable is declared.

- `Private` like: `Private CounterVariable as Integer`

> Private variables can be used only by procedures in the same module.

Source and more info:

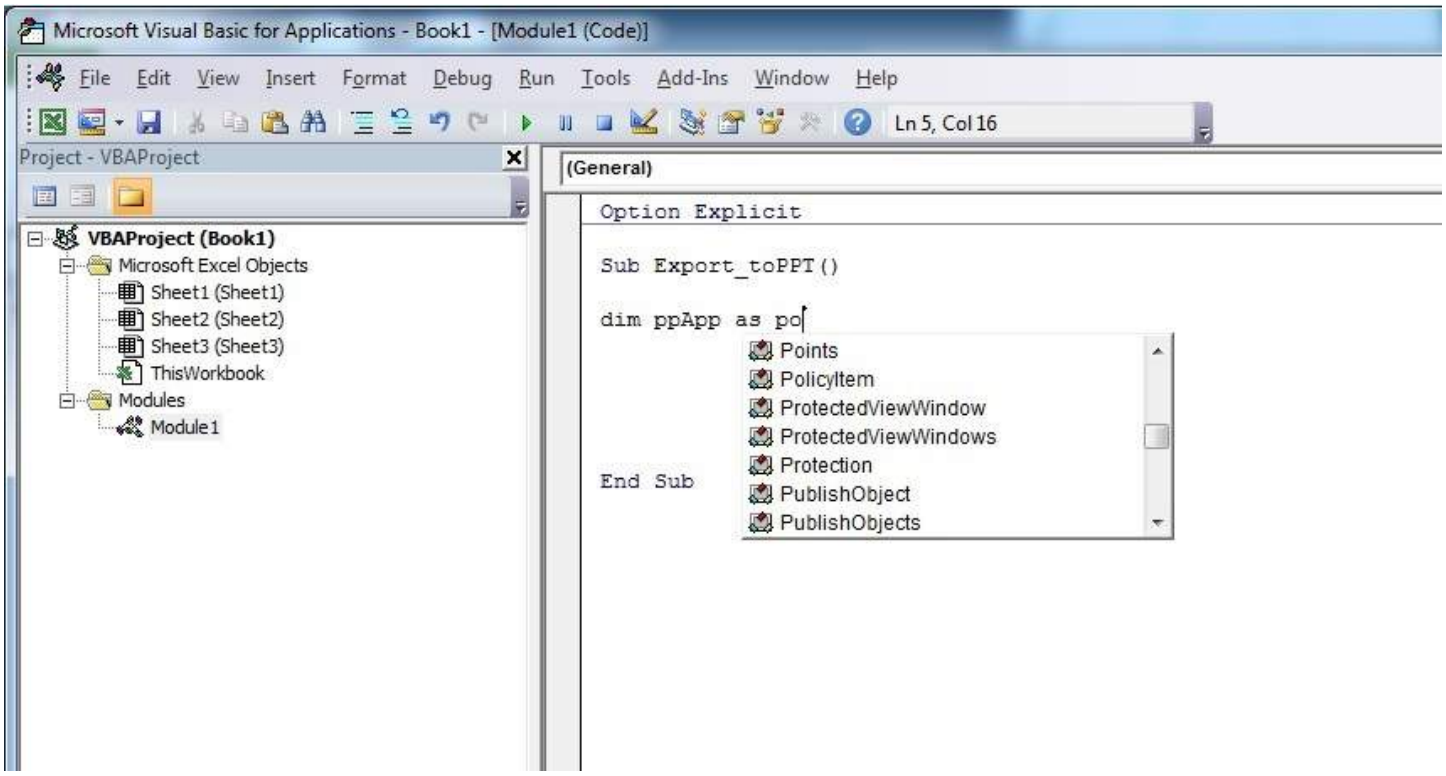
MSDN-Declaring Variables

Type Characters (Visual Basic)

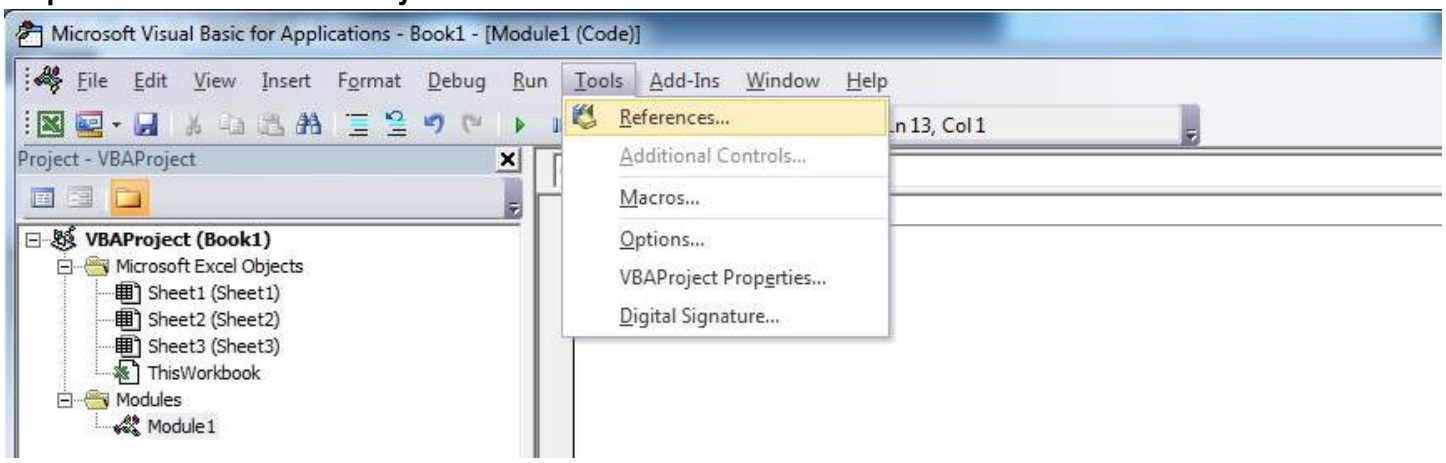
Section 1.3: Adding a new Object Library Reference

The procedure describes how to add an Object library reference, and afterwards how to declare new variables with reference to the new library class objects.

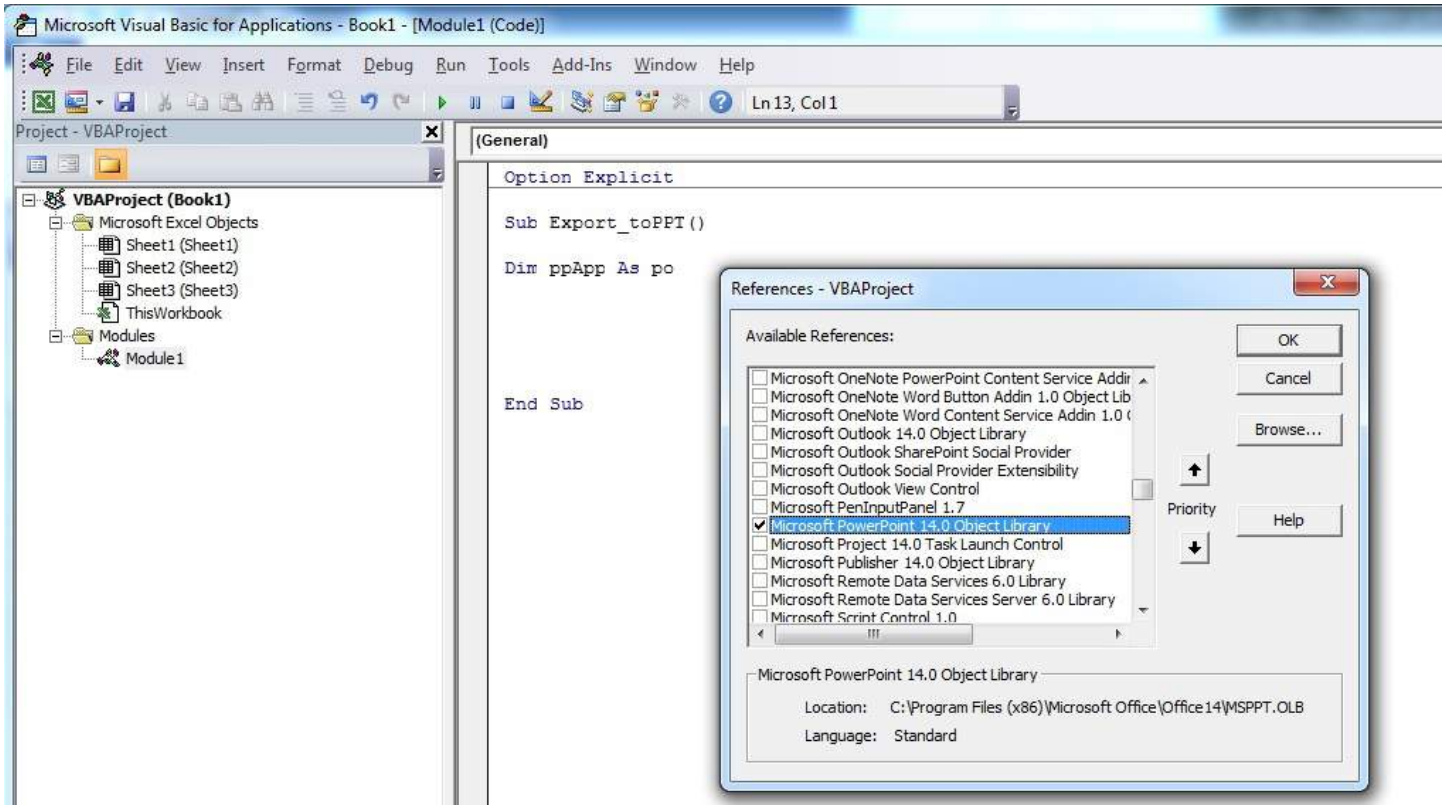
The example below shows how to add the *PowerPoint* library to the existing VB Project. As can be seen, currently the PowerPoint Object library is not available.



Step 1: Select Menu ***Tools --> References***…

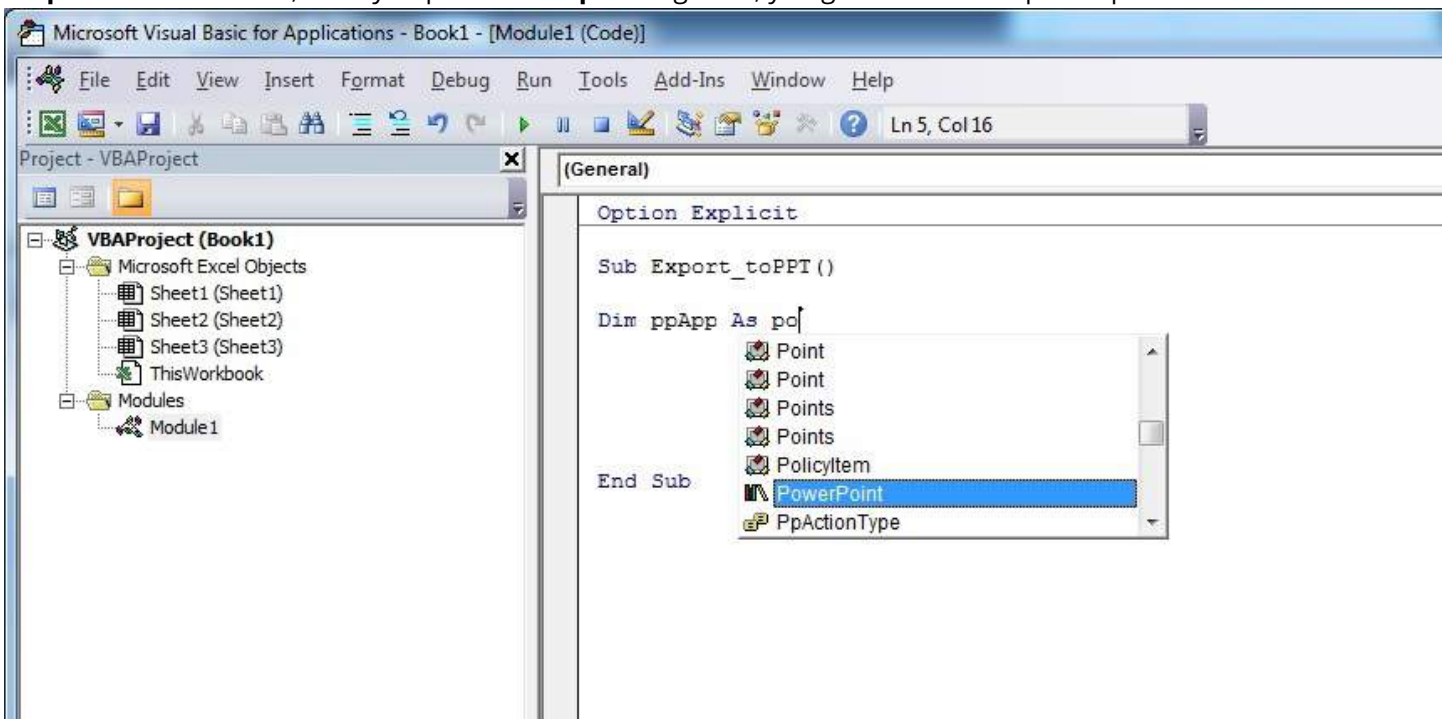


Step 2: Select the Reference you want to add. This example we scroll down to find "***Microsoft PowerPoint 14.0 Object Library***", and then press "**OK**".

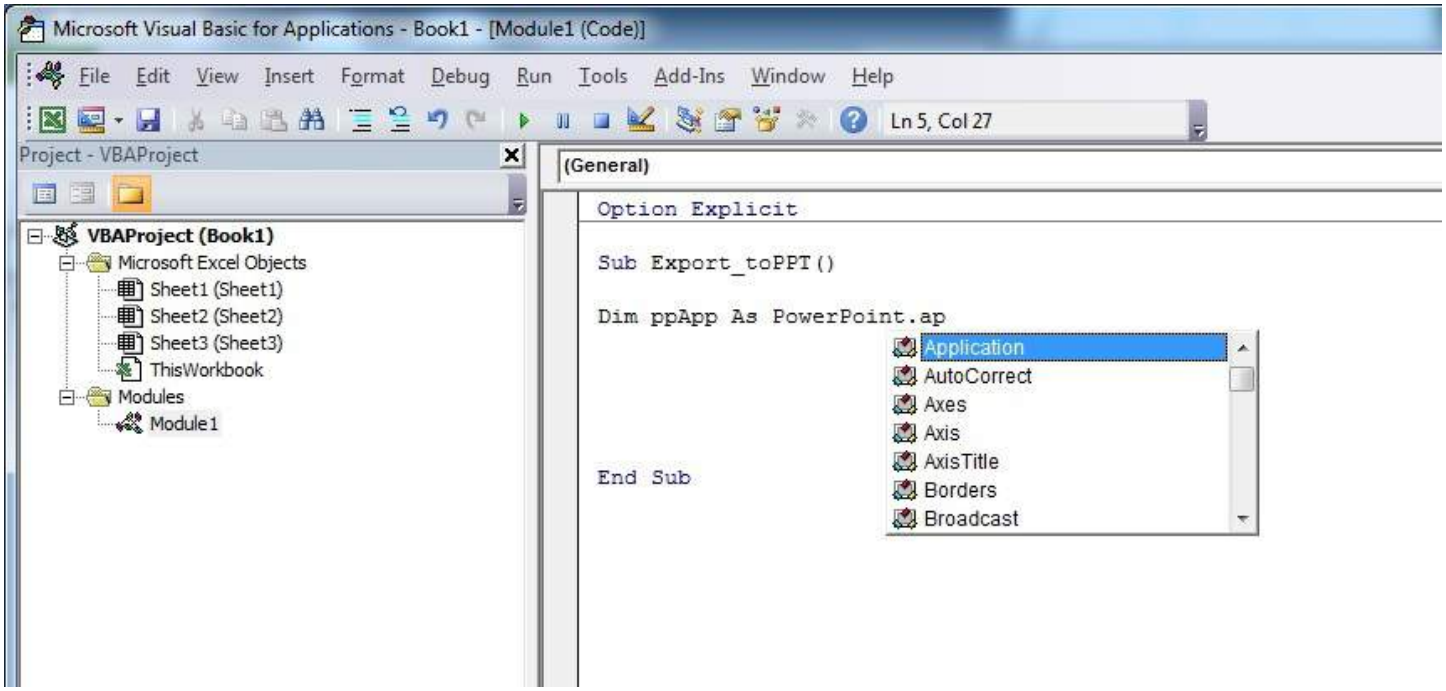


Note: PowerPoint 14.0 means that Office 2010 version is installed on the PC.

Step 3: in the VB Editor, once you press **Ctrl+Space** together, you get the autocomplete option of PowerPoint.

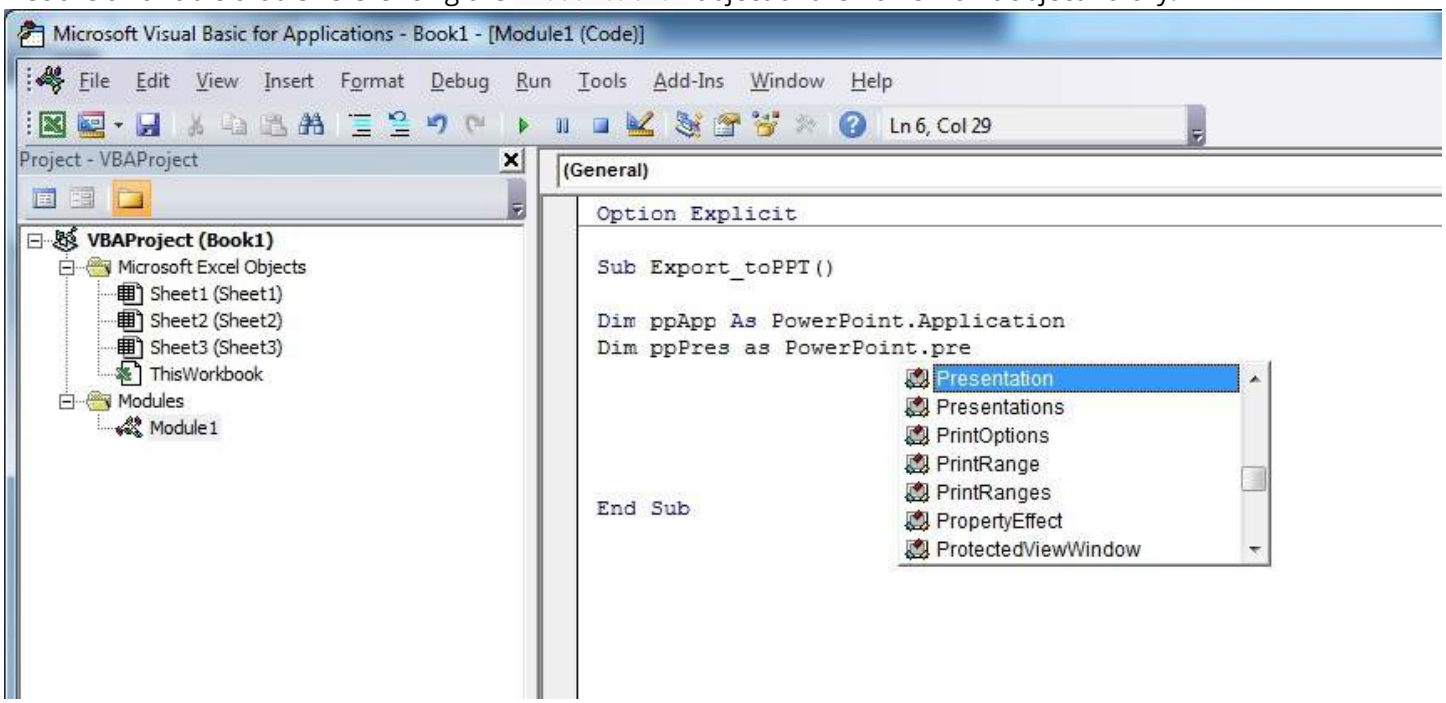


After selecting `PowerPoint` and pressing `.`, another menu appears with all objects options related to the PowerPoint Object Library. This example shows how to select the PowerPoint's object `Application`.

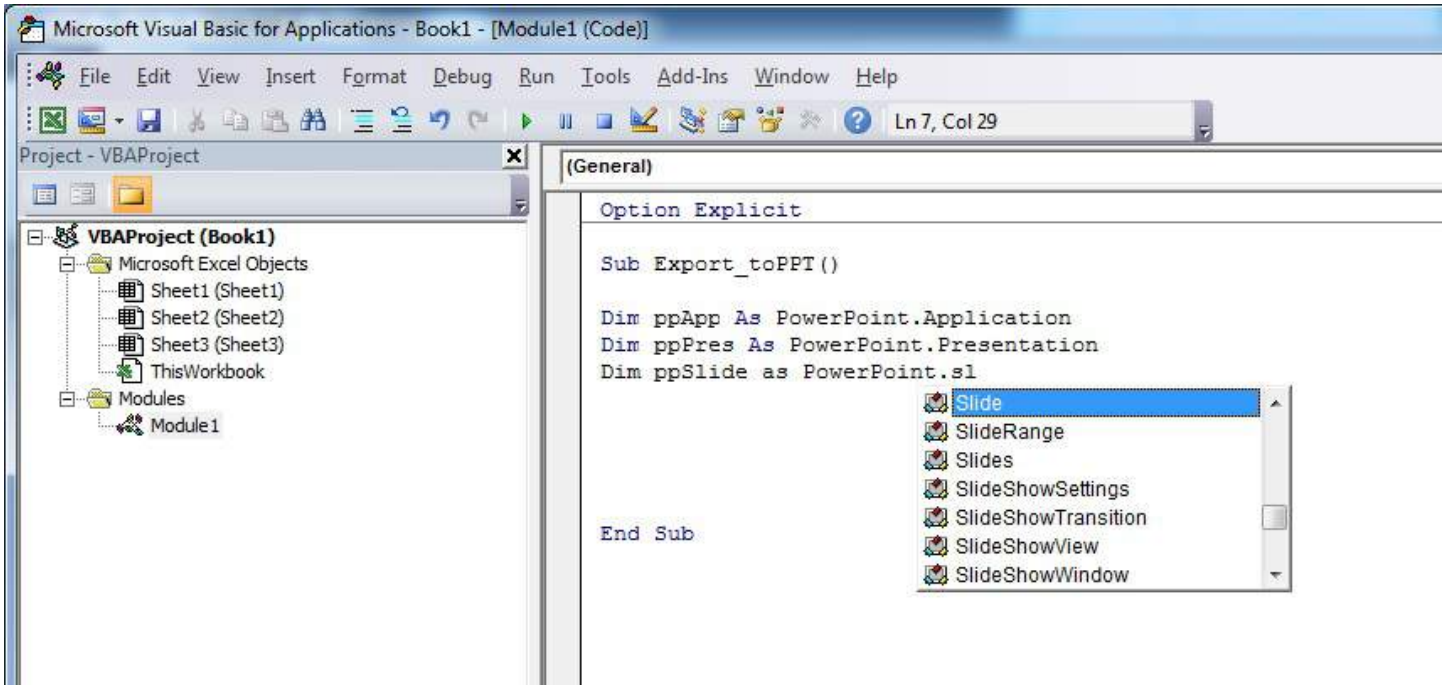


Step 4: Now the user can declare more variables using the PowerPoint object library.

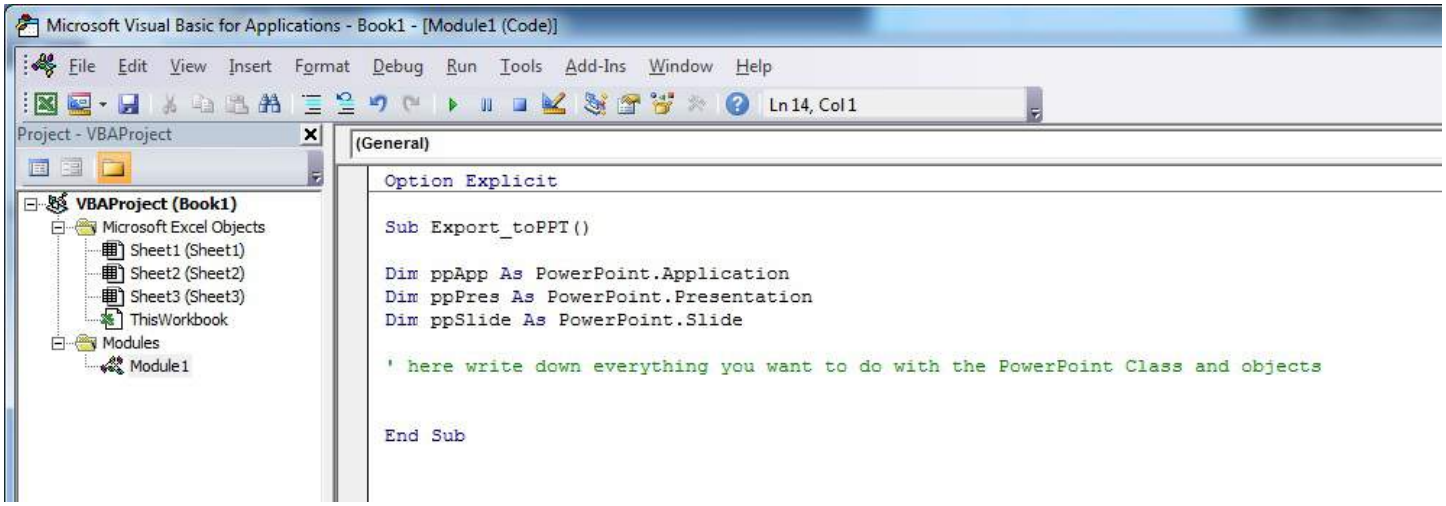
Declare a variable that is referencing the `Presentation` object of the PowerPoint object library.



Declare another variable that is referencing the `Slide` object of the PowerPoint object library.



Now the variables declaration section looks like in the screen-shot below, and the user can start using these variables in his code.

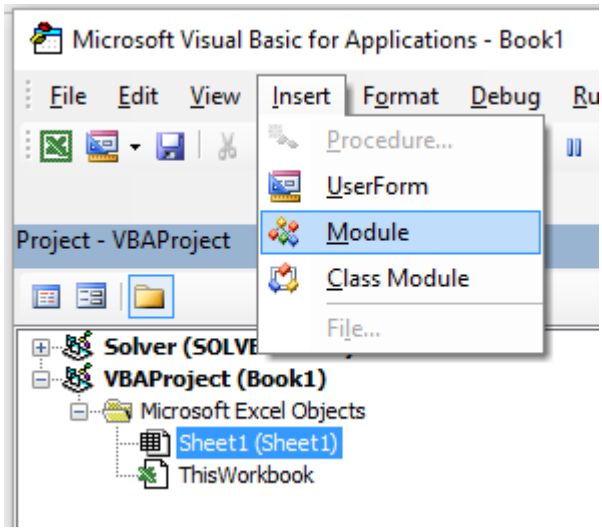


Code version of this tutorial:

```
Option Explicit

Sub Export_toPPT()

Dim ppApp As PowerPoint.Application
Dim ppPres As PowerPoint.Presentation
Dim ppSlide As PowerPoint.Slide

' here write down everything you want to do with the PowerPoint Class and objects


End Sub
```

Section 1.4: Hello World

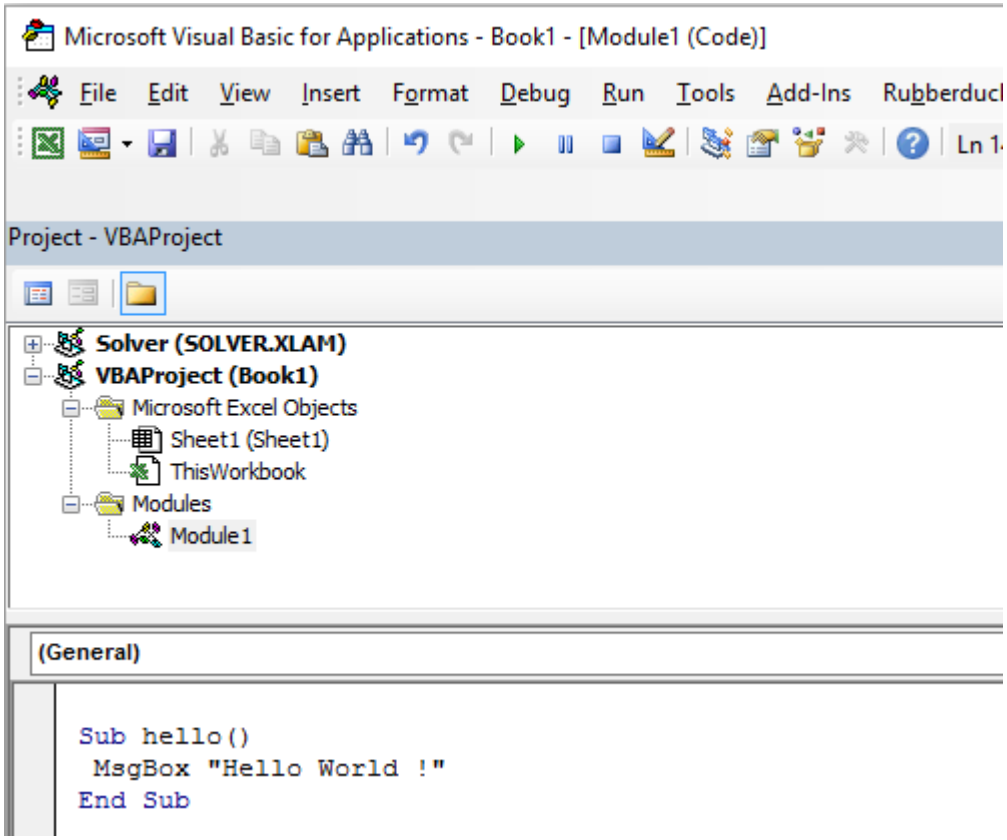
1. Open the Visual Basic Editor (see Opening the Visual Basic Editor)
2. Click Insert --> Module to add a new Module :



3. Copy and Paste the following code in the new module :

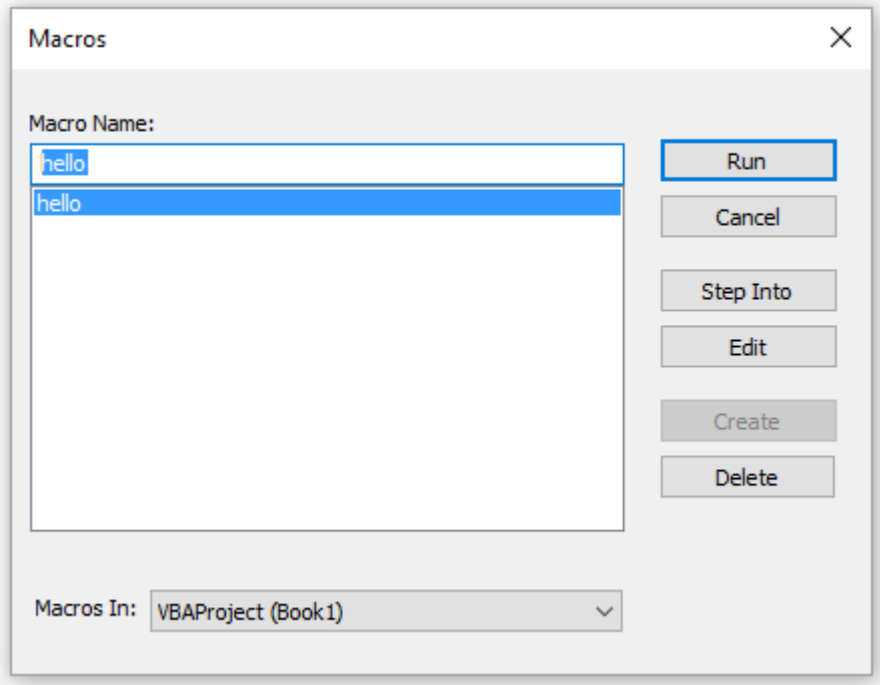
```
Sub hello()
  MsgBox "Hello World !"
End Sub
```

To obtain :

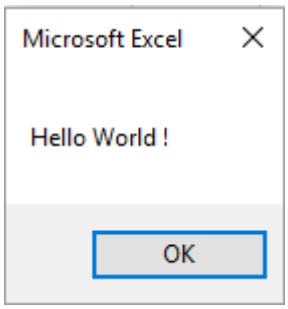


4. Click on the green “play” arrow (or press F5) in the Visual Basic toolbar to run the program:

5. Select the new created sub "hello" and click `Run` :



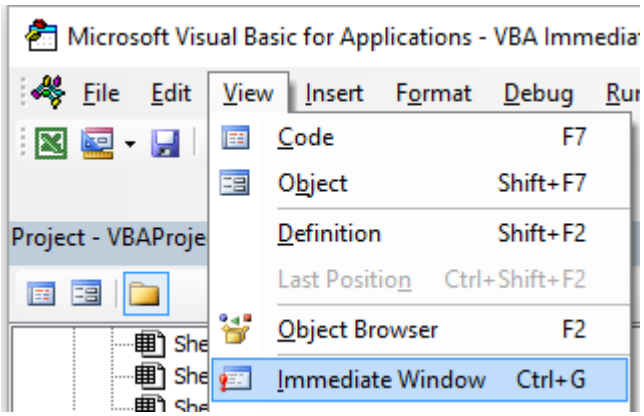
6. Done, your should see the following window:



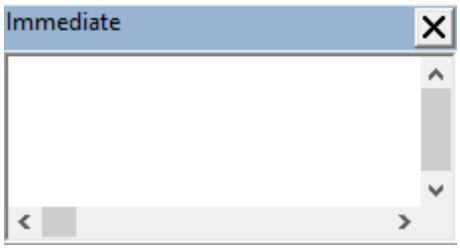
Section 1.5: Getting Started with the Excel Object Model

> This example intend to be a gentle introduction to the Excel Object Model **for beginners**.

1. Open the Visual Basic Editor (VBE)
2. Click View --> Immediate Window to open the Immediate Window (or `ctrl` + `G`):



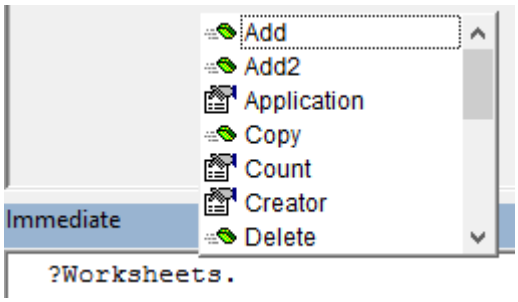
3. You should see the following Immediate Window at the bottom on VBE:



This window allow you to directly test some VBA code. So let's start, type in this console :

```
?Worksheets.
```

VBE has intellisense and then it should open a tooltip as in the following figure :



Select .Count in the list or directly type `.Cout` to obtain :

```
?Worksheets.Count
```

4. Then press Enter. The expression is evaluated and it should returns 1. This indicates the number of Worksheet currently present in the workbook. The question mark (`?`) is an alias for Debug.Print.

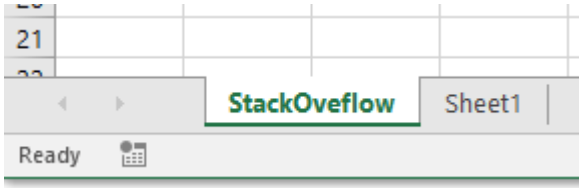
Worksheets is an **Object** and Count is a **Method**. Excel has several Object (`Workbook`, `Worksheet`, `Range`, `Chart` ..) and each of one contains specific methods and properties. You can find the complete list of Object in the Excel VBA reference. Worksheets Object is presented here .

> This Excel VBA reference should become your primary source of information regarding the Excel Object Model.

5. Now let's try another expression, type (without the `?` character):

```
Worksheets.Add().Name = "StackOveflow"
```

6. Press Enter. This should create a new worksheet called `StackOverflow.`:



To understand this expression you need to read the Add function in the aforementioned Excel reference. You will find the following:

```
Add:  Creates a new worksheet, chart, or macro sheet.
The new worksheet becomes the active sheet.
Return Value: An Object value that represents the new worksheet, chart,
```

```
or macro sheet.
```

So the `Worksheets.Add()` create a new worksheet and return it. Worksheet(**without s**) is itself a Object that can be found in the documentation and `Name` is one of its **property** (see here). It is defined as :

```
Worksheet.Name Property:  Returns or sets a String value that
 represents the object name.
```

So, by investigating the different objects definitions we are able to understand this code `Worksheets.Add().Name = "StackOveflow"`.

`Add()` creates and add a new worksheet and return a **reference** to it, then we set its Name **property** to "StackOverflow"

Now let's be more formal, Excel contains several Objects. These Objects may be composed of one or several collection(s) of Excel objects of the same class. It is the case for `WorkSheets` which is a collection of `Worksheet` object. Each Object has some properties and methods that the programmer can interact with.

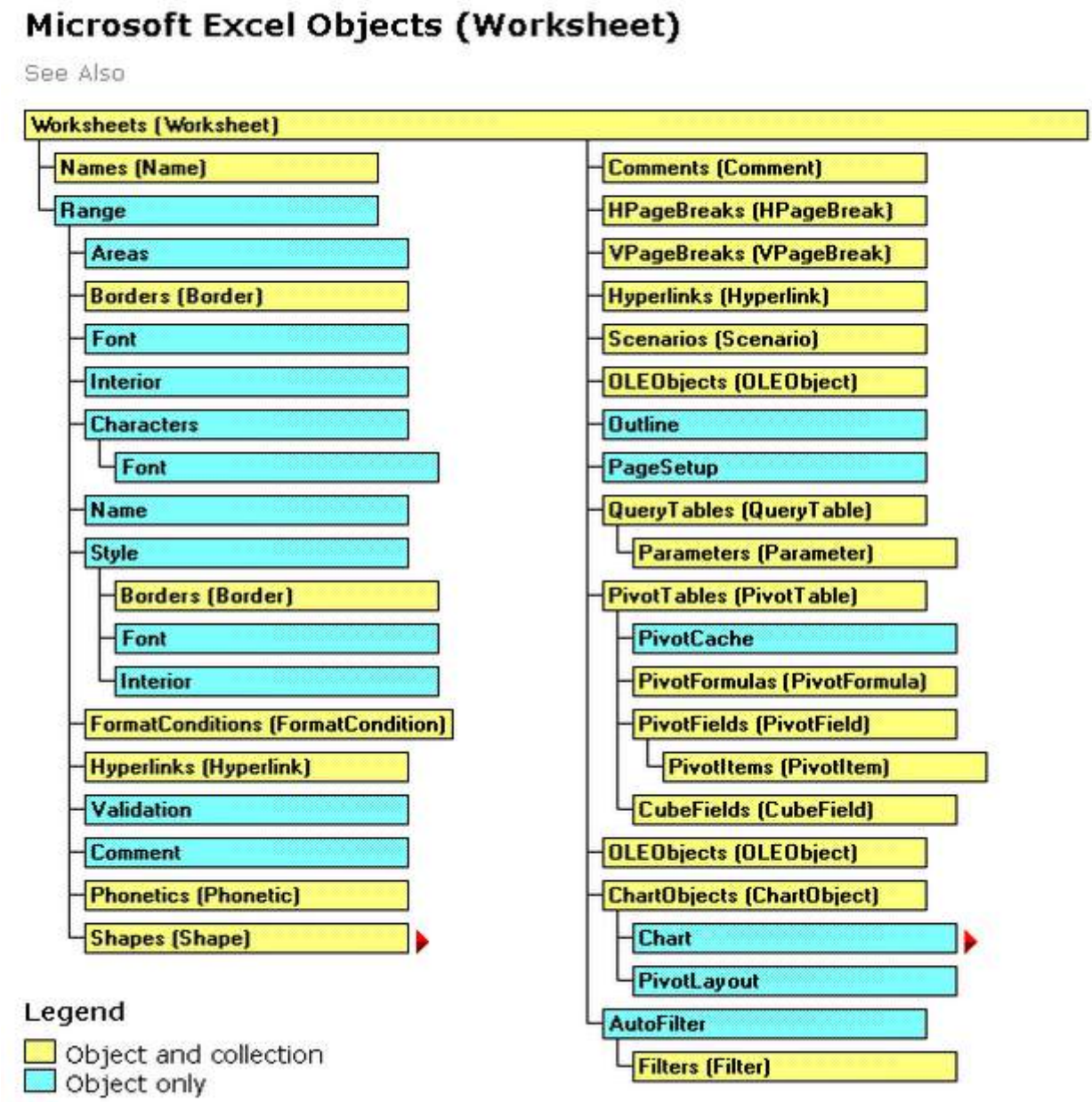
> The Excel Object model refers to the Excel **object hierarchy**

At the top of all objects is the `Application` object, it represents the Excel instance itself. Programming in VBA requires a good understanding of this hierarchy because we always need a reference to an object to be able to call a Method or to Set/Get a property.

The (very simplified) Excel Object Model can be represented as,

```
                        Application
                         Workbooks
                         Workbook
                        Worksheets
                         Worksheet
                          Range
```

A more detail version for the Worksheet Object (as it is in Excel 2007) is shown below,



> The full Excel Object Model can be found here.

Finally some objects may have `events` (ex: `Workbook.WindowActivate`) that are also part of the Excel Object Model.

Chapter 2: Arrays

Section 2.1: Dynamic Arrays (Array Resizing and Dynamic Handling)

Due to not being Excel-VBA exclusive contents this Example has been moved to VBA documentation.

Link: Dynamic Arrays (Array Resizing and Dynamic Handling)

Section 2.2: Populating arrays (adding values)

There are multiple ways to populate an array.

Directly

```
'one-dimensional
Dim arrayDirect1D(2) As String
arrayDirect(0) = "A"
arrayDirect(1) = "B"
arrayDirect(2) = "C"

'multi-dimensional (in this case 3D)
Dim arrayDirectMulti(1, 1, 2)
arrayDirectMulti(0, 0, 0) = "A"
arrayDirectMulti(0, 0, 1) = "B"
arrayDirectMulti(0, 0, 2) = "C"
arrayDirectMulti(0, 1, 0) = "D"
'...
```

Using Array() function

```
'one-dimensional only
Dim array1D As Variant 'has to be type variant
array1D = Array(1, 2, "A")
'-> array1D(0) = 1, array1D(1) = 2, array1D(2) = "A"
```

From range

```
Dim arrayRange As Variant 'has to be type variant

'putting ranges in an array always creates a 2D array (even if only 1 row or column)
'starting at 1 and not 0, first dimension is the row and the second the column
arrayRange = Range("A1:C10").Value
'-> arrayRange(1,1) = value in A1
'-> arrayRange(1,2) = value in B1
'-> arrayRange(5,3) = value in C5
'...

'Yoo can get an one-dimensional array from a range (row or column)
'by using the worksheet functions index and transpose:

'one row from range into 1D-Array:
arrayRange = Application.WorksheetFunction.Index(Range("A1:C10").Value, 3, 0)
'-> row 3 of range into 1D-Array
'-> arrayRange(1) = value in A3, arrayRange(2) = value in B3, arrayRange(3) = value in C3

'one column into 1D-Array:
'limited to 65536 rows in the column, reason: limit of .Transpose
```

```
arrayRange = Application.WorksheetFunction.Index( _
Application.WorksheetFunction.Transpose(Range("A1:C10").Value), 2, 0)
'-> column 2 of range into 1D-Array
'-> arrayRange(1) = value in B1, arrayRange(2) = value in B2, arrayRange(3) = value in B3
'...

'By using Evaluate() - shorthand [] - you can transfer the
'range to an array and change the values at the same time.
'This is equivalent to an array formula in the sheet:
arrayRange = [(A1:C10*3)]
arrayRange = [(A1:C10&"_test")]
arrayRange = [(A1:B10*C1:C10)]
'...
```

2D with Evaluate()

```
Dim array2D As Variant
'[] ist a shorthand for evaluate()
'Arrays defined with evaluate start at 1 not 0
array2D = [{"1A","1B","1C";"2A","2B","3B"}]
'-> array2D(1,1) = "1A", array2D(1,2) = "1B", array2D(2,1) = "2A" ...

'if you want to use a string to fill the 2D-Array:
Dim strValues As String
strValues = "{""1A"",""1B"",""1C"";""2A"",""2B"",""2C""}"
array2D = Evaluate(strValues)
```

Using Split() function

```
Dim arraySplit As Variant 'has to be type variant
arraySplit = Split("a,b,c", ",")
'-> arraySplit(0) = "a", arraySplit(1) = "b", arraySplit(2) = "c"
```

Section 2.3: Jagged Arrays (Arrays of Arrays)

Due to not being Excel-VBA exclusive contents this Example has been moved to VBA documentation.

Link: Jagged Arrays (Arrays of Arrays)

Section 2.4: Check if Array is Initialized (If it contains elements or not)

A common problem might be trying to iterate over Array which has no values in it. For example:

```
Dim myArray() As Integer
For i = 0 To UBound(myArray) 'Will result in a "Subscript Out of Range" error
```

To avoid this issue, and to check if an Array contains elements, use this *oneliner*:

```
If Not Not myArray Then MsgBox UBound(myArray) Else MsgBox "myArray not initialised"
```

Section 2.5: Dynamic Arrays [Array Declaration, Resizing]

```
Sub Array_clarity()

Dim arr() As Variant  'creates an empty array
Dim x As Long
Dim y As Long
```

```
x = Range("A1", Range("A1").End(xlDown)).Cells.Count
y = Range("A1", Range("A1").End(xlToRight)).Cells.Count

ReDim arr(0 To x, 0 To y) 'fixing the size of the array

For x = LBound(arr, 1) To UBound(arr, 1)
    For y = LBound(arr, 2) To UBound(arr, 2)
        arr(x, y) = Range("A1").Offset(x, y) 'storing the value of Range("A1:E10") from activesheet
in x and y variables
    Next
Next

'Put it on the same sheet according to the declaration:
Range("A14").Resize(UBound(arr, 1), UBound(arr, 2)).Value = arr

End Sub
```

Chapter 3: Conditional statements

Section 3.1: The If statement

The `If` control statement allows different code to be executed depending upon the evaluation of a conditional (Boolean) statement. A conditional statement is one that evaluates to either `True` or `False`, e.g. `x > 2`.

There are three patterns that can be used when implementing an `If` statement, which are described below. Note that an `If` conditional evaluation is always followed by a `Then`.

1. Evaluating one `If` conditional statement and doing something if it is `True`

Single line `If` statement

This is the shortest way to use an `If` and it is useful when only one statement needs to be carried out upon a `True` evaluation. When using this syntax, all of the code must be on a single line. Do not include an `End If` at the end of the line.

```
If [Some condition is True] Then [Do something]
```

`If` block

If multiple lines of code need to be executed upon a `True` evaluation, an `If` block may be used.

```
If [Some condition is True] Then
  [Do some things]
End If
```

Note that, if a multi-line `If` block is used, a corresponding `End If` is required.

2. Evaluating one conditional `If` statement, doing one thing if it is `True` and doing something else if it is `False`

Single line `If`, `Else` statement

This may be used if one statement is to be carried out upon a `True` evaluation and a different statement is to be carried out on a `False` evaluation. Be careful using this syntax, as it is often less clear to readers that there is an `Else` statement. When using this syntax, all of the code must be on a single line. Do not include an `End If` at the end of the line.

```
If [Some condition is True] Then [Do something] Else [Do something else]
```

`If`, `Else` block

Use an `If`, `Else` block to add clarity to your code, or if multiple lines of code need to be executed under either a `True` or a `False` evaluation.

```
If [Some condition is True] Then
   [Do some things]
Else
   [Do some other things]
End If
```

Note that, if a multi-line `If` block is used, a corresponding `End If` is required.

3. Evaluating many conditional statements, when preceding statements are all `False`, and doing something different for each one

This pattern is the most general use of `If` and would be used when there are many non-overlapping conditions that require different treatment. Unlike the first two patterns, this case requires the use of an `If` block, even if only one line of code will be executed for each condition.

`If`, `ElseIf`, `...`, `Else` block

Instead of having to create many `If` blocks one below another, an `ElseIf` may be used evaluate an extra condition. The `ElseIf` is only evaluated if any preceding `If` evaluation is `False`.

```
If [Some condition is True] Then
   [Do some thing(s)]
ElseIf [Some other condition is True] Then
   [Do some different thing(s)]
Else    'Everything above has evaluated to False
   [Do some other thing(s)]
End If
```

As many `ElseIf` control statements may be included between an `If` and an `End If` as required. An `Else` control statement is not required when using `ElseIf` (although it is recommended), but if it is included, it must be the final control statement before the `End If`.

Chapter 4: Ranges and Cells

Section 4.1: Ways to refer to a single cell

The simplest way to refer to a single cell on the current Excel worksheet is simply to enclose the A1 form of its reference in square brackets:

```
[a3] = "Hello!"
```

Note that square brackets are just convenient syntactic sugar for the `Evaluate` method of the `Application` object, so technically, this is identical to the following code:

```
Application.Evaluate("a3") = "Hello!"
```

You could also call the `Cells` method which takes a row and a column and returns a cell reference.

```
Cells(3, 1).Formula = "=A1+A2"
```

Remember that whenever you pass a row and a column to Excel from VBA, the row is always first, followed by the column, which is confusing because it is the opposite of the common `A1` notation where the column appears first.

In both of these examples, we did not specify a worksheet, so Excel will use the active sheet (the sheet that is in front in the user interface). You can specify the active sheet explicitly:

```
ActiveSheet.Cells(3, 1).Formula = "=SUM(A1:A2)"
```

Or you can provide the name of a particular sheet:

```
Sheets("Sheet2").Cells(3, 1).Formula = "=SUM(A1:A2)"
```

There are a wide variety of methods that can be used to get from one range to another. For example, the `Rows` method can be used to get to the individual rows of any range, and the `Cells` method can be used to get to individual cells of a row or column, so the following code refers to cell C1:

```
ActiveSheet.Rows(1).Cells(3).Formula = "hi!"
```

Section 4.2: Creating a Range

A Range cannot be created or populated the same way a string would:

```
Sub RangeTest()
    Dim s As String
    Dim r As Range 'Specific Type of Object, with members like Address, WrapText, AutoFill, etc.

    ' This is how we fill a String:
    s = "Hello World!"

    ' But we cannot do this for a Range:
    r = Range("A1") '//Run. Err.:  91 Object variable or With block variable not set//

    ' We have to use the Object approach, using keyword Set:
    Set r = Range("A1")
End Sub
```

It is considered best practice to qualify your references, so from now on we will use the same approach here. More about Creating Object Variables (e.g. Range) on MSDN . More about Set Statement on MSDN.

There are different ways to create the same Range:

```
Sub SetRangeVariable()
    Dim ws As Worksheet
    Dim r As Range

    Set ws = ThisWorkbook.Worksheets(1) ' The first Worksheet in Workbook with this code in it

    ' These are all equivalent:
    Set r = ws.Range("A2")
    Set r = ws.Range("A" & 2)
    Set r = ws.Cells(2, 1) ' The cell in row number 2, column number 1
    Set r = ws.[A2] 'Shorthand notation of Range.
    Set r = Range("NamedRangeInA2") 'If the cell A2 is named NamedRangeInA2. Note, that this is
Sheet independent.
    Set r = ws.Range("A1").Offset(1, 0) ' The cell that is 1 row and 0 columns away from A1
    Set r = ws.Range("A1").Cells(2,1) ' Similar to Offset. You can "go outside" the original Range.

    Set r = ws.Range("A1:A5").Cells(2) 'Second cell in bigger Range.
    Set r = ws.Range("A1:A5").Item(2) 'Second cell in bigger Range.
    Set r = ws.Range("A1:A5")(2) 'Second cell in bigger Range.
End Sub
```

Note in the example that Cells(2, 1) is equivalent to Range("A2"). This is because Cells returns a Range object. Some sources: Chip Pearson-Cells Within Ranges; MSDN-Range Object; John Walkenback-Referring To Ranges In Your VBA Code.

Also note that in any instance where a number is used in the declaration of the range, and the number itself is outside of quotation marks, such as Range("A" & 2), you can swap that number for a variable that contains an integer/long. For example:

```
Sub RangeIteration()
    Dim wb As Workbook, ws As Worksheet
    Dim r As Range

    Set wb = ThisWorkbook
    Set ws = wb.Worksheets(1)

    For i = 1 To 10
        Set r = ws.Range("A" & i)
        ' When i = 1, the result will be Range("A1")
        ' When i = 2, the result will be Range("A2")
        ' etc.
        ' Proof:
        Debug.Print r.Address
    Next i
End Sub
```

If you are using double loops, Cells is better:

```
Sub RangeIteration2()
    Dim wb As Workbook, ws As Worksheet
    Dim r As Range

    Set wb = ThisWorkbook
    Set ws = wb.Worksheets(1)
```

```
    For i = 1 To 10
        For j = 1 To 10
            Set r = ws.Cells(i, j)
            ' When i = 1 and j = 1, the result will be Range("A1")
            ' When i = 2 and j = 1, the result will be Range("A2")
            ' When i = 1 and j = 2, the result will be Range("B1")
            ' etc.
            ' Proof:
            Debug.Print r.Address
        Next j
    Next i
End Sub
```

Section 4.3: Offset Property

- **Offset(Rows, Columns)** - The operator used to statically reference another point from the current cell. Often used in loops. It should be understood that positive numbers in the rows section moves right, wheres as negatives move left. With the columns section positives move down and negatives move up.

i.e

```
Private Sub this()
    ThisWorkbook.Sheets("Sheet1").Range("A1").Offset(1, 1).Select
    ThisWorkbook.Sheets("Sheet1").Range("A1").Offset(1, 1).Value = "New Value"
    ActiveCell.Offset(-1, -1).Value = ActiveCell.Value
    ActiveCell.Value = vbNullString
End Sub
```

This code selects B2, puts a new string there, then moves that string back to A1 afterwards clearing out B2.

Section 4.4: Saving a reference to a cell in a variable

To save a reference to a cell in a variable, you must use the `Set` syntax, for example:

```
Dim R as Range
Set R = ActiveSheet.Cells(3, 1)
```

later...

```
R.Font.Color = RGB(255, 0, 0)
```

Why is the `Set` keyword required? `Set` tells Visual Basic that the value on the right hand side of the = is meant to be an object.

Section 4.5: How to Transpose Ranges (Horizontal to Vertical & vice versa)

```
Sub TransposeRangeValues()
    Dim TmpArray() As Variant, FromRange as Range, ToRange as Range

    set FromRange = Sheets("Sheet1").Range("a1:a12")          'Worksheets(1).Range("a1:p1")
    set ToRange = ThisWorkbook.Sheets("Sheet1").Range("a1")
 'ThisWorkbook.Sheets("Sheet1").Range("a1")

    TmpArray = Application.Transpose(FromRange.Value)
    FromRange.Clear
```

```
    ToRange.Resize(FromRange.Columns.Count,FromRange.Rows.Count).Value2 = TmpArray
End Sub
```

Note: Copy/PasteSpecial also has a Paste Transpose option which updates the transposed cells' formulas as well.

Chapter 5: Named Ranges

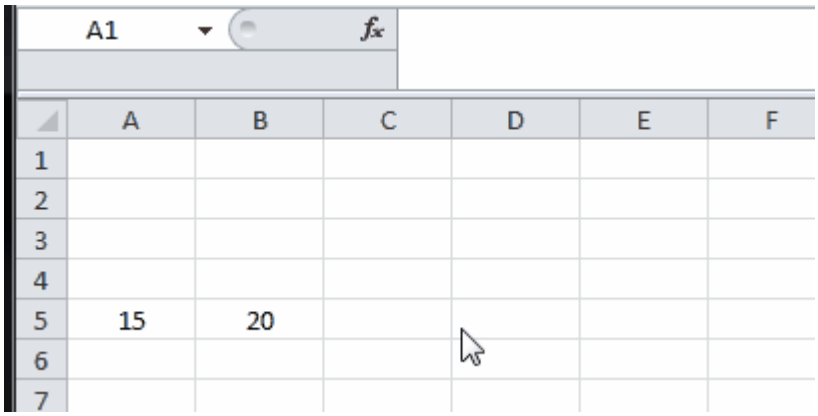
Topic should include information specifically related to named ranges in Excel including methods for creating, modifying, deleting, and accessing defined named ranges.

Section 5.1: Define A Named Range

Using named ranges allows you to describe the meaning of a cell(s) contents and use this defined name in place of an actual cell address.

For example, formula `=A5*B5` can be replaced with `=Width*Height` to make the formula much easier to read and understand.

To define a new named range, select cell or cells to name and then type new name into the Name Box next to the formula bar.



> Note: Named Ranges default to global scope meaning that they can be accessed from anywhere within the workbook. Older versions of Excel allow for duplicate names so care must be taken to prevent duplicate names of global scope otherwise results will be unpredictable. Use Name Manager from Formulas tab to change scope.

Section 5.2: Using Named Ranges in VBA

Create new named range called 'MyRange' assigned to cell `A1`

```
ThisWorkbook.Names.Add Name:="MyRange", _
    RefersTo:=Worksheets("Sheet1").Range("A1")
```

Delete defined named range by name

```
ThisWorkbook.Names("MyRange").Delete
```

Access Named Range by name

```
Dim rng As Range
Set rng = ThisWorkbook.Worksheets("Sheet1").Range("MyRange")
Call MsgBox("Width = " & rng.Value)
```

Access a Named Range with a Shortcut

Just like any other range, named ranges can be accessed directly with through a shortcut notation that does not require a `Range` object to be created. The three lines from the code excerpt above can be replaced by a single line:

```
Call MsgBox("Width = " & [MyRange])
```

> Note: The default property for a Range is its Value, so `[MyRange]` is the same as `[MyRange].Value`

You can also call methods on the range. The following selects `MyRange`:

```
[MyRange].Select
```

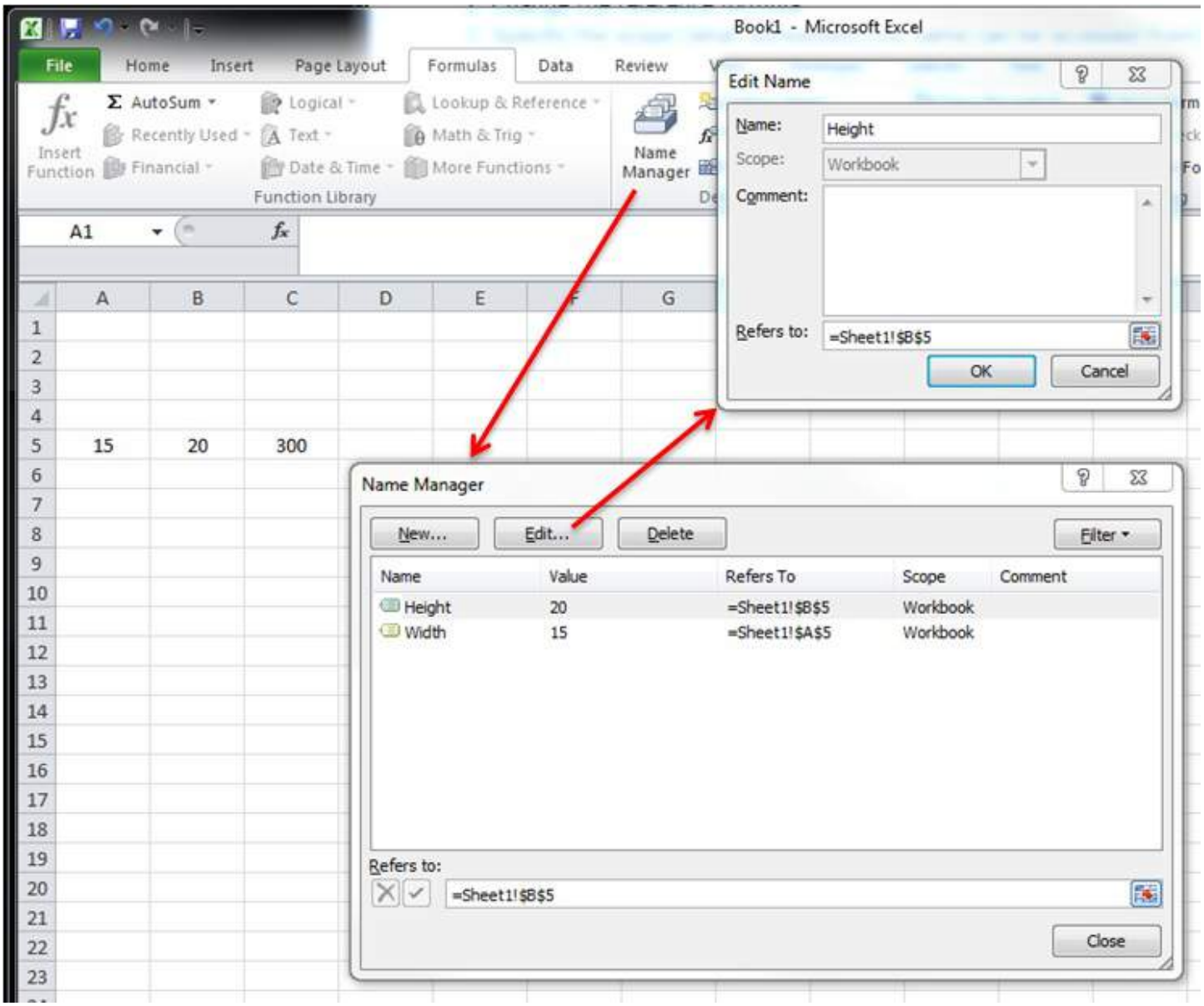
> Note: One caveat is that the shortcut notation does not work with words that are used elsewhere in the VBA library. For example, a range named `Width` would not be accessible as `[Width]` but would work as expected if accessed through `ThisWorkbook.Worksheets("Sheet1").Range("Width")`

Section 5.3: Manage Named Range(s) using Name Manager

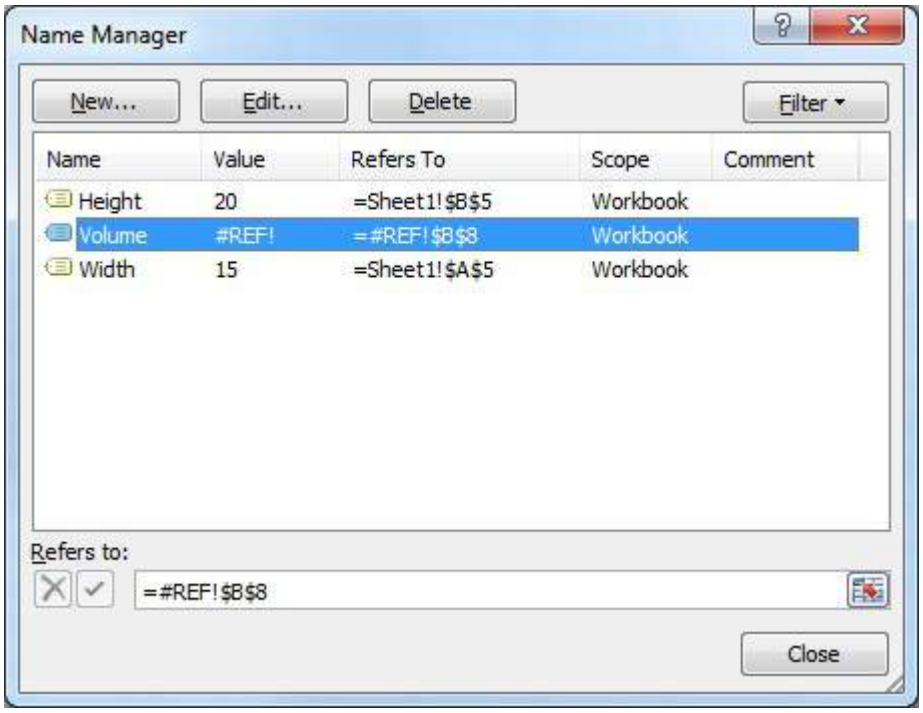
Formulas tab > Defined Names group > Name Manager button

Named Manager allows you to:

1. Create or change name
2. Create or change cell reference
3. Create or change scope
4. Delete existing named range

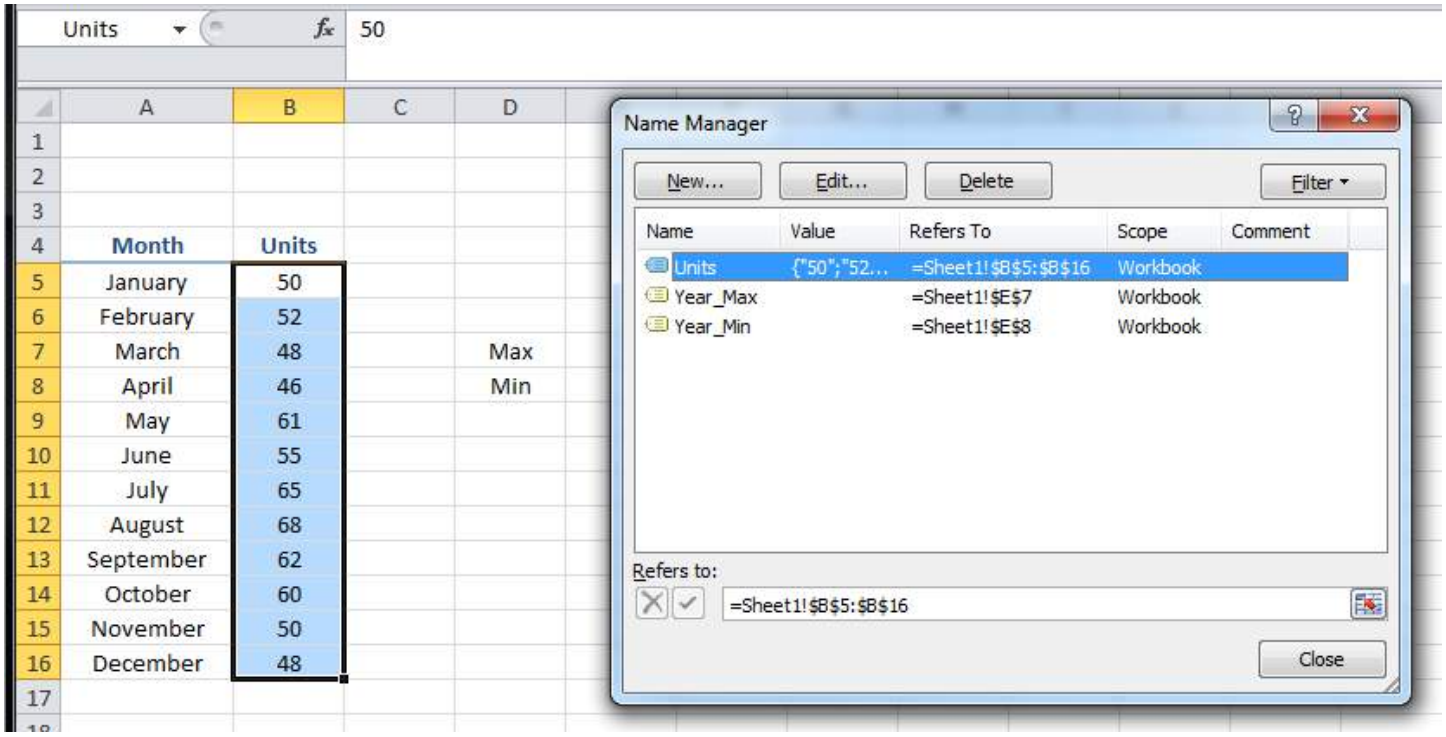


Named Manager provides a useful quick look for broken links.



Section 5.4: Named Range Arrays

Example sheet



Code

```
Sub Example()
    Dim wks As Worksheet
    Set wks = ThisWorkbook.Worksheets("Sheet1")

    Dim units As Range
    Set units = ThisWorkbook.Names("Units").RefersToRange

    Worksheets("Sheet1").Range("Year_Max").Value = WorksheetFunction.Max(units)
    Worksheets("Sheet1").Range("Year_Min").Value = WorksheetFunction.Min(units)
End Sub
```

Result

Month	Units			
January	50			
February	52			
March	48		Max	68
April	46		Min	46
May	61			
June	55			
July	65			
August	68			
September	62			
October	60			
November	50			
December	48			

Chapter 6: Merged Cells / Ranges

Section 6.1: Think twice before using Merged Cells/Ranges

First of all, Merged Cells are there only to improve the look of your sheets.

So it is literally the last thing that you should do, once your sheet and workbook are totally functional!

Where is the data in a Merged Range?

When you merge a Range, you'll only display one block.

The data will be in the very **first cell of that Range**, and the **others will be empty cells**!

One good point about it : no need to fill all the cells or the range once merged, just fill the first cell! ;)

The other aspects of this merged ranged are globally negative :

- If you use a method for finding last row or column, you'll risk some errors
- If you loop through rows and you have merged some ranges for a better readability, you'll encounter empty cells and not the value displayed by the merged range

Chapter 7: Locating duplicate values in a range

At certain points, you will be evaluating a range of data and you will need to locate the duplicates in it. For bigger data sets, there are a number of approaches you can take that use either VBA code or conditional functions. This example uses a simple if-then condition within two nested for-next loops to test whether each cell in the range is equal in value to any other cell in the range.

Section 7.1: Find duplicates in a range

The following tests range A2 to A7 for duplicate values. **Remark:** This example illustrates a possible solution as a first approach to a solution. It's faster to use an array than a range and one could use collections or dictionaries or xml methods to check for duplicates.

```
    Sub find_duplicates()
' Declare variables
  Dim ws     As Worksheet               ' worksheet
  Dim cell   As Range                   ' cell within worksheet range
  Dim n      As Integer                 ' highest row number
  Dim bFound As Boolean                 ' boolean flag, if duplicate is found
  Dim sFound As String: sFound = "|"    ' found duplicates
  Dim s      As String                  ' message string
  Dim s2     As String                  ' partial message string
' Set Sheet to memory
  Set ws = ThisWorkbook.Sheets("Duplicates")

' loop thru FULLY QUALIFIED REFERENCE
  For Each cell In ws.Range("A2:A7")
    bFound = False: s2 = ""             ' start each cell with empty values
 '  Check if first occurrence of this value as duplicate to avoid further searches
    If InStr(sFound, "|" & cell & "|") = 0 Then

      For n = cell.Row + 1 To 7           ' iterate starting point to avoid REDUNDANT SEARCH
        If cell = ws.Range("A" & n).Value Then
           If cell.Row <> n Then        ' only other cells, as same cell cannot be a duplicate
                 bFound = True           ' boolean flag
              '  found duplicates in cell A{n}
                 s2 = s2 & vbNewLine & " -> duplicate in A" & n
           End If
        End If
       Next
     End If
   ' notice all found duplicates
     If bFound Then
         ' add value to list of all found duplicate values
         ' (could be easily split to an array for further analyze)
           sFound = sFound & cell & "|"
           s = s & cell.Address & " (value=" & cell & ")" & s2 & vbNewLine & vbNewLine
     End If
   Next
' Messagebox with final result
  MsgBox "Duplicate values are " & sFound & vbNewLine & vbNewLine & s, vbInformation, "Found
duplicates"
End Sub
```

Depending on your needs, the example can be modified - for instance, the upper limit of n can be the row value of last cell with data in the range, or the action in case of a True If condition can be edited to extract the duplicate

value somewhere else. However, the mechanics of the routine would not change.

Chapter 8: User Defined Functions (UDFs)

Section 8.1: Allow full column references without penalty

It's easier to implement some UDFs on the worksheet if full column references can be passed in as parameters. However, due to the explicit nature of coding, any loop involving these ranges may be processing hundreds of thousands of cells that are completely empty. This reduces your VBA project (and workbook) to a frozen mess while unnecessary non-values are processed.

Looping through a worksheet's cells is one of the slowest methods of accomplishing a task but sometimes it is unavoidable. Cutting the work performed down to what is actually required makes perfect sense.

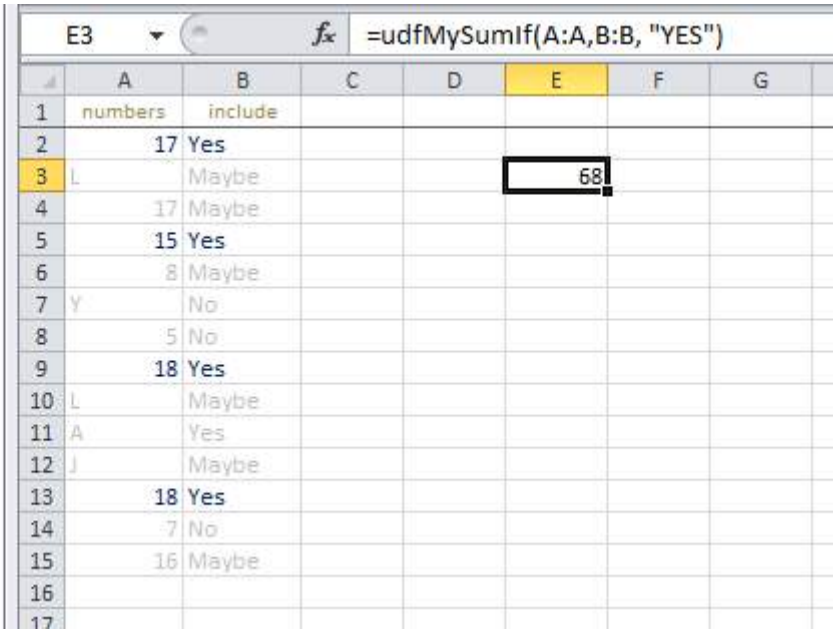
The solution is to truncate the full column or full row references to the Worksheet.UsedRange property with the Intersect method. The following sample will loosely replicate a worksheet's native SUMIF function so the *criteria_range* will also be resized to suit the *sum_range* since each value in the *sum_range* must be accompanied by a value in the *criteria_range*.

The Application.Caller for a UDF used on a worksheet is the cell in which it resides. The cell's .Parent property is the worksheet. This will be used to define the .UsedRange.

In a Module code sheet:

```
Option Explicit

Function udfMySumIf(rngA As Range, rngB As Range, _
                    Optional crit As Variant = "yes")
    Dim c As Long, ttl As Double

    With Application.Caller.Parent
        Set rngA = Intersect(rngA, .UsedRange)
        Set rngB = rngB.Resize(rngA.Rows.Count, rngA.Columns.Count)
    End With

    For c = 1 To rngA.Cells.Count
        If IsNumeric(rngA.Cells(c).Value2) Then
            If LCase(rngB(c).Value2) = LCase(crit) Then
                ttl = ttl + rngA.Cells(c).Value2
            End If
        End If
    Next c

    udfMySumIf = ttl

End Function
```

> Syntax:
>
> `=udfMySumIf(*sum_range*, *criteria_range*, [*criteria*])`



While this is a fairly simplistic example, it adequately demonstrates passing in two full column references (1,048,576 rows each) but only processing 15 rows of data and criteria.

Linked official MSDN documentation of individual methods and properties courtesy of Microsoft™.

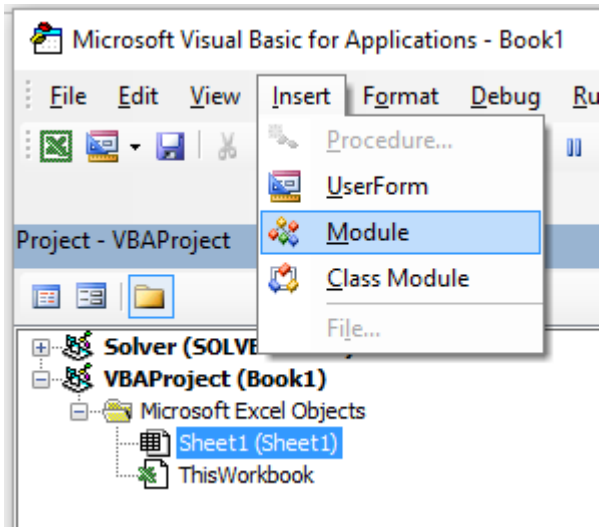
Section 8.2: Count Unique values in Range

```
Function countUnique(r As range) As Long
    'Application.Volatile False ' optional
    Set r = Intersect(r, r.Worksheet.UsedRange) ' optional if you pass entire rows or columns to the function
    Dim c As New Collection, v
    On Error Resume Next    ' to ignore the Run-time error 457: "This key is already associated with an element of this collection".
    For Each v In r.Value  ' remove .Value for ranges with more than one Areas
        c.Add 0, v & ""
    Next
    c.Remove "" ' optional to exclude blank values from the count
    countUnique = c.Count
End Function
```

Collections

Section 8.3: UDF - Hello World

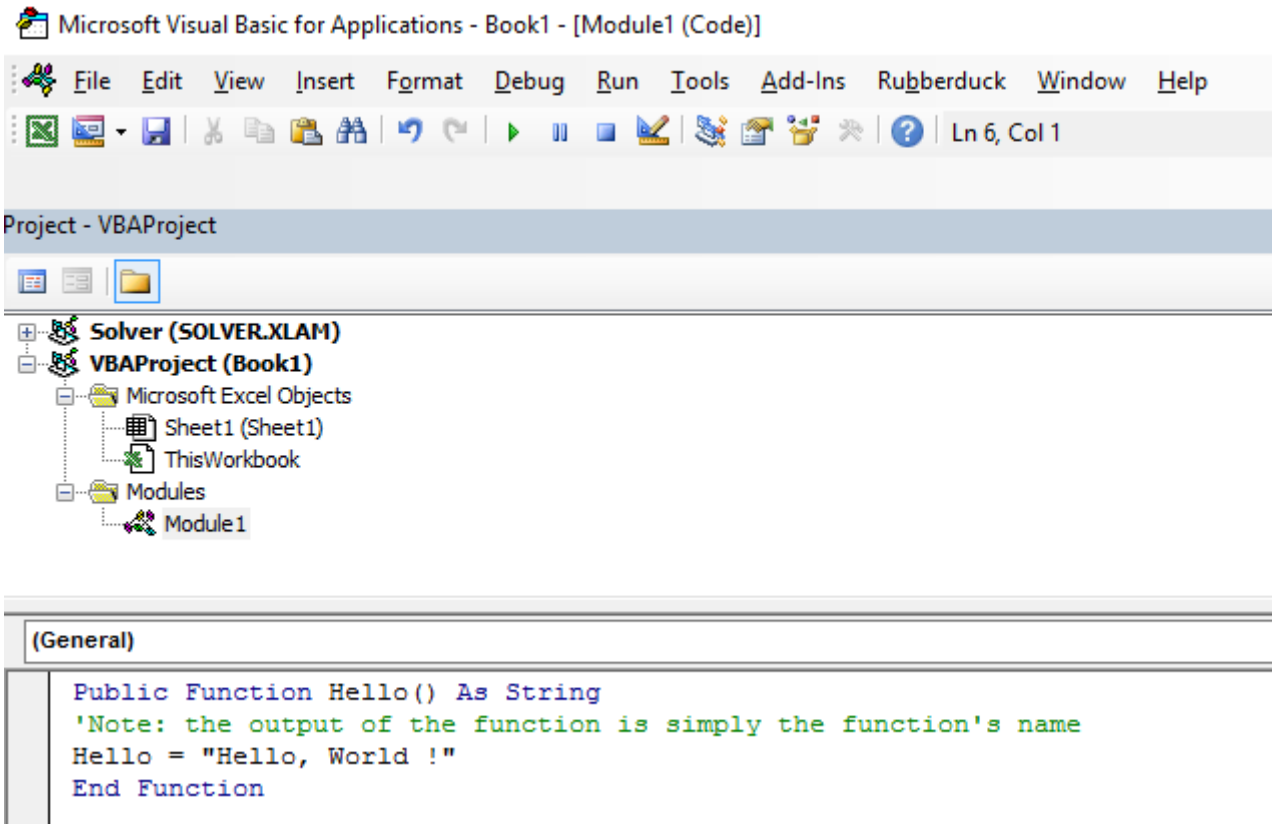
1. Open Excel
2. Open the Visual Basic Editor (see Opening the Visual Basic Editor)
3. Add a new module by clicking Insert --> Module :



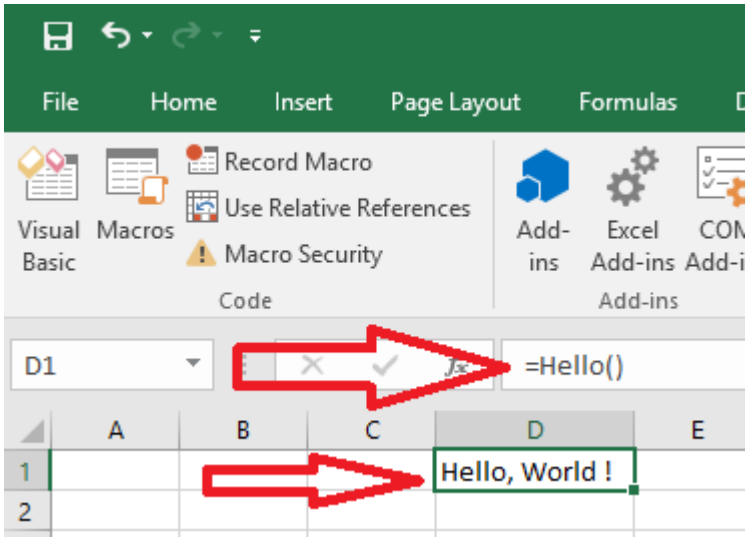
4. Copy and Paste the following code in the new module :

```
Public Function Hello() As String
'Note: the output of the function is simply the function's name
Hello = "Hello, World !"
End Function
```

To obtain :



5. Go back to your workbook and type "=Hello()" into a cell to see the "Hello World".


File
Home
Insert
Page Layout
Formulas
Record Macro
Use Relative References
Macro Security
Visual Basic
Macros
Code
Add-ins
Excel Add-ins
Add-ins
D1
=Hello()
A
B
C
D
E
1
2
Hello, World !

Chapter 9: Conditional formatting using VBA

Section 9.1: FormatConditions.Add

Syntax:

```
FormatConditions.Add(Type, Operator, Formula1, Formula2)
```

Parameters:

Name	Required / Optional	Data Type
Type	Required	XlFormatConditionType
Operator	Optional	Variant
Formula1	Optional	Variant
Formula2	Optional	Variant

XlFormatConditionType enumaration:

Name	Description
xlAboveAverageCondition	Above average condition
xlBlanksCondition	Blanks condition
xlCellValue	Cell value
xlColorScale	Color scale
xlDatabar	Databar
xlErrorsCondition	Errors condition
xlExpression	Expression
XlIconSet	Icon set
xlNoBlanksCondition	No blanks condition
xlNoErrorsCondition	No errors condition
xlTextString	Text string
xlTimePeriod	Time period
xlTop10	Top 10 values
xlUniqueValues	Unique values

Formatting by cell value:

```
With Range("A1").FormatConditions.Add(xlCellValue, xlGreater, "=100")
    With .Font
        .Bold = True
        .ColorIndex = 3
     End With
End With
```

Operators:

Name
xlBetween
xlEqual
xlGreater
xlGreaterEqual
xlLess
xlLessEqual
xlNotBetween
xlNotEqual

If Type is xlExpression, the Operator argument is ignored.

Formatting by text contains:

```
With Range("a1:a10").FormatConditions.Add(xlTextString, TextOperator:=xlContains, String:="egg")
    With .Font
        .Bold = True
        .ColorIndex = 3
    End With
End With
```

Operators:

Name	Description
xlBeginsWith	Begins with a specified value.
xlContains	Contains a specified value.
xlDoesNotContain	Does not contain the specified value.
xlEndsWith	Endswith the specified value

Formatting by time period

```
With Range("a1:a10").FormatConditions.Add(xlTimePeriod, DateOperator:=xlToday)
    With .Font
        .Bold = True
        .ColorIndex = 3
    End With
End With
```

Operators:

Name
xlYesterday
xlTomorrow
xlLast7Days
xlLastWeek
xlThisWeek
xlNextWeek
xlLastMonth
xlThisMonth
xlNextMonth

Section 9.2: Remove conditional format

Remove all conditional format in range:

```
Range("A1:A10").FormatConditions.Delete
```

Remove all conditional format in worksheet:

```
Cells.FormatConditions.Delete
```

Section 9.3: FormatConditions.AddUniqueValues

Highlighting Duplicate Values

```
With Range("E1:E100").FormatConditions.AddUniqueValues
   .DupeUnique = xlDuplicate
   With .Font
       .Bold = True
```

```
        .ColorIndex = 3
    End With
End With
```

Highlighting Unique Values

```
With Range("E1:E100").FormatConditions.AddUniqueValues
    With .Font
        .Bold = True
        .ColorIndex = 3
    End With
End With
```

Section 9.4: FormatConditions.AddTop10

Highlighting Top 5 Values

```
With Range("E1:E100").FormatConditions.AddTop10
    .TopBottom = xlTop10Top
    .Rank = 5
    .Percent = False
    With .Font
        .Bold = True
        .ColorIndex = 3
    End With
End With
```

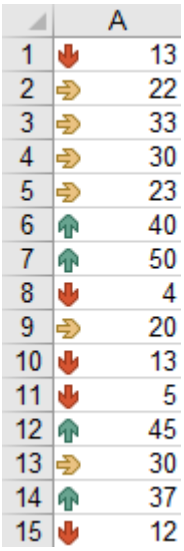
Section 9.5: FormatConditions.AddAboveAverage

```
With Range("E1:E100").FormatConditions.AddAboveAverage
    .AboveBelow = xlAboveAverage
    With .Font
        .Bold = True
        .ColorIndex = 3
    End With
End With
```

Operators:

Name	Description
XlAboveAverage	Above average
XlAboveStdDev	Above standard deviation
XlBelowAverage	Below average
XlBelowStdDev	Below standard deviation
XlEqualAboveAverage	Equal above average
XlEqualBelowAverage	Equal below average

Section 9.6: FormatConditions.AddIconSetCondition



	A
1	13
2	22
3	33
4	30
5	23
6	40
7	50
8	4
9	20
10	13
11	5
12	45
13	30
14	37
15	12

```
Range("a1:a10").FormatConditions.AddIconSetCondition
With Selection.FormatConditions(1)
    .ReverseOrder = False
    .ShowIconOnly = False
    .IconSet = ActiveWorkbook.IconSets(xl3Arrows)
End With

With Selection.FormatConditions(1).IconCriteria(2)
    .Type = xlConditionValuePercent
    .Value = 33
    .Operator = 7
End With

With Selection.FormatConditions(1).IconCriteria(3)
    .Type = xlConditionValuePercent
    .Value = 67
    .Operator = 7
End With
```

IconSet:

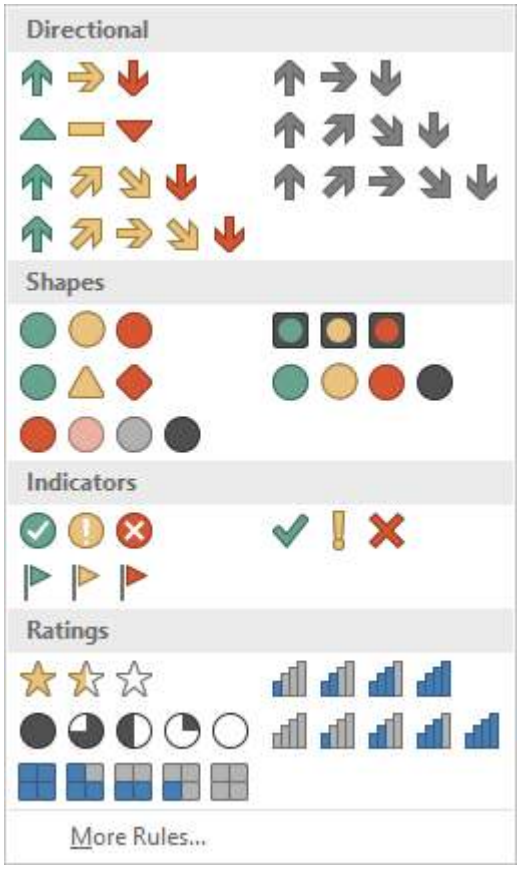
Name
xl3Arrows
xl3ArrowsGray
xl3Flags
xl3Signs
xl3Stars
xl3Symbols
xl3Symbols2
xl3TrafficLights1
xl3TrafficLights2
xl3Triangles
xl4Arrows
xl4ArrowsGray
xl4CRV
xl4RedToBlack
xl4TrafficLights
xl5Arrows

xl5ArrowsGray

xl5Boxes

xl5CRV

xl5Quarters



Type:

Name
xlConditionValuePercent
xlConditionValueNumber
xlConditionValuePercentile
xlConditionValueFormula

Operator:

Name	Value
xlGreater	5
xlGreaterEqual	7

Value:

Returns or sets the threshold value for an icon in a conditional format.

Chapter 10: Workbooks

Section 10.1: When To Use ActiveWorkbook and ThisWorkbook

It's a VBA Best Practice to always specify which workbook your VBA code refers. If this specification is omitted, then VBA assumes the code is directed at the currently active workbook (`ActiveWorkbook`).

```
'--- the currently active workbook (and worksheet) is implied
Range("A1").value = 3.1415
Cells(1, 1).value = 3.1415
```

However, when several workbooks are open at the same time -- particularly and especially when VBA code is running from an Excel Add-In -- references to the `ActiveWorkbook` may be confused or misdirected. For example, an add-in with a UDF that checks the time of day and compares it to a value stored on one of the add-in's worksheets (that are typically not readily visible to the user) will have to explicitly identify which workbook is being referenced. In our example, our open (and active) workbook has a formula in cell A1 `=EarlyOrLate()` and does NOT have any VBA written for that active workbook. In our add-in, we have the following User Defined Function (UDF):

```
Public Function EarlyOrLate() As String
    If Hour(Now) > ThisWorkbook.Sheets("WatchTime").Range("A1") Then
        EarlyOrLate = "It's Late!"
    Else
        EarlyOrLate = "It's Early!"
    End If
End Function
```

The code for the UDF is written and stored in the installed Excel add-in. It uses data stored on a worksheet in the add-in called "WatchTime". If the UDF had used `ActiveWorkbook` instead of `ThisWorkbook`, then it would never be able to guarantee which workbook was intended.

Section 10.2: Changing The Default Number of Worksheets In A New Workbook

The "factory default" number of worksheets created in a new Excel workbook is generally set to three. Your VBA code can explicitly set the number of worksheets in a new workbook.

```
'--- save the current Excel global setting
With Application
    Dim oldSheetsCount As Integer
    oldSheetsCount = .SheetsInNewWorkbook
    Dim myNewWB As Workbook
    .SheetsInNewWorkbook = 1
    Set myNewWB = .Workbooks.Add
    '--- restore the previous setting
    .SheetsInNewWorkbook = oldsheetcount
End With
```

Section 10.3: Application Workbooks

In many Excel applications, the VBA code takes actions directed at the workbook in which it's contained. You save that workbook with a ".xlsm" extension and the VBA macros only focus on the worksheets and data within. However, there are often times when you need to combine or merge data from other workbooks, or write some of your data to a separate workbook. Opening, closing, saving, creating, and deleting other workbooks is a common need for many VBA applications.

At any time in the VBA Editor, you can view and access any and all workbooks currently open by that instance of Excel by using the `Workbooks` property of the `Application` object. The MSDN Documentation explains it with references.

Section 10.4: Opening A (New) Workbook, Even If It's Already Open

If you want to access a workbook that's already open, then getting the assignment from the `Workbooks` collection is straightforward:

```
dim myWB as Workbook
Set myWB = Workbooks("UsuallyFullPathnameOfWorkbook.xlsx")
```

If you want to create a new workbook, then use the `Workbooks` collection object to `Add` a new entry.

```
Dim myNewWB as Workbook
Set myNewWB = Workbooks.Add
```

There are times when you may not or (or care) if the workbook you need is open already or not, or possible does not exist. The example function shows how to always return a valid workbook object.

```
Option Explicit
Function GetWorkbook(ByVal wbFilename As String) As Workbook
    '--- returns a workbook object for the given filename, including checks
    '    for when the workbook is already open, exists but not open, or
    '    does not yet exist (and must be created)
    '    ***  wbFilename must be a fully specified pathname
    Dim folderFile As String
    Dim returnedWB As Workbook

    '--- check if the file exists in the directory location
    folderFile = File(wbFilename)
    If folderFile = "" Then
        '--- the workbook doesn't exist, so create it
        Dim pos1 As Integer
        Dim fileExt As String
        Dim fileFormatNum As Long
        '--- in order to save the workbook correctly, we need to infer which workbook
        '    type the user intended from the file extension
        pos1 = InStrRev(sFullName, ".", , vbTextCompare)
        fileExt = Right(sFullName, Len(sFullName) - pos1)
        Select Case fileExt
            Case "xlsx"
                fileFormatNum = 51
            Case "xlsm"
                fileFormatNum = 52
            Case "xls"
                fileFormatNum = 56
            Case "xlsb"
                fileFormatNum = 50
            Case Else
                Err.Raise vbObjectError + 1000, "GetWorkbook function", _
                        "The file type you've requested (file extension) is not recognized. " & _
                        "Please use a known extension: xlsx, xlsm, xls, or xlsb."
        End Select
        Set returnedWB = Workbooks.Add
        Application.DisplayAlerts = False
        returnedWB.SaveAs filename:=wbFilename, FileFormat:=fileFormatNum
```

```
        Application.DisplayAlerts = True
        Set GetWorkbook = returnedWB
    Else
        '--- the workbook exists in the directory, so check to see if
        '    it's already open or not
        On Error Resume Next
        Set returnedWB = Workbooks(sFile)
        If returnedWB Is Nothing Then
            Set returnedWB = Workbooks.Open(sFullName)
        End If
    End If
End Function
```

Section 10.5: Saving A Workbook Without Asking The User

Often saving new data in an existing workbook using VBA will cause a pop-up question noting that the file already exists.

To prevent this pop-up question, you have to suppress these types of alerts.

```
Application.DisplayAlerts = False        'disable user prompt to overwrite file
myWB.SaveAs FileName:="NewOrExistingFilename.xlsx"
Application.DisplayAlerts = True         're-enable user prompt to overwrite file
```

Chapter 11: Working with Excel Tables in VBA

This topic is about working with tables in VBA, and assumes knowledge of Excel Tables. In VBA, or rather the Excel Object Model, tables are known as ListObjects. The most frequently used properties of a ListObject are ListRow(s), ListColumn(s), DataBodyRange, Range and HeaderRowRange.

Section 11.1: Instantiating a ListObject

```
Dim lo as ListObject
Dim MyRange as Range

Set lo = Sheet1.ListObjects(1)

'or

Set lo = Sheet1.ListObjects("Table1")

'or

Set lo = MyRange.ListObject
```

Section 11.2: Working with ListRows / ListColumns

```
Dim lo as ListObject
Dim lr as ListRow
Dim lc as ListColumn

Set lr = lo.ListRows.Add
Set lr = lo.ListRows(5)

For Each lr in lo.ListRows
    lr.Range.ClearContents
    lr.Range(1, lo.ListColumns("Some Column").Index).Value = 8
Next

Set lc = lo.ListColumns.Add
Set lc = lo.ListColumns(4)
Set lc = lo.ListColumns("Header 3")

For Each lc in lo.ListColumns
    lc.DataBodyRange.ClearContents   'DataBodyRange excludes the header row
    lc.Range(1,1).Value = "New Header Name"    'Range includes the header row
Next
```

Section 11.3: Converting an Excel Table to a normal range

```
Dim lo as ListObject

Set lo = Sheet1.ListObjects("Table1")
lo.Unlist
```

Chapter 12: Loop through all Sheets in Active Workbook

Section 12.1: Retrieve all Worksheets Names in Active Workbook

```
Option Explicit

Sub LoopAllSheets()

Dim sht As Excel.Worksheet
' declare an array of type String without committing to maximum number of members
Dim sht_Name() As String
Dim i As Integer

' get the number of worksheets in Active  Workbook , and put it as the maximum number of members in
the array
ReDim sht_Name(1 To ActiveWorkbook.Worksheets.count)

i = 1

' loop through all worksheets in Active Workbook
For Each sht In ActiveWorkbook.Worksheets
    sht_Name(i) = sht.Name ' get the name of each worksheet and save it in the array
    i = i + 1
Next sht

End Sub
```

Section 12.2: Loop Through all Sheets in all Files in a Folder

```
 Sub Theloopofloops()

 Dim wbk As Workbook
 Dim Filename As String
 Dim path As String
 Dim rCell As Range
 Dim rRng As Range
 Dim wsO As Worksheet
 Dim sheet As Worksheet


 path = "pathtofile(s)" & "\"
 Filename = Dir(path & "*.xl??")
 Set wsO = ThisWorkbook.Sheets("Sheet1") 'included in case you need to differentiate_
             between workbooks i.e currently opened workbook vs workbook containing code

 Do While Len(Filename) > 0
     DoEvents
     Set wbk = Workbooks.Open(path & Filename, True, True)
         For Each sheet In ActiveWorkbook.Worksheets  'this needs to be adjusted for specifiying
sheets. Repeat loop for each sheet so thats on a per sheet basis
                Set rRng = sheet.Range("a1:a1000") 'OBV needs to be changed
                For Each rCell In rRng.Cells
                If rCell <> "" And rCell.Value <> vbNullString And rCell.Value <> 0 Then

                   'code that does stuff
```

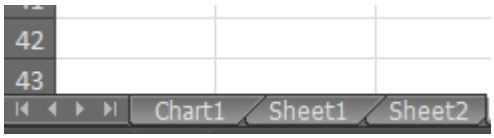
```
                End If
                Next rCell
        Next sheet
    wbk.Close False
    Filename = Dir
Loop
End Sub
```

Chapter 13: Use Worksheet object and not Sheet object

Plenty of VBA users consider Worksheets and Sheets objects synonyms. They are not.

Sheets object consists of both Worksheets and Charts. Thus, if we have charts in our Excel Workbook, we should be careful, not to use `Sheets` and `Worksheets` as synonyms.

Section 13.1: Print the name of the first object




```
Option Explicit

Sub CheckWorksheetsDiagram()

    Debug.Print Worksheets(1).Name
    Debug.Print Charts(1).Name
    Debug.Print Sheets(1).Name

End Sub
```

The result:

```
Sheet1
Chart1
Chart1
```

Chapter 14: Methods for Finding the Last Used Row or Column in a Worksheet

Section 14.1: Find the Last Non-Empty Cell in a Column

In this example, we will look at a method for returning the last non-empty row in a column for a data set.

This method will work regardless of empty regions within the data set.

However ***caution*** *should be used if* ***merged cells*** *are involved*, as the `End` method will be "stopped" against a merged region, returning the first cell of the merged region.

In addition non-empty cells in ***hidden rows*** will not be taken into account.

```
Sub FindingLastRow()
    Dim wS As Worksheet, LastRow As Long
    Set wS = ThisWorkbook.Worksheets("Sheet1")

    'Here we look in Column A
    LastRow = wS.Cells(wS.Rows.Count, "A").End(xlUp).Row
    Debug.Print LastRow
End Sub
```

To address the limitations indicated above, the line:
`LastRow = wS.Cells(wS.Rows.Count, "A").End(xlUp).Row`

may be replaced with:

1. for last used row of `"Sheet1"`:
   `LastRow = wS.UsedRange.Row - 1 + wS.UsedRange.Rows.Count`.
2. for last non-empty cell of Column `"A"` in `"Sheet1"`:

```
Dim i As Long
For i = LastRow To 1 Step -1
    If Not (IsEmpty(Cells(i, 1))) Then Exit For
Next i
LastRow = i
```

Section 14.2: Find the Last Non-Empty Row in Worksheet

```
Private Sub Get_Last_Used_Row_Index()
    Dim wS As Worksheet

    Set wS = ThisWorkbook.Sheets("Sheet1")
    Debug.Print LastRow_1(wS)
    Debug.Print LastRow_0(wS)
End Sub
```

You can choose between 2 options, regarding if you want to know if there is no data in the worksheet :

- NO : Use LastRow_1 : You can use it directly within `wS.Cells(LastRow_1(wS),...)`
- YES : Use LastRow_0 : You need to test if the result you get from the function is 0 or not before using it

```
Public Function LastRow_1(wS As Worksheet) As Double
    With wS
        If Application.WorksheetFunction.CountA(.Cells) <> 0 Then
            LastRow_1 = .Cells.Find(What:="*", _
                                    After:=.Range("A1"), _
                                    Lookat:=xlPart, _
                                    LookIn:=xlFormulas, _
                                    SearchOrder:=xlByRows, _
                                    SearchDirection:=xlPrevious, _
                                    MatchCase:=False).Row
        Else
            LastRow_1 = 1
        End If
    End With
End Function

Public Function LastRow_0(wS As Worksheet) As Double
    On Error Resume Next
    LastRow_0 = wS.Cells.Find(What:="*", _
                              After:=ws.Range("A1"), _
                              Lookat:=xlPart, _
                              LookIn:=xlFormulas, _
                              SearchOrder:=xlByRows, _
                              SearchDirection:=xlPrevious, _
                              MatchCase:=False).Row
End Function
```

Section 14.3: Find the Last Non-Empty Column in Worksheet

```
Private Sub Get_Last_Used_Row_Index()
    Dim wS As Worksheet

    Set wS = ThisWorkbook.Sheets("Sheet1")
    Debug.Print LastCol_1(wS)
    Debug.Print LastCol_0(wS)
End Sub
```

You can choose between 2 options, regarding if you want to know if there is no data in the worksheet :

- NO : Use LastCol_1 : You can use it directly within `wS.Cells(...,LastCol_1(wS))`
- YES : Use LastCol_0 : You need to test if the result you get from the function is 0 or not before using it

```
Public Function LastCol_1(wS As Worksheet) As Double
    With wS
        If Application.WorksheetFunction.CountA(.Cells) <> 0 Then
            LastCol_1 = .Cells.Find(What:="*", _
                                    After:=.Range("A1"), _
                                    Lookat:=xlPart, _
                                    LookIn:=xlFormulas, _
                                    SearchOrder:=xlByColumns, _
                                    SearchDirection:=xlPrevious, _
                                    MatchCase:=False).Column
        Else
            LastCol_1 = 1
        End If
    End With
End Function
```

The Err object's properties are automatically reset to zero upon function exit.

```
Public Function LastCol_0(wS As Worksheet) As Double
    On Error Resume Next
    LastCol_0 = wS.Cells.Find(What:="*", _
                              After:=ws.Range("A1"), _
                              Lookat:=xlPart, _
                              LookIn:=xlFormulas, _
                              SearchOrder:=xlByColumns, _
                              SearchDirection:=xlPrevious, _
                              MatchCase:=False).Column
End Function
```

Section 14.4: Find the Last Non-Empty Cell in a Row

In this example, we will look at a method for returning the last non-empty column in a row.

This method will work regardless of empty regions within the data set.

However ***caution*** *should be used if* ***merged cells*** *are involved*, as the `End` method will be "stopped" against a merged region, returning the first cell of the merged region.

In addition non-empty cells in ***hidden columns*** will not be taken into account.

```
Sub FindingLastCol()
    Dim wS As Worksheet, LastCol As Long
    Set wS = ThisWorkbook.Worksheets("Sheet1")

    'Here we look in Row 1
    LastCol = wS.Cells(1, wS.Columns.Count).End(xlToLeft).Column
    Debug.Print LastCol
End Sub
```

Section 14.5: Get the row of the last cell in a range

```
'if only one area (not multiple areas):
With Range("A3:D20")
    Debug.Print .Cells(.Cells.CountLarge).Row
    Debug.Print .Item(.Cells.CountLarge).Row 'using .item is also possible
End With 'Debug prints: 20

'with multiple areas (also works if only one area):
Dim rngArea As Range, LastRow As Long
With Range("A3:D20, E5:I50, H20:R35")
    For Each rngArea In .Areas
        If rngArea(rngArea.Cells.CountLarge).Row > LastRow Then
            LastRow = rngArea(rngArea.Cells.CountLarge).Row
        End If
    Next
    Debug.Print LastRow 'Debug prints: 50
End With
```

Section 14.6: Find Last Row Using Named Range

In case you have a Named Range in your Sheet, and you want to dynamically get the last row of that Dynamic Named Range. Also covers cases where the Named Range doesn't start from the first Row.

```
Sub FindingLastRow()
```

```
Dim sht As Worksheet
Dim LastRow As Long
Dim FirstRow As Long

Set sht = ThisWorkbook.Worksheets("form")

'Using Named Range "MyNameRange"
FirstRow = sht.Range("MyNameRange").Row

' in case "MyNameRange" doesn't start at Row 1
LastRow = sht.Range("MyNameRange").Rows.count + FirstRow - 1

End Sub
```

Update:
A potential loophole was pointed out by @Jeeped for a a named range with non-contiguous rows as it generates unexpected result. To addresses that issue, the code is revised as below.
Asumptions: targes sheet = `form`, named range = `MyNameRange`

```
Sub FindingLastRow()
    Dim rw As Range, rwMax As Long
    For Each rw In Sheets("form").Range("MyNameRange").Rows
        If rw.Row > rwMax Then rwMax = rw.Row
    Next
    MsgBox "Last row of 'MyNameRange' under Sheets 'form': " & rwMax
End Sub
```

Section 14.7: Last cell in Range.CurrentRegion

`Range.CurrentRegion` is a rectangular range area surrounded by empty cells. Blank cells with formulas such as `=""` or `'` are not considered blank (even by the `ISBLANK` Excel function).

```
Dim rng As Range, lastCell As Range
Set rng = Range("C3").CurrentRegion       ' or Set rng = Sheet1.UsedRange.CurrentRegion
Set lastCell = rng(rng.Rows.Count, rng.Columns.Count)
```

Section 14.8: Find the Last Non-Empty Cell in Worksheet - Performance (Array)

- The first function, using an array, is **much faster**
- If called without the optional parameter, will default to `.ThisWorkbook.ActiveSheet`
- If the range is empty will returns `Cell( 1, 1 )` as default, instead of `Nothing`

Speed:

> ```
> GetMaxCell (Array): Duration: 0.0000790063 seconds
> GetMaxCell (Find ): Duration: 0.0002903480 seconds
> ```
>
> .Measured with MicroTimer

```
Public Function GetLastCell(Optional ByVal ws As Worksheet = Nothing) As Range
    Dim uRng As Range, uArr As Variant, r As Long, c As Long
    Dim ubR As Long, ubC As Long, lRow As Long
```

```
    If ws Is Nothing Then Set ws = Application.ThisWorkbook.ActiveSheet
    Set uRng = ws.UsedRange
    uArr = uRng
    If IsEmpty(uArr) Then
        Set GetLastCell = ws.Cells(1, 1):   Exit Function
    End If
    If Not IsArray(uArr) Then
        Set GetLastCell = ws.Cells(uRng.Row, uRng.Column):  Exit Function
    End If
    ubR = UBound(uArr, 1):  ubC = UBound(uArr, 2)
    For r = ubR To 1 Step -1    '------------------------------------------- last row
        For c = ubC To 1 Step -1
            If Not IsError(uArr(r, c)) Then
                If Len(Trim$(uArr(r, c))) > 0 Then
                    lRow = r:   Exit For
                End If
            End If
        Next
        If lRow > 0 Then Exit For
    Next
    If lRow = 0 Then lRow = ubR
    For c = ubC To 1 Step -1    '------------------------------------------- last col
        For r = lRow To 1 Step -1
            If Not IsError(uArr(r, c)) Then
                If Len(Trim$(uArr(r, c))) > 0 Then
                    Set GetLastCell = ws.Cells(lRow + uRng.Row - 1, c + uRng.Column - 1)
                    Exit Function
                End If
            End If
        Next
    Next
End Function
```

```
'Returns last cell (max row & max col) using Find

Public Function GetMaxCell2(Optional ByRef rng As Range = Nothing) As Range 'Using Find

    Const NONEMPTY As String = "*"

    Dim lRow As Range, lCol As Range

    If rng Is Nothing Then Set rng = Application.ThisWorkbook.ActiveSheet.UsedRange

    If WorksheetFunction.CountA(rng) = 0 Then
        Set GetMaxCell2 = rng.Parent.Cells(1, 1)
    Else
        With rng
            Set lRow = .Cells.Find(What:=NONEMPTY, LookIn:=xlFormulas, _
                                   After:=.Cells(1, 1), _
                                   SearchDirection:=xlPrevious, _
                                   SearchOrder:=xlByRows)
            If Not lRow Is Nothing Then
                Set lCol = .Cells.Find(What:=NONEMPTY, LookIn:=xlFormulas, _
                                       After:=.Cells(1, 1), _
                                       SearchDirection:=xlPrevious, _
                                       SearchOrder:=xlByColumns)

                Set GetMaxCell2 = .Parent.Cells(lRow.Row, lCol.Column)
            End If
        End With
    End If
```

```
End Function
```

.

MicroTimer:

```
Private Declare PtrSafe Function getFrequency Lib "Kernel32" Alias "QueryPerformanceFrequency"
(cyFrequency As Currency) As Long
Private Declare PtrSafe Function getTickCount Lib "Kernel32" Alias "QueryPerformanceCounter"
(cyTickCount As Currency) As Long

Function MicroTimer() As Double
    Dim cyTicks1 As Currency
    Static cyFrequency As Currency

    MicroTimer = 0
    If cyFrequency = 0 Then getFrequency cyFrequency        'Get frequency
    getTickCount cyTicks1                                   'Get ticks
    If cyFrequency Then MicroTimer = cyTicks1 / cyFrequency 'Returns Seconds
End Function
```

Chapter 15: Creating a drop-down menu in the Active Worksheet with a Combo Box

This is a simple example demonstrating how to create a drop down menu in the Active Sheet of your workbook by inserting a Combo Box Activex object in the sheet. You'll be able to insert one of five Jimi Hendrix songs in any activated cell of the sheet and be able to clear it, accordingly.

Section 15.1: Example 2: Options Not Included

This example is used in specifying options that might not be included in a database of available housing and its attendant amenities.

It builds on the previous example, with some differences:

1. Two procedures are no longer necessary for a single combo box, done by combining the code into a single procedure.
2. The use of the LinkedCell property to allow for the correct input of the user selection every time
3. The inclusion of a backup feature for ensuring the active cell is in the correct column and an error prevention code, based on previous experience, where numeric values would formatted as strings when populated to the active cell.

```
Private Sub cboNotIncl_Change()

Dim n As Long
Dim notincl_array(1 To 9) As String

n = myTarget.Row

    If n >= 3 And n < 10000 Then

        If myTarget.Address = "$G$" & n Then

            'set up the array elements for the not included services
            notincl_array(1) = "Central Air"
            notincl_array(2) = "Hot Water"
            notincl_array(3) = "Heater Rental"
            notincl_array(4) = "Utilities"
            notincl_array(5) = "Parking"
            notincl_array(6) = "Internet"
            notincl_array(7) = "Hydro"
            notincl_array(8) = "Hydro/Hot Water/Heater Rental"
            notincl_array(9) = "Hydro and Utilities"

            cboNotIncl.List = notincl_array()

        Else

            Exit Sub

        End If

        With cboNotIncl

            'make sure the combo box moves to the target cell
            .Left = myTarget.Left
            .Top = myTarget.Top
```

```
            'adjust the size of the cell to fit the combo box
            myTarget.ColumnWidth = .Width * 0.18

            'make it look nice by editing some of the font attributes
            .Font.Size = 11
            .Font.Bold = False

            'populate the cell with the user choice, with a backup guarantee that it's in column G

            If myTarget.Address = "$G$" & n Then

                    .LinkedCell = myTarget.Address

                    'prevent an error where a numerical value is formatted as text
                    myTarget.EntireColumn.TextToColumns

            End If

        End With

    End If 'ensure that the active cell is only between rows 3 and 1000

End Sub
```

The above macro is initiated every time a cell is activated with the SelectionChange event in the worksheet module:

```
Public myTarget As Range

Private Sub Worksheet_SelectionChange(ByVal Target As Range)

    Set myTarget = Target

    'switch for Not Included
    If Target.Column = 7 And Target.Cells.Count = 1 Then

        Application.Run "Module1.cboNotIncl_Change"

    End If

End Sub
```

Section 15.2: Jimi Hendrix Menu

In general, the code is placed in the module of a sheet.

This is the Worksheet_SelectionChange event, which fires each time a different cell is selected in the active sheet. You can select "Worksheet" from the first drop-down menu above the code window, and "Selection_Change" from the drop down menu next to it. In this case, every time you activate a cell, the code is redirected to the Combo Box's code.

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

   ComboBox1_Change

End Sub
```

Here, the routine dedicated to the ComboBox is coded to the Change event by default. In it, there is a fixed array, populated with all the options. Not the CLEAR option in the last position, which will be used to clear the contents of a cell. The array then is handed to to the Combo Box and passed to the routine that does the work.

```
Private Sub ComboBox1_Change()

Dim myarray(0 To 5)
    myarray(0) = "Hey Joe"
    myarray(1) = "Little Wing"
    myarray(2) = "Voodoo Child"
    myarray(3) = "Purple Haze"
    myarray(4) = "The Wind Cries Mary"
    myarray(5) = "CLEAR"

    With ComboBox1
        .List = myarray()
    End With

    FillACell myarray()

End Sub
```

The array is passed to the routine that fills the cells with the song name or null value to empty them. First, an integer variable is given the value of the position of the choice that the user makes. Then, the Combo Box is moved to the TOP LEFT corner of the cell the user activates and its dimensions adjusted to make the experience more fluid. The active cell is then assigned the value in the position in the integer variable, which tracks the user choice. In case the user selects CLEAR from the options, the cell is emptied.

The entire routine repeats for each selected cell.

```
Sub FillACell(MyArray As Variant)

Dim n As Integer

n = ComboBox1.ListIndex

ComboBox1.Left = ActiveCell.Left
ComboBox1.Top = ActiveCell.Top
Columns(ActiveCell.Column).ColumnWidth = ComboBox1.Width * 0.18

ActiveCell = MyArray(n)

If ComboBox1 = "CLEAR" Then
    Range(ActiveCell.Address) = ""
End If

End Sub
```

Chapter 16: File System Object

Section 16.1: File, folder, drive exists

File exists:

```
Sub FileExists()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    If fso.FileExists("D:\test.txt") = True Then
       MsgBox "The file is exists."
    Else
       MsgBox "The file isn't exists."
    End If
End Sub
```

Folder exists:

```
Sub FolderExists()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    If fso.FolderExists("D:\testFolder") = True Then
       MsgBox "The folder is exists."
    Else
       MsgBox "The folder isn't exists."
    End If
End Sub
```

Drive exists:

```
Sub DriveExists()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    If fso.DriveExists("D:\") = True Then
       MsgBox "The drive is exists."
    Else
       MsgBox "The drive isn't exists."
    End If
End Sub
```

Section 16.2: Basic file operations

Copy:

```
Sub CopyFile()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    fso.CopyFile "c:\Documents and Settings\Makro.txt", "c:\Documents and Settings\Macros\"
End Sub
```

Move:

```
Sub MoveFile()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    fso.MoveFile "c:\*.txt", "c:\Documents and Settings\"
End Sub
```

Delete:

```
Sub DeleteFile()
    Dim fso
    Set fso = CreateObject("Scripting.FileSystemObject")
    fso.DeleteFile "c:\Documents and Settings\Macros\Makro.txt"
```

```
End Sub
```

Section 16.3: Basic folder operations

Create:

```
Sub CreateFolder()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    fso.CreateFolder "c:\Documents and Settings\NewFolder"
End Sub
```

Copy:

```
Sub CopyFolder()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    fso.CopyFolder "C:\Documents and Settings\NewFolder", "C:\"
End Sub
```

Move:

```
Sub MoveFolder()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    fso.MoveFolder "C:\Documents and Settings\NewFolder", "C:\"
End Sub
```

Delete:

```
Sub DeleteFolder()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    fso.DeleteFolder "C:\Documents and Settings\NewFolder"
End Sub
```

Section 16.4: Other operations

Get file name:

```
Sub GetFileName()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    MsgBox fso.GetFileName("c:\Documents and Settings\Makro.txt")
End Sub
```

Result: *Makro.txt*

Get base name:

```
Sub GetBaseName()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    MsgBox fso.GetBaseName("c:\Documents and Settings\Makro.txt")
End Sub
```

Result: *Makro*

Get extension name:

```
Sub GetExtensionName()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    MsgBox fso.GetExtensionName("c:\Documents and Settings\Makro.txt")
```

```
End Sub
```

Result: *txt*

Get drive name:

```
Sub GetDriveName()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    MsgBox fso.GetDriveName("c:\Documents and Settings\Makro.txt")
End Sub
```

Result: *c:*

Chapter 17: Pivot Tables

Section 17.1: Adding Fields to a Pivot Table

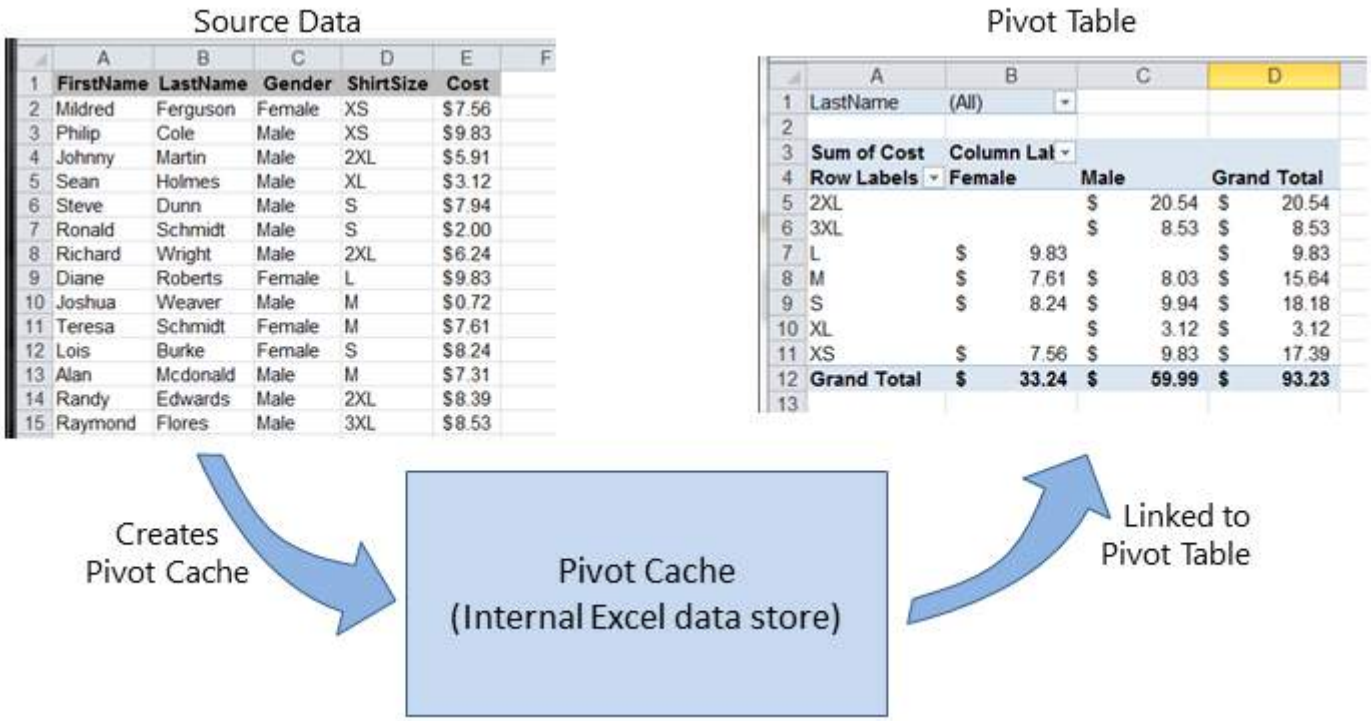
Two important things to note when adding fields to a Pivot Table are Orientation and Position. Sometimes a developer may assume where a field is placed, so it's always clearer to explicitly define these parameters. These actions only affect the given Pivot Table, not the Pivot Cache.

```
Dim thisPivot As PivotTable
Dim ptSheet As Worksheet
Dim ptField As PivotField

Set ptSheet = ThisWorkbook.Sheets("SheetNameWithPivotTable")
Set thisPivot = ptSheet.PivotTables(1)

With thisPivot
    Set ptField = .PivotFields("Gender")
    ptField.Orientation = xlRowField
    ptField.Position = 1
    Set ptField = .PivotFields("LastName")
    ptField.Orientation = xlRowField
    ptField.Position = 2
    Set ptField = .PivotFields("ShirtSize")
    ptField.Orientation = xlColumnField
    ptField.Position = 1
    Set ptField = .AddDataField(.PivotFields("Cost"), "Sum of Cost", xlSum)
    .InGridDropZones = True
    .RowAxisLayout xlTabularRow
End With
```

Section 17.2: Creating a Pivot Table

One of the most powerful capabilities in Excel is the use of Pivot Tables to sort and analyze data. Using VBA to create and manipulate the Pivots is easier if you understand the relationship of Pivot Tables to Pivot Caches and how to reference and use the different parts of the Tables.

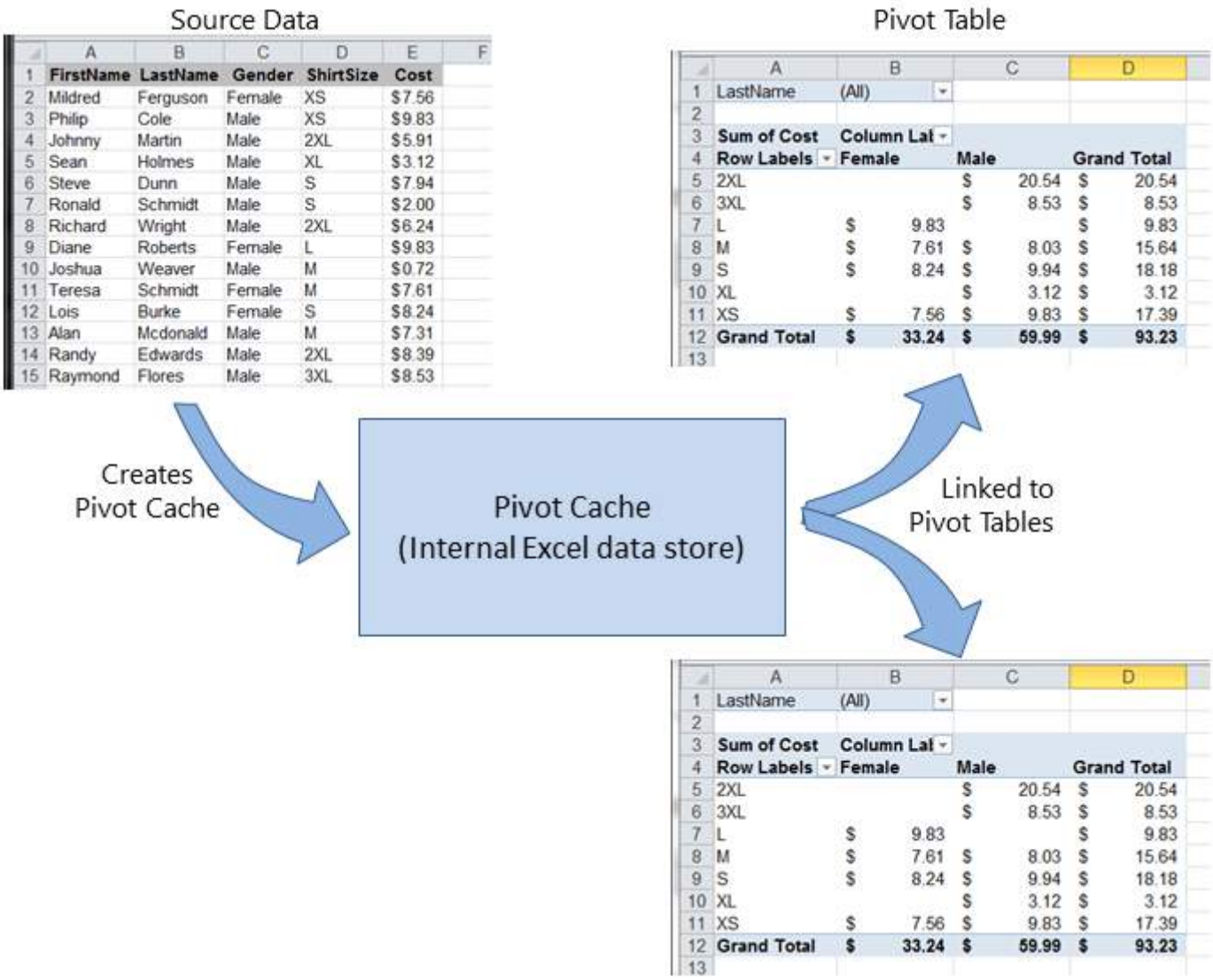
At its most basic, your source data is a `Range` area of data on a `Worksheet`. This data area **MUST** identify the data columns with a header row as the first row in the range. Once the Pivot Table is created, the user may view and change the source data at any time. However, changes may not be automatically or immediately reflected in the Pivot Table itself because there is an intermediate data storage structure called the Pivot Cache that is directly connected to the Pivot Table itself.



	A	B	C	D	E	F
1	**FirstName**	**LastName**	**Gender**	**ShirtSize**	**Cost**	
2	Mildred	Ferguson	Female	XS	$7.56	
3	Philip	Cole	Male	XS	$9.83	
4	Johnny	Martin	Male	2XL	$5.91	
5	Sean	Holmes	Male	XL	$3.12	
6	Steve	Dunn	Male	S	$7.94	
7	Ronald	Schmidt	Male	S	$2.00	
8	Richard	Wright	Male	2XL	$6.24	
9	Diane	Roberts	Female	L	$9.83	
10	Joshua	Weaver	Male	M	$0.72	
11	Teresa	Schmidt	Female	M	$7.61	
12	Lois	Burke	Female	S	$8.24	
13	Alan	Mcdonald	Male	M	$7.31	
14	Randy	Edwards	Male	2XL	$8.39	
15	Raymond	Flores	Male	3XL	$8.53	

	A	B	C	D
1	LastName	(All)		
2				
3	**Sum of Cost**	**Column Lal**		
4	**Row Labels**	**Female**	**Male**	**Grand Total**
5	2XL		$ 20.54	$ 20.54
6	3XL		$ 8.53	$ 8.53
7	L	$ 9.83		$ 9.83
8	M	$ 7.61	$ 8.03	$ 15.64
9	S	$ 8.24	$ 9.94	$ 18.18
10	XL		$ 3.12	$ 3.12
11	XS	$ 7.56	$ 9.83	$ 17.39
12	**Grand Total**	**$ 33.24**	**$ 59.99**	**$ 93.23**
13				

If multiple Pivot Tables are needed, based on the same source data, the Pivot Cache may be re-used as the internal data store for each of the Pivot Tables. This is a good practice because it saves memory and reduces the size of the Excel file for storage.



Source Data

	A	B	C	D	E	F
1	FirstName	LastName	Gender	ShirtSize	Cost	
2	Mildred	Ferguson	Female	XS	$7.56	
3	Philip	Cole	Male	XS	$9.83	
4	Johnny	Martin	Male	2XL	$5.91	
5	Sean	Holmes	Male	XL	$3.12	
6	Steve	Dunn	Male	S	$7.94	
7	Ronald	Schmidt	Male	S	$2.00	
8	Richard	Wright	Male	2XL	$6.24	
9	Diane	Roberts	Female	L	$9.83	
10	Joshua	Weaver	Male	M	$0.72	
11	Teresa	Schmidt	Female	M	$7.61	
12	Lois	Burke	Female	S	$8.24	
13	Alan	Mcdonald	Male	M	$7.31	
14	Randy	Edwards	Male	2XL	$8.39	
15	Raymond	Flores	Male	3XL	$8.53	

Pivot Table

	A	B	C	D
1	LastName	(All)		
2				
3	Sum of Cost	Column Lal		
4	Row Labels	Female	Male	Grand Total
5	2XL		$ 20.54	$ 20.54
6	3XL		$ 8.53	$ 8.53
7	L	$ 9.83		$ 9.83
8	M	$ 7.61	$ 8.03	$ 15.64
9	S	$ 8.24	$ 9.94	$ 18.18
10	XL		$ 3.12	$ 3.12
11	XS	$ 7.56	$ 9.83	$ 17.39
12	Grand Total	$ 33.24	$ 59.99	$ 93.23
13				

	A	B	C	D
1	LastName	(All)		
2				
3	Sum of Cost	Column Lal		
4	Row Labels	Female	Male	Grand Total
5	2XL		$ 20.54	$ 20.54
6	3XL		$ 8.53	$ 8.53
7	L	$ 9.83		$ 9.83
8	M	$ 7.61	$ 8.03	$ 15.64
9	S	$ 8.24	$ 9.94	$ 18.18
10	XL		$ 3.12	$ 3.12
11	XS	$ 7.56	$ 9.83	$ 17.39
12	Grand Total	$ 33.24	$ 59.99	$ 93.23
13				

As an example, to create a Pivot Table based on the source data shown in the Figures above:

```
Sub test()
    Dim pt As PivotTable
    Set pt = CreatePivotTable(ThisWorkbook.Sheets("Sheet1").Range("A1:E15"))
End Sub

Function CreatePivotTable(ByRef srcData As Range) As PivotTable
    '--- creates a Pivot Table from the given source data and
    '    assumes that the first row contains valid header data
    '    for the columns
    Dim thisPivot As PivotTable
    Dim dataSheet As Worksheet
    Dim ptSheet As Worksheet
    Dim ptCache As PivotCache

    '--- the Pivot Cache must be created first...
    Set ptCache = ThisWorkbook.PivotCaches.Create(SourceType:=xlDatabase, _
                                                  SourceData:=srcData)
    '--- ... then use the Pivot Cache to create the Table
    Set ptSheet = ThisWorkbook.Sheets.Add
    Set thisPivot = ptCache.CreatePivotTable(TableDestination:=ptSheet.Range("A3"))
    Set CreatePivotTable = thisPivot
End Function
```

> **References** MSDN Pivot Table Object

Section 17.3: Pivot Table Ranges

These excellent reference sources provide descriptions and illustrations of the various ranges in Pivot Tables.

> **References**
>
> - Referencing Pivot Table Ranges in VBA - from Jon Peltier's Tech Blog
> - Referencing an Excel Pivot Table Range using VBA - from globaliconnect Excel VBA

Section 17.4: Formatting the Pivot Table Data

This example changes/sets several formats in the data range area (`DataBodyRange`) of the given Pivot Table. All formattable parameters in a standard `Range` are available. Formatting the data only affects the Pivot Table itself, not the Pivot Cache.

NOTE: the property is named `TableStyle2` because the `TableStyle` property is not a member of the `PivotTable`'s object properties.

```
Dim thisPivot As PivotTable
Dim ptSheet As Worksheet
Dim ptField As PivotField

Set ptSheet = ThisWorkbook.Sheets("SheetNameWithPivotTable")
Set thisPivot = ptSheet.PivotTables(1)

With thisPivot
    .DataBodyRange.NumberFormat = "_($* #,##0.00_);_($* (#,##0.00);_($* "-"??_);_(@_)"
    .DataBodyRange.HorizontalAlignment = xlRight
    .ColumnRange.HorizontalAlignment = xlCenter
    .TableStyle2 = "PivotStyleMedium9"
End With
```

Chapter 18: Binding

Section 18.1: Early Binding vs Late Binding

Binding is the process of assigning an object to an identifier or variable name. Early binding (also known as static binding) is when an object declared in Excel is of a specific object type, such as a Worksheet or Workbook. Late binding occurs when general object associations are made, such as the Object and Variant declaration types.

Early binding of references some advantages over late binding.

- Early binding is operationally faster than late binding during run-time. Creating the object with late binding in run-time takes time that early binding accomplishes when the VBA project is initially loaded.
- Early binding offers additional functionality through the identification of Key/Item pairs by their ordinal position.
- Depending on code structure, early binding may offer an additional level of type checking and reduce errors.
- The VBE's capitalization correction when typing a bound object's properties and methods is active with early binding but unavailable with late binding.

> **Note:** You must add the appropriate reference to the VBA project through the VBE's Tools → References command in order to implement early binding.
> This library reference is then carried with the project; it does not have to be re-referenced when the VBA project is distributed and run on another computer.

```
'Looping through a dictionary that was created with late binding'
Sub iterateDictionaryLate()
    Dim k As Variant, dict As Object

    Set dict = CreateObject("Scripting.Dictionary")
    dict.comparemode = vbTextCompare          'non-case sensitive compare model

    'populate the dictionary
    dict.Add Key:="Red", Item:="Balloon"
    dict.Add Key:="Green", Item:="Balloon"
    dict.Add Key:="Blue", Item:="Balloon"

    'iterate through the keys
    For Each k In dict.Keys
        Debug.Print k & " - " & dict.Item(k)
    Next k

    dict.Remove "blue"      'remove individual key/item pair by key
    dict.RemoveAll          'remove all remaining key/item pairs

End Sub

'Looping through a dictionary that was created with early binding'
Sub iterateDictionaryEarly()
    Dim d As Long, k As Variant
    Dim dict As New Scripting.Dictionary

    dict.CompareMode = vbTextCompare          'non-case sensitive compare model

    'populate the dictionary
    dict.Add Key:="Red", Item:="Balloon"
    dict.Add Key:="Green", Item:="Balloon"
```

```
    dict.Add Key:="Blue", Item:="Balloon"
    dict.Add Key:="White", Item:="Balloon"

    'iterate through the keys
    For Each k In dict.Keys
        Debug.Print k & " - " & dict.Item(k)
    Next k

    'iterate through the keys by the count
    For d = 0 To dict.Count - 1
        Debug.Print dict.Keys(d) & " - " & dict.Items(d)
    Next d

    'iterate through the keys by the boundaries of the keys collection
    For d = LBound(dict.Keys) To UBound(dict.Keys)
        Debug.Print dict.Keys(d) & " - " & dict.Items(d)
    Next d

    dict.Remove "blue"                          'remove individual key/item pair by key
    dict.Remove dict.Keys(0)                    'remove first key/item by index position
    dict.Remove dict.Keys(UBound(dict.Keys))    'remove last key/item by index position
    dict.RemoveAll                              'remove all remaining key/item pairs

End Sub
```

However, if you are using early binding and the document is run on a system that lacks one of the libraries you have referenced, you will encounter problems. Not only will the routines that utilize the missing library not function properly, but the behavior of all code within the document will become erratic. It is likely that none of the document's code will function on that computer.

This is where late binding is advantageous. When using late binding you do not have to add the reference in the Tools>References menu. On machines that have the appropriate library, the code will still work. On machines without that library, the commands that reference the library will not work, but all the other code in your document will continue to function.

If you are not thoroughly familiar with the library you are referencing, it may be useful to use early binding while writing the code, then switch to late binding before deployment. That way you can take advantage of the VBE's IntelliSense and Object Browser during development.

Chapter 19: autofilter ; Uses and best practices

Autofilter ultimate goal is to provide in the quickest way possible data mining from hundreds or thousands of rows data in order to get the attention in the items we want to focus on. It can receive parameters such as "text/values/colors" and they can be stacked among columns. You may connect up to 2 criteria per column based in logical connectors and sets of rules. Remark: Autofilter works by filtering rows, there is no Autofilter to filter columns (at least not natively).

Section 19.1: Smartfilter!

Problem situation

Warehouse administrator has a sheet ("Record") where every logistics movement performed by the facility is stored, he may filter as needed, although, this is very time consuming and he would like to improve the process in order to calculate inquiries faster, for example: How many "pulp" do we have now (in all racks)? How many pulp do we have now (in rack #5)? Filters are a great tool but, they are somewhat limited to answer these kind of question in matter of seconds.

	A	B	C	D	E	F	G	H	I	J
1	Control Num	DESCRIPTIO	QUANTI	LOCATI	DATE	ACTIO		1. How many "Pulp" do we have now? (Total)		1. How many "Pulp" do we have now? (In Rack
2	9005124	Pulp	42	Rack #5	4-Oct-16	In				
15	9005137	Pulp	67	Rack #1	21-Nov-15	Out				
16	9005138	Pulp	92	Rack #3	19-Jun-15	Out				
42	9005164	Pulp	48	Rack #5	1-Dec-15	In				
45	9005167	Pulp	53	Rack #5	17-Mar-15	Out				
50	9005172	Pulp	13	Rack #3	5-Dec-15	In				
55	9005177	Pulp	30	Rack #2	15-Sep-16	In				
56	9005178	Pulp	90	Rack #3	27-Jan-16	Out				
68	9005190	Pulp	67	Rack #7	25-Aug-16	Out				
70	9005192	Pulp	62	Rack #6	7-Nov-15	Out				
71	9005193	Pulp	46	Rack #7	1-Dec-15	Out				
72	9005194	Pulp	6	Rack #2	18-Dec-16	Out				
83	9005205	Pulp	86	Rack #6	30-Mar-16	Out				
102	9005224	Pulp	78	Rack #3	7-Sep-16	Out				
109	9005231	Pulp	19	Rack #1	21-May-15	In				
115	9005237	Pulp	33	Rack #6	14-Jan-15	Out				
121	9005243	Pulp	46	Rack #1	25-Sep-15	Out				
124	9005246	Pulp	48	Rack #1	3-Jan-15	In				
125	9005247	Pulp	39	Rack #3	8-May-16	Out				
142	9005264	Pulp	68	Rack #1	15-Nov-15	In				
146	9005268	Pulp	50	Rack #2	30-Nov-16	In				
154	9005276	Pulp	11	Rack #4	8-Dec-15	In				
156	9005278	Pulp	40	Rack #1	5-Jun-16	In				
169	9005291	Pulp	84	Rack #4	21-Sep-16	Out				
174	9005296	Pulp	31	Rack #1	3-May-16	In				
182	9005304	Pulp	61	Rack #7	9-Apr-16	Out				
190	9005312	Pulp	57	Rack #1	2-Jul-15	Out				
192	9005314	Pulp	56	Rack #2	12-Feb-15	In				
200	9005322	Pulp	43	Rack #7	27-Sep-16	Out				
202	9005324	Pulp	97	Rack #1	16-Apr-16	In				
205	9005327	Pulp	80	Rack #6	8-Nov-16	In				
214	9005336	Pulp	82	Rack #5	27-Jul-15	In				
215	9005337	Pulp	27	Rack #4	17-Sep-16	In				
218	9005340	Pulp	51	Rack #3	16-Nov-15	Out				

Record

Macro solution:

The coder knows that ***autofilters are the best, fast and most reliable solution*** in these kind of scenarios since ***the data exists already in the worksheet*** and the ***input for them can be obtained easily*** -in this case, by user input-.
The approach used is to create a sheet called "SmartFilter" where administrator can easily filter multiple data as needed and calculation will be performed instantly as well.
He uses 2 modules and the `Worksheet_Change` event for this matter

Code For SmartFilter Worksheet:

```
Private Sub Worksheet_Change(ByVal Target As Range)
Dim ItemInRange As Range
Const CellsFilters As String = "C2,E2,G2"
    Call ExcelBusy
    For Each ItemInRange In Target
    If Not Intersect(ItemInRange, Range(CellsFilters)) Is Nothing Then Call Inventory_Filter
    Next ItemInRange
    Call ExcelNormal
End Sub
```

Code for module 1, called "General_Functions"

```
Sub ExcelNormal()
        With Excel.Application
        .EnableEvents = True
        .Cursor = xlDefault
        .ScreenUpdating = True
        .DisplayAlerts = True
        .StatusBar = False
        .CopyObjectsWithCells = True
        End With
End Sub
Sub ExcelBusy()
        With Excel.Application
        .EnableEvents = False
        .Cursor = xlWait
        .ScreenUpdating = False
        .DisplayAlerts = False
        .StatusBar = False
        .CopyObjectsWithCells = True
        End With
End Sub
Sub Select_Sheet(NameSheet As String, Optional VerifyExistanceOnly As Boolean)
    On Error GoTo Err01Select_Sheet
    Sheets(NameSheet).Visible = True
    If VerifyExistanceOnly = False Then ' 1. If VerifyExistanceOnly = False
    Sheets(NameSheet).Select
    Sheets(NameSheet).AutoFilterMode = False
    Sheets(NameSheet).Cells.EntireRow.Hidden = False
    Sheets(NameSheet).Cells.EntireColumn.Hidden = False
    End If ' 1. If VerifyExistanceOnly = False
    If 1 = 2 Then '99. If error
Err01Select_Sheet:
    MsgBox "Err01Select_Sheet: Sheet " & NameSheet & " doesn't exist!", vbCritical: Call
ExcelNormal: On Error GoTo -1: End
    End If '99. If error
End Sub
Function General_Functions_Find_Title(InSheet As String, TitleToFind As String, Optional InRange As
Range, Optional IsNeededToExist As Boolean, Optional IsWhole As Boolean) As Range
Dim DummyRange As Range
    On Error GoTo Err01General_Functions_Find_Title
    If InRange Is Nothing Then ' 1. If InRange Is Nothing
    Set DummyRange = IIf(IsWhole = True, Sheets(InSheet).Cells.Find(TitleToFind, LookAt:=xlWhole),
Sheets(InSheet).Cells.Find(TitleToFind, LookAt:=xlPart))
    Else ' 1. If InRange Is Nothing
    Set DummyRange = IIf(IsWhole = True, Sheets(InSheet).Range(InRange.Address).Find(TitleToFind,
LookAt:=xlWhole), Sheets(InSheet).Range(InRange.Address).Find(TitleToFind, LookAt:=xlPart))
    End If ' 1. If InRange Is Nothing
    Set General_Functions_Find_Title = DummyRange
```

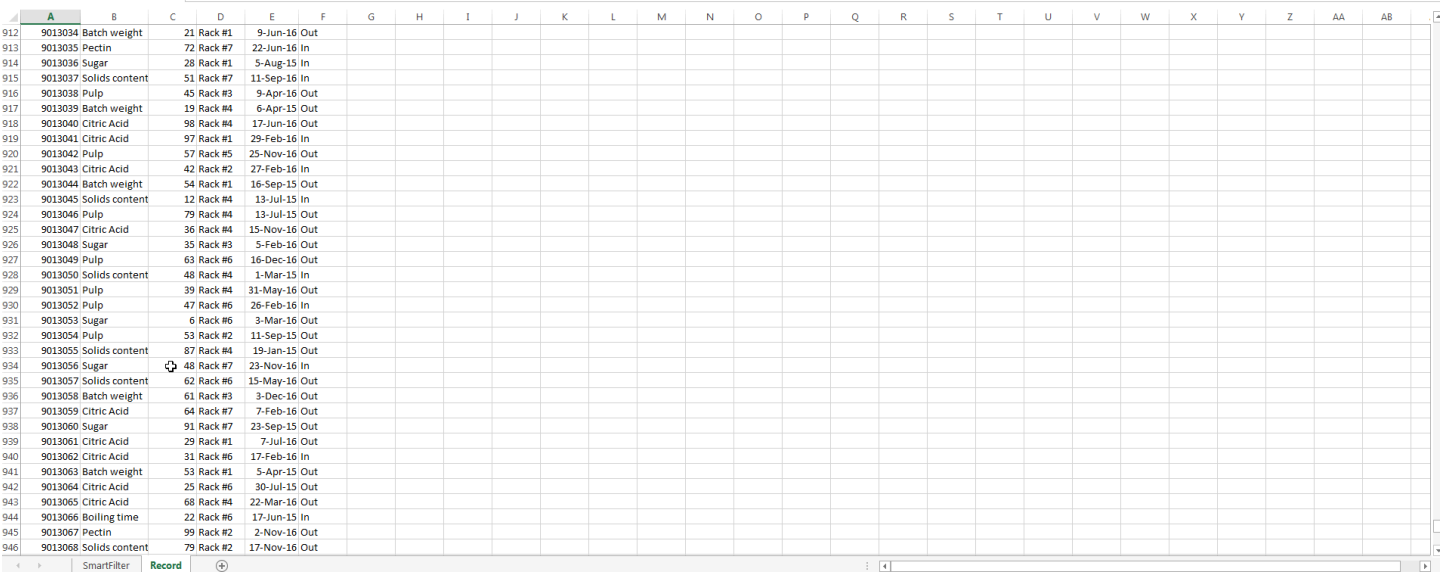
```
    If 1 = 2 Or DummyRange Is Nothing Then '99. If error
Err01General_Functions_Find_Title:
    If IsNeededToExist = True Then MsgBox "Err01General_Functions_Find_Title: Ttile '" &
TitleToFind & "' was not found in sheet '" & InSheet & "'", vbCritical: Call ExcelNormal: On Error
GoTo -1: End
    End If '99. If error
End Function
```

Code for module 2, called "Inventory_Handling"

```
Const TitleDesc As String = "DESCRIPTION"
Const TitleLocation As String = "LOCATION"
Const TitleActn As String = "ACTION"
Const TitleQty As String = "QUANTITY"
Const SheetRecords As String = "Record"
Const SheetSmartFilter As String = "SmartFilter"
Const RowFilter As Long = 2
Const ColDataToPaste As Long = 2
Const RowDataToPaste As Long = 7
Const RangeInResult As String = "K1"
Const RangeOutResult As String = "K2"
Sub Inventory_Filter()
Dim ColDesc As Long: ColDesc = General_Functions_Find_Title(SheetSmartFilter, TitleDesc,
IsNeededToExist:=True, IsWhole:=True).Column
Dim ColLocation As Long: ColLocation = General_Functions_Find_Title(SheetSmartFilter,
TitleLocation, IsNeededToExist:=True, IsWhole:=True).Column
Dim ColActn As Long: ColActn = General_Functions_Find_Title(SheetSmartFilter, TitleActn,
IsNeededToExist:=True, IsWhole:=True).Column
Dim ColQty As Long: ColQty = General_Functions_Find_Title(SheetSmartFilter, TitleQty,
IsNeededToExist:=True, IsWhole:=True).Column
Dim CounterQty As Long
Dim TotalQty As Long
Dim TotalIn As Long
Dim TotalOut As Long
Dim RangeFiltered As Range
    Call Select_Sheet(SheetSmartFilter)
    If Cells(Rows.Count, ColDataToPaste).End(xlUp).Row > RowDataToPaste - 1 Then
Rows(RowDataToPaste & ":" & Cells(Rows.Count, "B").End(xlUp).Row).Delete
    Sheets(SheetRecords).AutoFilterMode = False
    If Cells(RowFilter, ColDesc).Value <> "" Or Cells(RowFilter, ColLocation).Value <> "" Or
Cells(RowFilter, ColActn).Value <> "" Then ' 1. If Cells(RowFilter, ColDesc).Value <> "" Or
Cells(RowFilter, ColLocation).Value <> "" Or Cells(RowFilter, ColActn).Value <> ""
    With Sheets(SheetRecords).UsedRange
    If Sheets(SheetSmartFilter).Cells(RowFilter, ColDesc).Value <> "" Then .AutoFilter
Field:=General_Functions_Find_Title(SheetRecords, TitleDesc, IsNeededToExist:=True,
IsWhole:=True).Column, Criteria1:=Sheets(SheetSmartFilter).Cells(RowFilter, ColDesc).Value
    If Sheets(SheetSmartFilter).Cells(RowFilter, ColLocation).Value <> "" Then .AutoFilter
Field:=General_Functions_Find_Title(SheetRecords, TitleLocation, IsNeededToExist:=True,
IsWhole:=True).Column, Criteria1:=Sheets(SheetSmartFilter).Cells(RowFilter, ColLocation).Value
    If Sheets(SheetSmartFilter).Cells(RowFilter, ColActn).Value <> "" Then .AutoFilter
Field:=General_Functions_Find_Title(SheetRecords, TitleActn, IsNeededToExist:=True,
IsWhole:=True).Column, Criteria1:=Sheets(SheetSmartFilter).Cells(RowFilter, ColActn).Value
    'If we don't use a filter we would need to use a cycle For/to or For/Each Cell in range
    'to determine whether or not the row meets the criteria that we are looking and then
    'save it on an array, collection, dictionary, etc
    'IG: For CounterRow = 2 To TotalRows
    'If Sheets(SheetSmartFilter).Cells(RowFilter, ColDesc).Value <> "" and
Sheets(SheetRecords).cells(CounterRow,ColDescInRecords).Value=
Sheets(SheetSmartFilter).Cells(RowFilter, ColDesc).Value then
    'Redim Preserve MyUnecessaryArray(UnecessaryNumber) ''Save to array:
(UnecessaryNumber)=MyUnecessaryArray. Or in a dictionary, etc. At the end, we would transpose this
```

```
values into the sheet, at the end
    'both are the same, but, just try to see the time invested on each logic.
    If .Cells(1, 1).End(xlDown).Value <> "" Then Set RangeFiltered = .Rows("2:" &
Sheets(SheetRecords).Cells(Rows.Count, "A").End(xlUp).Row).SpecialCells(xlCellTypeVisible)
    'If it is not <>"" means that there was not filtered data!
    If RangeFiltered Is Nothing Then MsgBox "Err01Inventory_Filter: No data was found with the
given criteria!", vbCritical: Call ExcelNormal: End
    RangeFiltered.Copy Destination:=Cells(RowDataToPaste, ColDataToPaste)
    TotalQty = Cells(Rows.Count, ColQty).End(xlUp).Row
    For CounterQty = RowDataToPaste + 1 To TotalQty
    If Cells(CounterQty, ColActn).Value = "In" Then ' 2. If Cells(CounterQty, ColActn).Value = "In"
    TotalIn = Cells(CounterQty, ColQty).Value + TotalIn
    ElseIf Cells(CounterQty, ColActn).Value = "Out" Then ' 2. If Cells(CounterQty, ColActn).Value =
"In"
    TotalOut = Cells(CounterQty, ColQty).Value + TotalOut
    End If ' 2. If Cells(CounterQty, ColActn).Value = "In"
    Next CounterQty
    Range(RangeInResult).Value = TotalIn
    Range(RangeOutResult).Value = -(TotalOut)
    End With
    End If ' 1. If Cells(RowFilter, ColDesc).Value <> "" Or Cells(RowFilter, ColLocation).Value <>
"" Or Cells(RowFilter, ColActn).Value <> ""
End Sub
```

Testing and results:



	A	B	C	D	E	F
912	9013034	Batch weight	21	Rack #1	9-Jun-16	Out
913	9013035	Pectin	72	Rack #7	22-Jun-16	In
914	9013036	Sugar	28	Rack #1	5-Aug-15	In
915	9013037	Solids content	51	Rack #7	11-Sep-16	In
916	9013038	Pulp	45	Rack #3	9-Apr-16	Out
917	9013039	Batch weight	19	Rack #4	6-Apr-15	Out
918	9013040	Citric Acid	98	Rack #4	17-Jun-16	Out
919	9013041	Citric Acid	97	Rack #1	29-Feb-16	In
920	9013042	Pulp	57	Rack #5	25-Nov-16	Out
921	9013043	Citric Acid	42	Rack #2	27-Feb-16	In
922	9013044	Batch weight	54	Rack #1	16-Sep-15	Out
923	9013045	Solids content	12	Rack #4	13-Jul-15	In
924	9013046	Pulp	79	Rack #4	13-Jul-15	Out
925	9013047	Citric Acid	36	Rack #4	15-Nov-16	Out
926	9013048	Sugar	35	Rack #3	5-Feb-16	Out
927	9013049	Pulp	63	Rack #6	16-Dec-16	Out
928	9013050	Solids content	48	Rack #4	1-Mar-15	In
929	9013051	Pulp	39	Rack #4	31-May-16	Out
930	9013052	Pulp	47	Rack #6	26-Feb-16	In
931	9013053	Sugar	6	Rack #6	3-Mar-16	Out
932	9013054	Pulp	53	Rack #2	11-Sep-15	Out
933	9013055	Solids content	87	Rack #4	19-Jan-15	Out
934	9013056	Sugar	48	Rack #7	23-Nov-16	In
935	9013057	Solids content	62	Rack #6	15-May-16	Out
936	9013058	Batch weight	61	Rack #3	3-Dec-16	Out
937	9013059	Citric Acid	64	Rack #7	7-Feb-16	Out
938	9013060	Sugar	91	Rack #7	23-Sep-15	Out
939	9013061	Citric Acid	29	Rack #1	7-Jul-16	Out
940	9013062	Citric Acid	31	Rack #6	17-Feb-16	In
941	9013063	Batch weight	53	Rack #1	5-Apr-15	Out
942	9013064	Citric Acid	25	Rack #6	30-Jul-15	Out
943	9013065	Citric Acid	68	Rack #4	22-Mar-16	Out
944	9013066	Boiling time	22	Rack #6	17-Jun-15	In
945	9013067	Pectin	99	Rack #2	2-Nov-16	Out
946	9013068	Solids content	79	Rack #2	17-Nov-16	Out

SmartFilter | Record

As we saw in the previous image, this task has been achieved easily. By using ***autofilters*** a solution was provided that just ***takes seconds to compute, is easy to explain to the user*** -since s/he is familiar with this command- and ***took a few lines to the coder.***

Chapter 20: Application object

Section 20.1: Simple Application Object example: Display Excel and VBE Version

```
Sub DisplayExcelVersions()

    MsgBox "The version of Excel is " & Application.Version
    MsgBox "The version of the VBE is " & Application.VBE.Version

End Sub
```

The use of the Application.Version property is useful for ensuring code only operates on a compatible version of Excel.

Section 20.2: Simple Application Object example: Minimize the Excel window

This code uses the top level **Application** object to minimize the main Excel window.

```
Sub MinimizeExcel()

    Application.WindowState = xlMinimized

End Sub
```

Chapter 21: Charts and Charting

Section 21.1: Creating a Chart with Ranges and a Fixed Name

Charts can be created by working directly with the `Series` object that defines the chart data. In order to get to the `Series` without an exisitng chart, you create a `ChartObject` on a given `Worksheet` and then get the `Chart` object from it. The upside of working with the `Series` object is that you can set the `Values` and `XValues` by referring to `Range` objects. These data properties will properly define the `Series` with references to those ranges. The downside to this approach is that the same conversion is not handled when setting the `Name`; it is a fixed value. It will not adjust with the underlying data in the original `Range`. Checking the `SERIES` formula and it is obvious that the name is fixed. This must be handled by creating the `SERIES` formula directly.

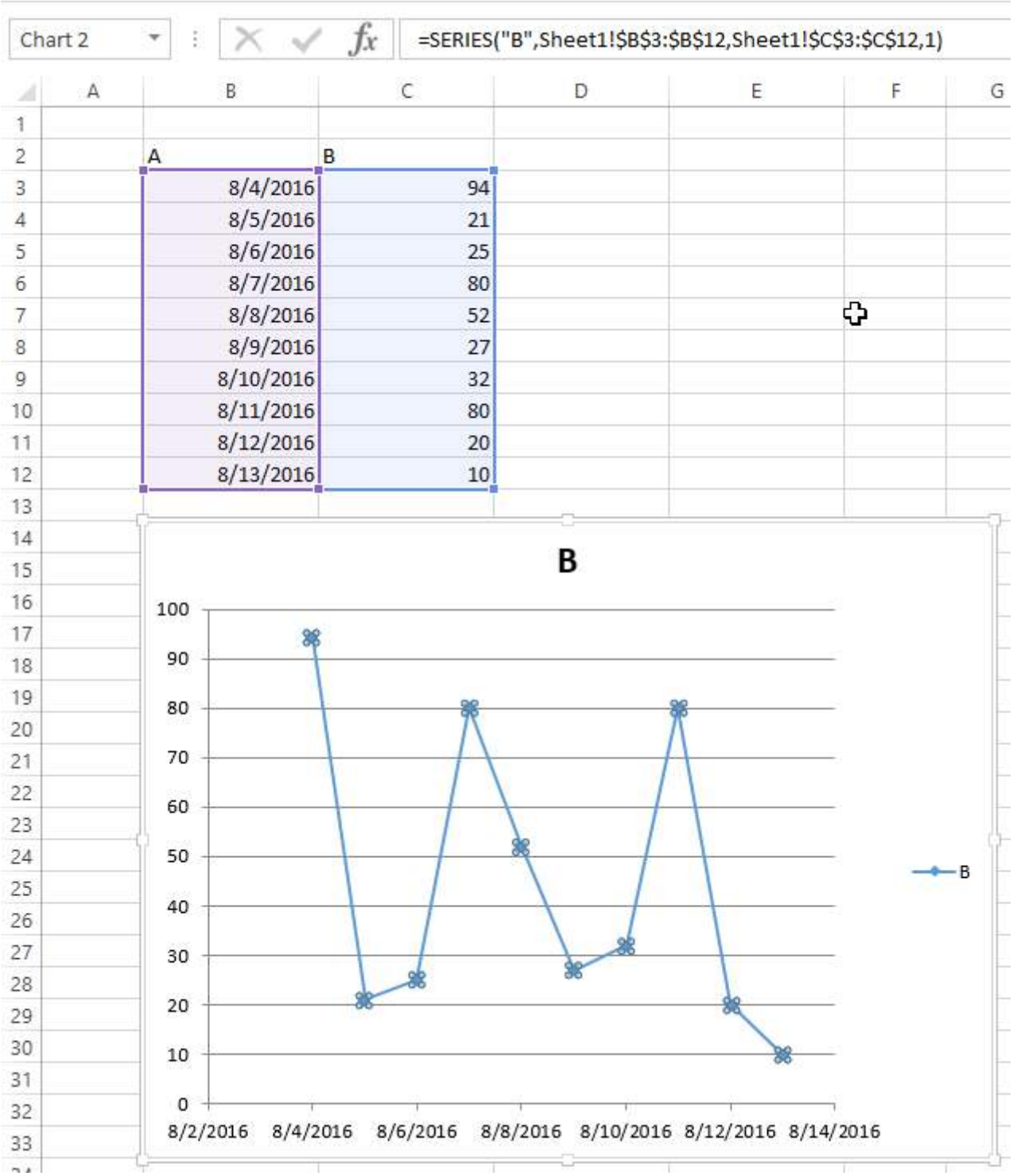
Code used to create chart

Note that this code contains extra variable declarations for the `Chart` and `Worksheet`. These can be omitted if they're not used. They can be useful however if you are modifying the style or any other chart properties.

```
Sub CreateChartWithRangesAndFixedName()

    Dim xData As Range
    Dim yData As Range
    Dim serName As Range

    'set the ranges to get the data and y value label
    Set xData = Range("B3:B12")
    Set yData = Range("C3:C12")
    Set serName = Range("C2")

    'get reference to ActiveSheet
    Dim sht As Worksheet
    Set sht = ActiveSheet

    'create a new ChartObject at position (48, 195) with width 400 and height 300
    Dim chtObj As ChartObject
    Set chtObj = sht.ChartObjects.Add(48, 195, 400, 300)

    'get reference to chart object
    Dim cht As Chart
    Set cht = chtObj.Chart

    'create the new series
    Dim ser As Series
    Set ser = cht.SeriesCollection.NewSeries

    ser.Values = yData
    ser.XValues = xData
    ser.Name = serName

    ser.ChartType = xlXYScatterLines

End Sub
```

Original data/ranges and resulting `Chart` after code runs

Note that the `SERIES` formula includes a `"B"` for the series name instead of a reference to the `Range` that created it.



Section 21.2: Creating an empty Chart

The starting point for the vast majority of charting code is to create an empty `Chart`. Note that this `Chart` is subject to the default chart template that is active and may not actually be empty (if the template has been modified).

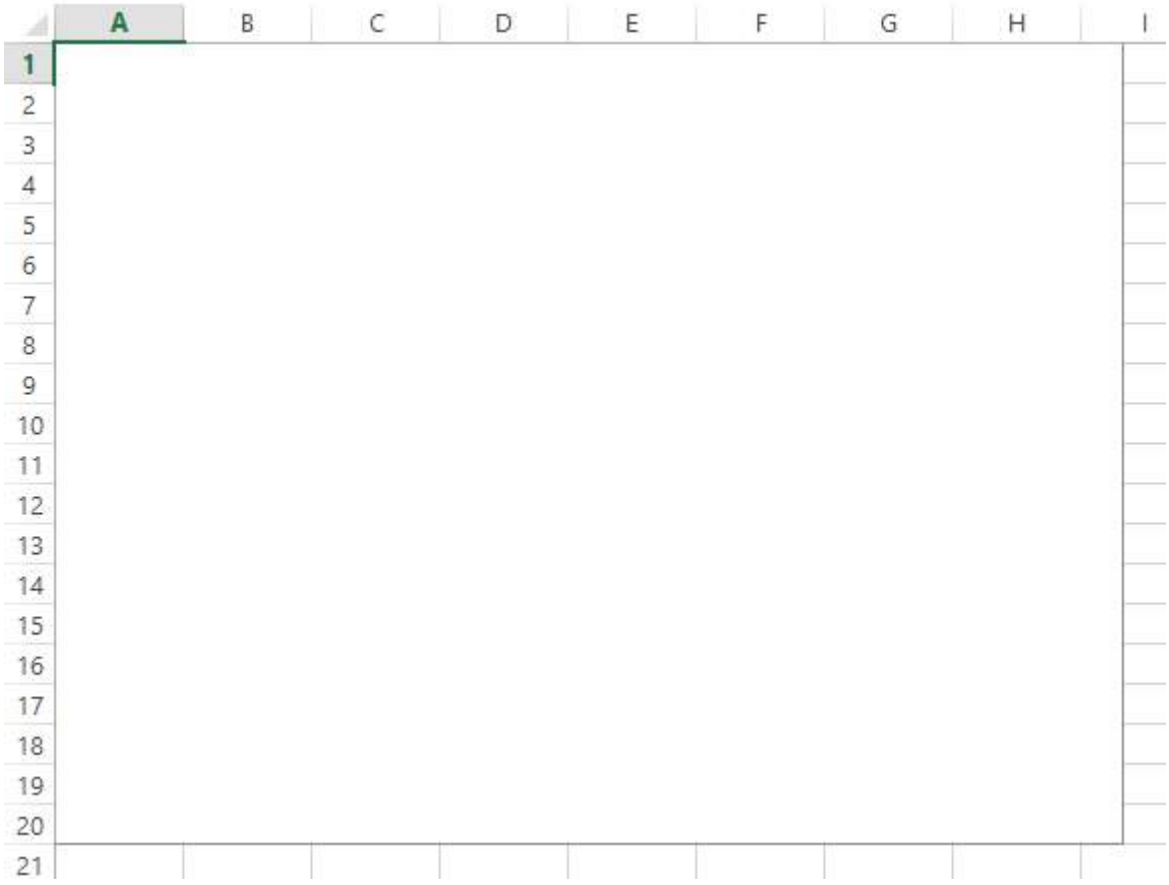
The key to the `ChartObject` is determining its location. The syntax for the call is `ChartObjects.Add(Left, Top, Width, Height)`. Once the `ChartObject` is created, you can use its `Chart` object to actually modify the chart. The `ChartObject` behaves more like a `Shape` to position the chart on the sheet.

Code to create an empty chart

```
Sub CreateEmptyChart()

    'get reference to ActiveSheet
```

```
    Dim sht As Worksheet
    Set sht = ActiveSheet

    'create a new ChartObject at position (0, 0) with width 400 and height 300
    Dim chtObj As ChartObject
    Set chtObj = sht.ChartObjects.Add(0, 0, 400, 300)

    'get refernce to chart object
    Dim cht As Chart
    Set cht = chtObj.Chart

    'additional code to modify the empty chart
    '...

End Sub
```

Resulting Chart



Section 21.3: Create a Chart by Modifying the SERIES formula

For complete control over a new `Chart` and `Series` object (especially for a dynamic `Series` name), you must resort to modifying the `SERIES` formula directly. The process to set up the `Range` objects is straightforward and the main hurdle is simply the string building for the `SERIES` formula.

The `SERIES` formula takes the following syntax:

```
=SERIES(Name,XValues,Values,Order)
```

These contents can be supplied as references or as array values for the data items. `Order` represents the series position within the chart. Note that the references to the data will not work unless they are fully qualified with the sheet name. For an example of a working formula, click any existing series and check the formula bar.

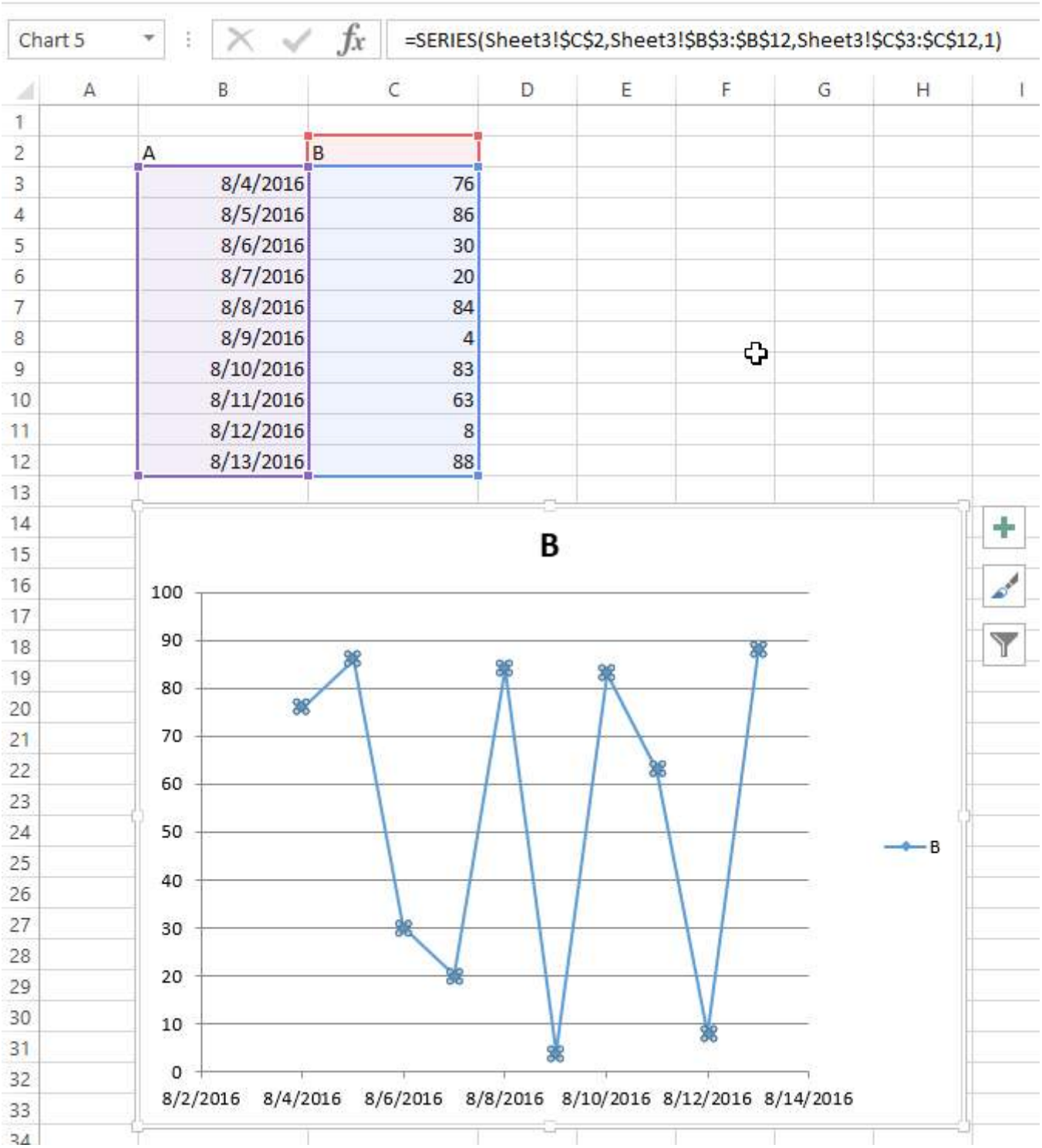
Code to create a chart and set up data using the `SERIES` formula

Note that the string building to create the `SERIES` formula uses `.Address(,,,True)`. This ensures that the *external* Range reference is used so that a fully qualified address with the sheet name is included. You **will get an error if the sheet name is excluded**.

```
Sub CreateChartUsingSeriesFormula()

    Dim xData As Range
    Dim yData As Range
    Dim serName As Range

    'set the ranges to get the data and y value label
    Set xData = Range("B3:B12")
    Set yData = Range("C3:C12")
    Set serName = Range("C2")

    'get reference to ActiveSheet
    Dim sht As Worksheet
    Set sht = ActiveSheet

    'create a new ChartObject at position (48, 195) with width 400 and height 300
    Dim chtObj As ChartObject
    Set chtObj = sht.ChartObjects.Add(48, 195, 400, 300)

    'get refernce to chart object
    Dim cht As Chart
    Set cht = chtObj.Chart

    'create the new series
    Dim ser As Series
    Set ser = cht.SeriesCollection.NewSeries

    'set the SERIES formula
    '=SERIES(name, xData, yData, plotOrder)

    Dim formulaValue As String
    formulaValue = "=SERIES(" & _
        serName.Address(, , , True) & "," & _
        xData.Address(, , , True) & "," & _
        yData.Address(, , , True) & ",1)"

    ser.Formula = formulaValue
    ser.ChartType = xlXYScatterLines

End Sub
```

Original data and resulting chart

Note that for this chart, the series name is properly set with a range to the desired cell. This means that updates will propagate to the `Chart`.



Section 21.4: Arranging Charts into a Grid

A common chore with charts in Excel is standardizing the size and layout of multiple charts on a single sheet. If done manually, you can hold down `ALT` while resizing or moving the chart to "stick" to cell boundaries. This works for a couple charts, but a VBA approach is much simpler.

Code to create a grid

This code will create a grid of charts starting at a given (Top, Left) position, with a defined number of columns, and a defined common chart size. The charts will be placed in the order they were created and wrap around the edge to form a new row.

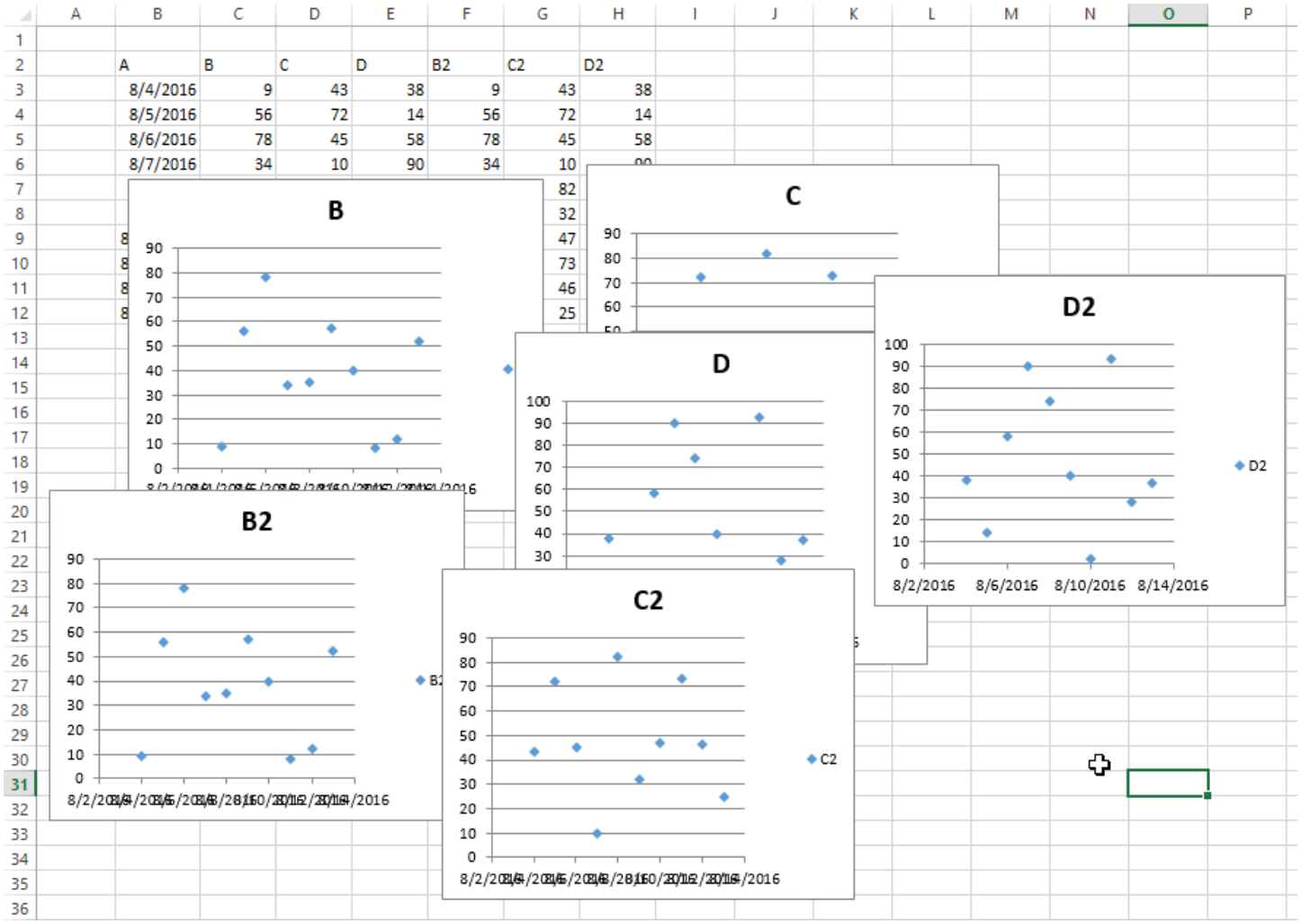
```
Sub CreateGridOfCharts()
```

```
    Dim int_cols As Integer
    int_cols = 3

    Dim cht_width As Double
    cht_width = 250

    Dim cht_height As Double
    cht_height = 200

    Dim offset_vertical As Double
    offset_vertical = 195

    Dim offset_horz As Double
    offset_horz = 40

    Dim sht As Worksheet
    Set sht = ActiveSheet

    Dim count As Integer
    count = 0

    'iterate through ChartObjects on current sheet
    Dim cht_obj As ChartObject
    For Each cht_obj In sht.ChartObjects

        'use integer division and Mod to get position in grid
        cht_obj.Top = (count \ int_cols) * cht_height + offset_vertical
        cht_obj.Left = (count Mod int_cols) * cht_width + offset_horz
        cht_obj.Width = cht_width
        cht_obj.Height = cht_height

        count = count + 1

    Next cht_obj
End Sub
```

Result with several charts

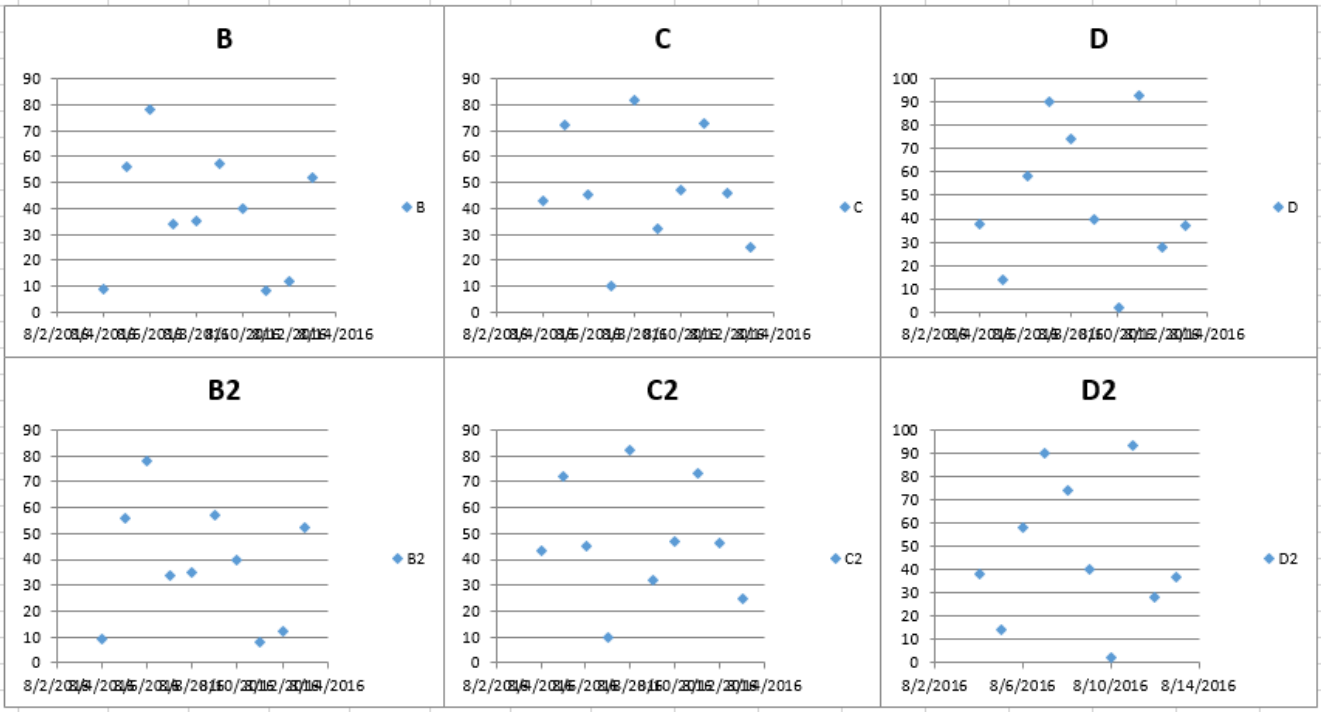
These pictures show the original random layout of charts and the resulting grid from running the code above.

Before


A
B
C
D
B2
C2
D2
8/4/2016 9 43 38 9 43 38
8/5/2016 56 72 14 56 72 14
8/6/2016 78 45 58 78 45 58
8/7/2016 34 10 90 34 10
B
C
D
B2
C2
D2
8/2/2016
8/6/2016
8/10/2016
8/14/2016

After

	A	B	C	D	E	F	G
1							
2		A	B	C	D	B2	C2	D2
3		8/4/2016	9	43	38	9	43	38
4		8/5/2016	56	72	14	56	72	14
5		8/6/2016	78	45	58	78	45	58
6		8/7/2016	34	10	90	34	10	90
7		8/8/2016	35	82	74	35	82	74
8		8/9/2016	57	32	40	57	32	40
9		8/10/2016	40	47	2	40	47	2
10		8/11/2016	8	73	93	8	73	93
11		8/12/2016	12	46	28	12	46	28
12		8/13/2016	52	25	37	52	25	37
13								



Chapter 22: CustomDocumentProperties in practice

Using CustomDocumentProperties (CDPs) is a good method to store user defined values in a relatively safe way within the same work book, but avoiding to show related cell values simply in an unprotected work sheet *).

Note: CDPs represent a separate collection comparable to BuiltInDocumentProperties, but allow to create user defined property names of your own instead of a fixed collection.

*) Alternatively, you could enter values also in a hidden or "very hidden" workbook.

Section 22.1: Organizing new invoice numbers

Incrementing an invoice number and saving its value is a frequent task. Using CustomDocumentProperties (CDPs) is a good method to store such numbers in a relatively safe way within the same work book, but avoiding to show related cell values simply in an unprotected work sheet.

Additional hint:

Alternatively, you could enter values also in a hidden worksheet or even a so called "very hidden" worksheet (see Using xlVeryHidden Sheets. Of course, it's possible to save data also to external files (e.g. ini file, csv or any other type) or the registry.

Example content:

The example below shows

- a function NextInvoiceNo that sets and returns the next invoice number,
- a procedure DeleteInvoiceNo, that deletes the invoice CDP completely, as well as
- a procedure showAllCDPs listing the complete CDPs collection with all names. Not using VBA, you can also list them via the workbook's information: Info | Properties [DropDown:] | Advanced Properties | Custom

You can get and set the next invoice number (last no plus one) simply by calling the above mentioned function, returning a string value in order to facilitate adding prefixes. "InvoiceNo" is implicitly used as CDP name in all procedures.

```
Dim sNumber As String
sNumber = NextInvoiceNo ()
```

Example code:

```
Option Explicit

Sub Test()
  Dim sNumber As String
  sNumber = NextInvoiceNo()
  MsgBox "New Invoice No: " & sNumber, vbInformation, "New Invoice Number"
End Sub

Function NextInvoiceNo() As String
' Purpose: a) Set Custom Document Property (CDP) "InvoiceNo" if not yet existing
'          b) Increment CDP value and return new value as string
' Declarations
  Dim prop As Object
```

```
  Dim ret  As String
  Dim wb   As Workbook
' Set workbook and CDPs
  Set wb = ThisWorkbook
  Set prop = wb.CustomDocumentProperties

  ' ------------------------------------------------------
  ' Generate new CDP "InvoiceNo" if not yet existing
  ' ------------------------------------------------------
    If Not CDPExists("InvoiceNo") Then
    '  set temporary starting value "0"
       prop.Add "InvoiceNo", False, msoPropertyTypeString, "0"
    End If

  ' ------------------------------------------------------
  ' Increment invoice no and return function value as string
  ' ------------------------------------------------------
       ret = Format(Val(prop("InvoiceNo")) + 1, "0")
  ' a) Set CDP "InvoiceNo" = ret
       prop("InvoiceNo").value = ret
  ' b) Return function value
       NextInvoiceNo = ret
End Function

Private Function CDPExists(sCDPName As String) As Boolean
' Purpose: return True if custom document property (CDP) exists
' Method: loop thru CustomDocumentProperties collection and check if name parameter exists
' Site: cf.
http://stackoverflow.com/questions/23917977/alternatives-to-public-variables-in-vba/23918236#23918236
' vgl.:
https://answers.microsoft.com/en-us/msoffice/forum/msoffice_word-mso_other/using-customdocumentproper
ties-with-vba/91ef15eb-b089-4c9b-a8a7-1685d073fb9f
' Declarations
  Dim cdp As Variant      ' element of CustomDocumentProperties Collection
  Dim boo As Boolean      ' boolean value showing element exists
  For Each cdp In ThisWorkbook.CustomDocumentProperties
    If LCase(cdp.Name) = LCase(sCDPName) Then
       boo = True      ' heureka
       Exit For        ' exit loop
    End If
  Next
  CDPExists = boo          ' return value to function
End Function


Sub DeleteInvoiceNo()
' Declarations
  Dim wb     As Workbook
  Dim prop   As Object
' Set workbook and CDPs
  Set wb = ThisWorkbook
  Set prop = wb.CustomDocumentProperties

' ----------------------
' Delete CDP "InvoiceNo"
' ----------------------
 If CDPExists("InvoiceNo") Then
    prop("InvoiceNo").Delete
 End If
```

End Sub

```vba
Sub showAllCDPs()
' Purpose: Show all CustomDocumentProperties (CDP) and values (if set)
' Declarations
  Dim wb      As Workbook
  Dim cdp     As Object

  Dim i       As Integer
  Dim maxi    As Integer
  Dim s       As String
' Set workbook and CDPs
  Set wb = ThisWorkbook
  Set cdp = wb.CustomDocumentProperties
' Loop thru CDP getting name and value
  maxi = cdp.Count
  For i = 1 To maxi
    On Error Resume Next     ' necessary in case of unset value
    s = s & Chr(i + 96) & ") " & _
            cdp(i).Name & "=" & cdp(i).value & vbCr
  Next i
' Show result string
  Debug.Print s
End Sub
```

Chapter 23: PowerPoint Integration Through VBA

Section 23.1: The Basics: Launching PowerPoint from VBA

While there are many parameters that can be changed and variations that can be added depending on the desired functionality, this example lays out the basic framework for launching PowerPoint.

> **Note:** This code requires that the PowerPoint reference has been added to the active VBA Project. See the References Documentation entry to learn how to enable the reference.

First, define variables for the Application, Presentation, and Slide Objects. While this can be done with late binding, it is always best to use early binding when applicable.

```
Dim PPApp As PowerPoint.Application
Dim PPPres As PowerPoint.Presentation
Dim PPSlide As PowerPoint.Slide
```

Next, open or create a new instance of the PowerPoint application. Here, the `On Error Resume Next` call is used to avoid an error being thrown by `GetObject` if PowerPoint has not yet been opened. See the Error Handling example of the Best Practices Topic for a more detailed explanation.

```
'Open PPT if not running, otherwise select active instance
On Error Resume Next
Set PPApp = GetObject(, "PowerPoint.Application")
On Error GoTo ErrHandler
If PPApp Is Nothing Then
    'Open PowerPoint
    Set PPApp = CreateObject("PowerPoint.Application")
    PPApp.Visible = True
End If
```

Once the application has been launched, a new presentation and subsequently contained slide is generated for use.

```
'Generate new Presentation and slide for graphic creation
Set PPPres = PPApp.Presentations.Add
Set PPSlide = PPPres.Slides.Add(1, ppLayoutBlank)

'Here, the slide type is set to the 4:3 shape with slide numbers enabled and the window
'maximized on the screen.  These properties can, of course, be altered as needed

PPApp.ActiveWindow.ViewType = ppViewSlide
PPPres.PageSetup.SlideOrientation = msoOrientationHorizontal
PPPres.PageSetup.SlideSize = ppSlideSizeOnScreen
PPPres.SlideMaster.HeadersFooters.SlideNumber.Visible = msoTrue
PPApp.ActiveWindow.WindowState = ppWindowMaximized
```

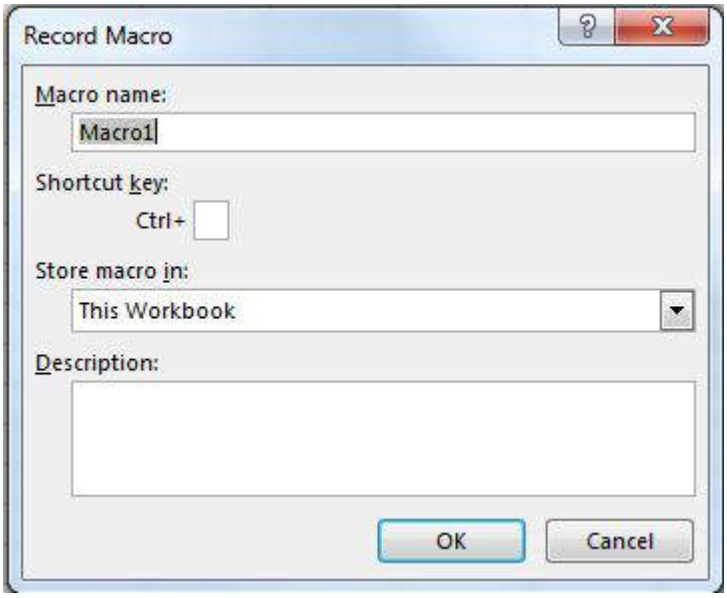
Upon completion of this code, a new PowerPoint window with a blank slide will be open. By using the object variables, shapes, text, graphics, and excel ranges can be added as desired

Chapter 24: How to record a Macro

Section 24.1: How to record a Macro

The easiest way to record a macro is the button in the lower left corner of Excel looks like this: 

When you click on this you will get a pop-up asking you to name the Macro and decide if you want to have a shortcut key. Also, asks where to store the macro and for a description. You can choose any name you want, no spaces are allowed.



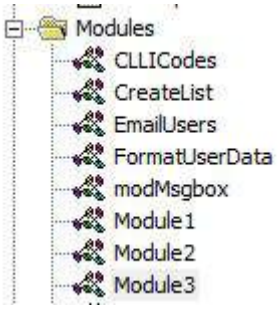
If you want to have a shortcut assigned to your macro for quick use choose a letter that you will remember so that you can quickly and easily use the macro over and over.

You can store the macro in "This Workbook," "New Workbook," or "Personal Macro Workbook." If you want the macro you're about to record to be available only in the current workbook, choose "This Workbook." If you want it saved to a brand new workbook, choose "New Workbook." And if you want the macro to be available to any workbook you open, choose "Personal Macro Workbook."

After you have filled out this pop-up click on "Ok".

Then perform whatever actions you want to repeat with the macro. When finished click the same button to stop recording. It now looks like this:

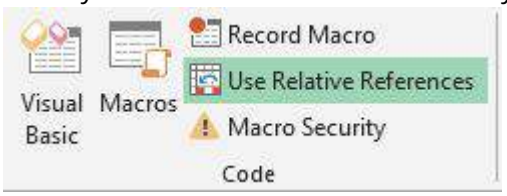
Now you can go to the Developer Tab and open Visual Basic. (or use Alt + F11)

You will now have a new Module under the Modules folder. 

The newest module will contain the macro you just recorded. Double-click on it to bring it up.

I did a simple copy and paste:

```
Sub Macro1()
'
' Macro1 Macro
'

'
    Selection.Copy
    Range("A12").Select
    ActiveSheet.Paste
End Sub
```

If you don't want it to always paste into "A12" you can use Relative References by checking the "Use Relative References" box on the Developer Tab: 

Following the same steps as before will now turn the Macro into this:

```
Sub Macro2()
'
' Macro2 Macro
'

'
    Selection.Copy
    ActiveCell.Offset(11, 0).Range("A1").Select
    ActiveSheet.Paste
End Sub
```

Still copying the value from "A1" into a cell 11 rows down, but now you can perform the same macro with any starting cell and the value from that cell will be copied to the cell 11 rows down.

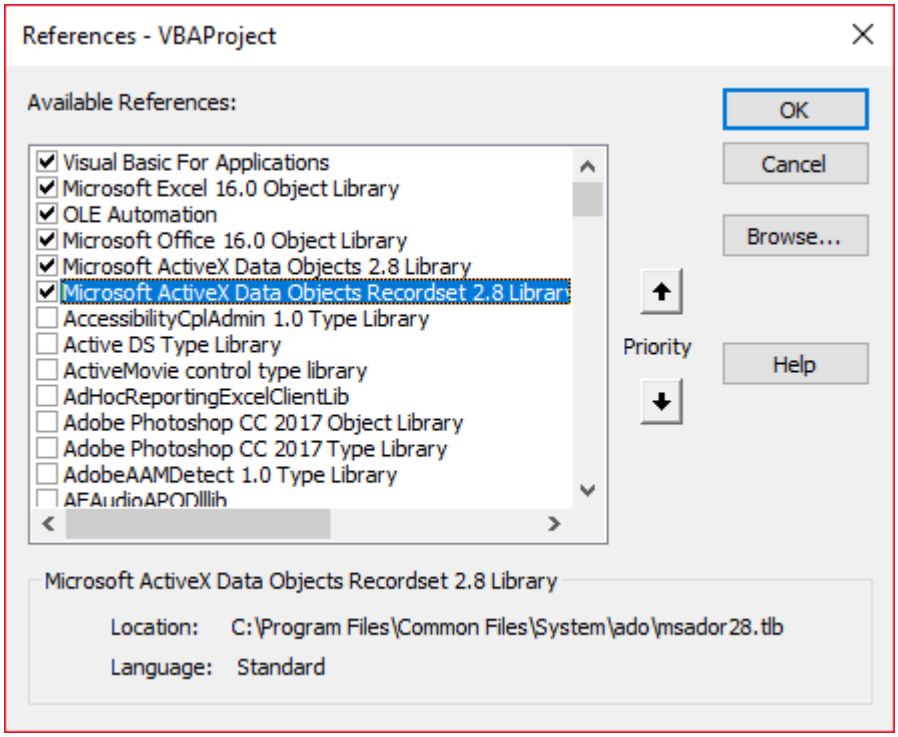
Chapter 25: SQL in Excel VBA - Best Practices

Section 25.1: How to use ADODB.Connection in VBA?

Requirements:

Add following references to the project:

- Microsoft ActiveX Data Objects 2.8 Library
- Microsoft ActiveX Data Objects Recordset 2.8 Library



Declare variables

```
Private mDataBase As New ADODB.Connection
Private mRS As New ADODB.Recordset
Private mCmd As New ADODB.Command
```

Create connection

a. with Windows Authentication

```
Private Sub OpenConnection(pServer As String, pCatalog As String)
    Call mDataBase.Open("Provider=SQLOLEDB;Initial Catalog=" & pCatalog & ";Data Source=" & pServer
& ";Integrated Security=SSPI")
    mCmd.ActiveConnection = mDataBase
End Sub
```

b. with SQL Server Authentication

```
Private Sub OpenConnection2(pServer As String, pCatalog As String, pUser As String, pPsw As String)
    Call mDataBase.Open("Provider=SQLOLEDB;Initial Catalog=" & pCatalog & ";Data Source=" & pServer
& ";Integrated Security=SSPI;User ID=" & pUser & ";Password=" & pPsw)
    mCmd.ActiveConnection = mDataBase
End Sub
```

Execute sql command

```
Private Sub ExecuteCmd(sql As String)
    mCmd.CommandText = sql
    Set mRS = mCmd.Execute
End Sub
```

Read data from record set

```
Private Sub ReadRS()
    Do While Not (mRS.EOF)
        Debug.Print "ShipperID: " & mRS.Fields("ShipperID").Value & " CompanyName: " &
mRS.Fields("CompanyName").Value & " Phone: " & mRS.Fields("Phone").Value
        Call mRS.MoveNext
    Loop
End Sub
```

Close connection

```
Private Sub CloseConnection()
    Call mDataBase.Close
    Set mRS = Nothing
    Set mCmd = Nothing
    Set mDataBase = Nothing
End Sub
```

How to use it?

```
Public Sub Program()
    Call OpenConnection("ServerName", "NORTHWND")
    Call ExecuteCmd("INSERT INTO [NORTHWND].[dbo].[Shippers]([CompanyName],[Phone]) Values ('speedy
shipping','(503) 555-1234')")
    Call ExecuteCmd("SELECT * FROM [NORTHWND].[dbo].[Shippers]")
    Call ReadRS
    Call CloseConnection
End Sub
```

Result

ShipperID: 1 CompanyName: Speedy Express Phone: (503) 555-9831

ShipperID: 2 CompanyName: United Package Phone: (503) 555-3199

ShipperID: 3 CompanyName: Federal Shipping Phone: (503) 555-9931

ShipperID: 4 CompanyName: speedy shipping Phone: (503) 555-1234

Chapter 26: Excel-VBA Optimization

Excel-VBA Optimization refers also to coding better error handling by documentation and additional details. This is shown here.

Section 26.1: Optimizing Error Search by Extended Debugging

Using Line Numbers ... and documenting them in case of error ("The importance of seeing Erl")

Detecting which line raises an error is a substantial part of any debugging and narrows the search for the cause. To document identified error lines with a short description completes a successful error tracking, at best together with the names of module and procedure. The example below saves these data to a log file.

Back ground

The error object returns error number (Err.Number) and error description (Err.Description), but doesn't explicitly respond to the question where to locate the error. The **Erl** function, however, does, but on condition that you add **line numbers)* to the code (BTW one of several other concessions to former Basic times).

If there are no error lines at all, then the Erl function returns 0, if numbering is incomplete you'll get the procedure's last preceding line number.

```
Option Explicit


Public Sub MyProc1()
Dim i As Integer
Dim j As Integer
On Error GoTo LogErr
10     j = 1 / 0     ' raises an error
okay:
Debug.Print "i=" & i
Exit Sub

LogErr:
MsgBox LogErrors("MyModule", "MyProc1", Err), vbExclamation, "Error " & Err.Number
Stop
Resume Next
End Sub

Public Function LogErrors( _
           ByVal sModule As String, _
           ByVal sProc As String, _
           Err As ErrObject) As String
' Purpose: write error number, description and Erl to log file and return error text
  Dim sLogFile As String: sLogFile = ThisWorkbook.Path & Application.PathSeparator &
"LogErrors.txt"
  Dim sLogTxt  As String
  Dim lFile    As Long

' Create error text
  sLogTxt = sModule & "|" & sProc & "|Erl " & Erl & "|Err " & Err.Number & "|" & Err.Description

  On Error Resume Next
  lFile = FreeFile

  Open sLogFile For Append As lFile
  Print #lFile, Format$(Now(), "yy.mm.dd hh:mm:ss "); sLogTxt
```

```
      Print #lFile,
  Close lFile
' Return error text
  LogErrors = sLogTxt
 End Function
```

'Additional Code to show log file

```
Sub ShowLogFile()
Dim sLogFile As String: sLogFile = ThisWorkbook.Path & Application.PathSeparator & "LogErrors.txt"

On Error GoTo LogErr
Shell "notepad.exe " & sLogFile, vbNormalFocus

okay:
On Error Resume Next
Exit Sub

LogErr:
MsgBox LogErrors("MyModule", "ShowLogFile", Err), vbExclamation, "Error No " & Err.Number
Resume okay
End Sub
```

Section 26.2: Disabling Worksheet Updating

Disabling calculation of the worksheet can decrease running time of the macro significantly. Moreover, disabling events, screen updating and page breaks would be beneficial. Following `Sub` can be used in any macro for this purpose.

```
Sub OptimizeVBA(isOn As Boolean)
    Application.Calculation = IIf(isOn, xlCalculationManual, xlCalculationAutomatic)
    Application.EnableEvents = Not(isOn)
    Application.ScreenUpdating = Not(isOn)
    ActiveSheet.DisplayPageBreaks = Not(isOn)
End Sub
```

For optimization follow the below pseudo-code:

```
Sub MyCode()

    OptimizeVBA True

    'Your code goes here

    OptimizeVBA False

End Sub
```

Section 26.3: Row Deletion - Performance

- Deleting rows is slow, specially when looping through cells and deleting rows, one by one
- A different approach is using an AutoFilter to hide the rows to be deleted
- Copy the visible range and Paste it into a new WorkSheet
- Remove the initial sheet entirely

- With this method, the more rows to delete, the faster it will be

Example:

```
Option Explicit

'Deleted rows: 775,153, Total Rows: 1,000,009, Duration: 1.87 sec

Public Sub DeleteRows()
    Dim oldWs As Worksheet, newWs As Worksheet, wsName As String, ur As Range

    Set oldWs = ThisWorkbook.ActiveSheet
    wsName = oldWs.Name
    Set ur = oldWs.Range("F2", oldWs.Cells(oldWs.Rows.Count, "F").End(xlUp))

    Application.ScreenUpdating = False
    Set newWs = Sheets.Add(After:=oldWs)    'Create a new WorkSheet

    With ur    'Copy visible range after Autofilter (modify Criteria1 and 2 accordingly)
        .AutoFilter Field:=1, Criteria1:="<>0", Operator:=xlAnd, Criteria2:="<>"
        oldWs.UsedRange.Copy
    End With
    'Paste all visible data into the new WorkSheet (values and formats)
    With newWs.Range(oldWs.UsedRange.Cells(1).Address)
        .PasteSpecial xlPasteColumnWidths
        .PasteSpecial xlPasteAll
        newWs.Cells(1, 1).Select: newWs.Cells(1, 1).Copy
    End With

    With Application
        .CutCopyMode = False
        .DisplayAlerts = False
            oldWs.Delete
        .DisplayAlerts = True
        .ScreenUpdating = True
    End With
    newWs.Name = wsName
End Sub
```

Section 26.4: Disabling All Excel Functionality Before executing large macros

The procedures bellow will temporarily disable all Excel features at WorkBook and WorkSheet level

- FastWB() is a toggle that accepts On or Off flags
- FastWS() accepts an Optional WorkSheet object, or none
- If the ws parameter is missing it will turn all features on and off for all WorkSheets in the collection
    - A custom type can be used to capture all settings before turning them off
    - At the end of the process, the initial settings can be restored

```
Public Sub FastWB(Optional ByVal opt As Boolean = True)
    With Application
        .Calculation = IIf(opt, xlCalculationManual, xlCalculationAutomatic)
        If .DisplayAlerts <> Not opt Then .DisplayAlerts = Not opt
```

```
        If .DisplayStatusBar <> Not opt Then .DisplayStatusBar = Not opt
        If .EnableAnimations <> Not opt Then .EnableAnimations = Not opt
        If .EnableEvents <> Not opt Then .EnableEvents = Not opt
        If .ScreenUpdating <> Not opt Then .ScreenUpdating = Not opt
    End With
    FastWS , opt
End Sub
```

```
Public Sub FastWS(Optional ByVal ws As Worksheet, Optional ByVal opt As Boolean = True)
    If ws Is Nothing Then
        For Each ws In Application.ThisWorkbook.Sheets
            OptimiseWS ws, opt
        Next
    Else
        OptimiseWS ws, opt
    End If
End Sub
Private Sub OptimiseWS(ByVal ws As Worksheet, ByVal opt As Boolean)
    With ws
        .DisplayPageBreaks = False
        .EnableCalculation = Not opt
        .EnableFormatConditionsCalculation = Not opt
        .EnablePivotTable = Not opt
    End With
End Sub
```

Restore all Excel settings to default

```
Public Sub XlResetSettings()    'default Excel settings
    With Application
        .Calculation = xlCalculationAutomatic
        .DisplayAlerts = True
        .DisplayStatusBar = True
        .EnableAnimations = False
        .EnableEvents = True
        .ScreenUpdating = True
        Dim sh As Worksheet
        For Each sh In Application.ThisWorkbook.Sheets
            With sh
                .DisplayPageBreaks = False
                .EnableCalculation = True
                .EnableFormatConditionsCalculation = True
                .EnablePivotTable = True
            End With
        Next
    End With
End Sub
```

Section 26.5: Checking time of execution

Different procedures can give out the same result, but they would use different processing time. In order to check out which one is faster, a code like this can be used:

```
time1 = Timer

For Each iCell In MyRange
   iCell = "text"
Next iCell
```

```
time2 = Timer

For i = 1 To 30
   MyRange.Cells(i) = "text"
Next i

time3 = Timer

debug.print "Proc1 time: " & cStr(time2-time1)
debug.print "Proc2 time: " & cStr(time3-time2)
```

MicroTimer:

```
Private Declare PtrSafe Function getFrequency Lib "Kernel32" Alias "QueryPerformanceFrequency"
(cyFrequency As Currency) As Long
Private Declare PtrSafe Function getTickCount Lib "Kernel32" Alias "QueryPerformanceCounter"
(cyTickCount As Currency) As Long

Function MicroTimer() As Double
    Dim cyTicks1 As Currency
    Static cyFrequency As Currency

    MicroTimer = 0
    If cyFrequency = 0 Then getFrequency cyFrequency        'Get frequency
    getTickCount cyTicks1                                   'Get ticks
    If cyFrequency Then MicroTimer = cyTicks1 / cyFrequency 'Returns Seconds
End Function
```

Section 26.6: Using With blocks

Using with blocks can accelerate the process of running a macro. Instead writing a range, chart name, worksheet, etc. you can use with-blocks like below;

```
With ActiveChart
    .Parent.Width = 400
    .Parent.Height = 145
    .Parent.Top = 77.5 + 165 * step - replacer * 15
    .Parent.Left = 5
End With
```

Which is faster than this:

```
ActiveChart.Parent.Width = 400
ActiveChart.Parent.Height = 145
ActiveChart.Parent.Top = 77.5 + 165 * step - replacer * 15
ActiveChart.Parent.Left = 5
```

Notes:

- Once a With block is entered, object can't be changed. As a result, you can't use a single With statement to affect a number of different objects
- **Don't jump into or out of With blocks**. If statements in a With block are executed, but either the With or End With statement is not executed, **a temporary variable containing a reference to the object remains in memory until you exit the procedure**
- Don't Loop inside With statements, especially if the cached object is used as an iterator
- You can nest With statements by placing one With block within another. However, because members of outer

With blocks are masked within the inner With blocks, you must provide a fully qualified object reference in an inner With block to any member of an object in an outer With block.

Nesting Example:

This example uses the With statement to execute a series of statements on a single object.
The object and its properties are generic names used for illustration purposes only.

```
With MyObject
    .Height = 100                'Same as MyObject.Height = 100.
    .Caption = "Hello World"     'Same as MyObject.Caption = "Hello World".
    With .Font
        .Color = Red             'Same as MyObject.Font.Color = Red.
        .Bold = True             'Same as MyObject.Font.Bold = True.
        MyObject.Height = 200    'Inner-most With refers to MyObject.Font (must be qualified
    End With
End With
```

More Info on MSDN

Chapter 27: VBA Security

Section 27.1: Password Protect your VBA

Sometimes you have sensitive information in your VBA (e.g., passwords) that you don't want users to have access to. You can achieve basic security on this information by password-protecting your VBA project.

Follow these steps:

1. Open your Visual Basic Editor (Alt + F11)
2. Navigate to Tools -> VBAProject Properties...
3. Navigate to the Protection tab
4. Check off the "Lock project for viewing" checkbox
5. Enter your desired password in the Password and Confirm Password textboxes

Now when someone wants to access your code within an Office application, they will first need to enter the password. Be aware, however, that even a strong VBA project password is trivial to break.

Chapter 28: Debugging and Troubleshooting

Section 28.1: Immediate Window

If you would like to test a line of macro code without needing to run an entire sub, you can type commands directly into the Immediate Window and hit `ENTER` to run the line.

For testing the output of a line, you can precede it with a question mark `?` to print directly to the Immediate Window. Alternatively, you can also use the `print` command to have the output printed.

While in the Visual Basic Editor, press `CTRL + G` to open the Immediate Window. To rename your currently selected sheet to "ExampleSheet", type the following in the Immediate Window and hit `ENTER`

```
   ActiveSheet.Name = "ExampleSheet"
```

To print the currently selected sheet's name directly in the Immediate Window

```
? ActiveSheet.Name
ExampleSheet
```

This method can be very useful to test the functionality of built in or user defined functions before implementing them in code. The example below demonstrates how the Immediate Window can be used to test the output of a function or series of functions to confirm an expected.

```
'In this example, the Immediate Window was used to confirm that a series of Left and Right
'string methods would return the desired string

'expected output: "value"
print Left(Right("1111value1111",9),5) ' <---- written code here, ENTER pressed
value                                  ' <---- output
```

The Immediate Window can also be used to set or reset Application, Workbook, or other needed properties. This can be useful if you have `Application.EnableEvents = False` in a subroutine that unexpectedly throws an error, causing it to close without resetting the value to `True` (which can cause frustrating and unexpected functionality. In that case, the commands can be typed directly into the Immediate Window and run:

```
? Application.EnableEvents       ' <---- Testing the current state of "EnableEvents"
False                            ' <---- Output
Application.EnableEvents = True  ' <---- Resetting the property value to True
? Application.EnableEvents       ' <---- Testing the current state of "EnableEvents"
True                             ' <---- Output
```

For more advanced debugging techniques, a colon `:` can be used as a line separator. This can be used for multi-line expressions such as looping in the example below.

```
x = Split("a,b,c",","): For i = LBound(x,1) to UBound(x,1): Debug.Print x(i): Next i '<----Input
this and press enter
a '<----Output
b '<----Output
c '<----Output
```

Section 28.2: Use Timer to Find Bottlenecks in Performance

The first step in optimizing for speed is finding the slowest sections of code. The `Timer` VBA function returns the number of seconds elapsed since midnight with a precision of 1/256th of a second (3.90625 milliseconds) on Windows based PCs. The VBA functions `Now` and `Time` are only accurate to a second.

```
Dim start As Double        ' Timer returns Single, but converting to Double to avoid
start = Timer              ' scientific notation like 3.90625E-03 in the Immediate window
' ... part of the code
Debug.Print Timer - start; "seconds in part 1"

start = Timer
' ... another part of the code
Debug.Print Timer - start; "seconds in part 2"
```

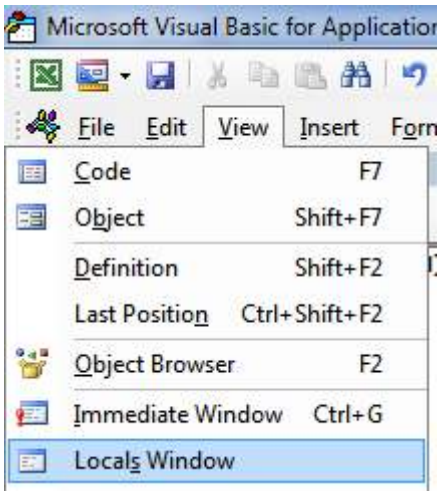
Section 28.3: Debugger Locals Window

The Locals window provides easy access to the current value of variables and objects within the scope of the function or subroutine you are running. It is an essential tool to debugging your code and stepping through changes in order to find issues. It also allows you to explore properties you might not have known existed.

Take the following example,

```
Option Explicit
Sub LocalsWindowExample()
    Dim findMeInLocals As Integer
    Dim findMEInLocals2 As Range

    findMeInLocals = 1
    Set findMEInLocals2 = ActiveWorkbook.Sheets(1).Range("A1")
End Sub
```

In the VBA Editor, click View --> Locals Window



Then by stepping through the code using F8 after clicking inside the subroutine, we have stopped before getting to assigning findMeinLocals. Below you can see the value is 0 --- and this is what would be used if you never assigned it a value. The range object is 'Nothing'.

```
Option Explicit
Sub LocalsWindowExample()
    Dim findMeInLocals As Integer
    Dim findMEInLocals2 As Range

    findMeInLocals = 1
    Set findMEInLocals2 = ActiveWorkbook.Sheets(1).Range("A1")
End Sub
```

Locals

VBAProject.Sheet1.LocalsWindowExample

Expression	Value	Type
Me		Sheet1/Sheet1
findMeInLocals	0	Integer
findMEInLocals2	Nothing	Range

If we stop right before the subroutine ends, we can see the final values of the variables.

```
Option Explicit
Sub LocalsWindowExample()
    Dim findMeInLocals As Integer
    Dim findMEInLocals2 As Range

    findMeInLocals = 1
    Set findMEInLocals2 = ActiveWorkbook.Sheets(1).Range("A1")
End Sub
```

We can see findMeInLocals with a value of 1 and type of Integer, and FindMeInLocals2 with a type of Range/Range. If we click the + sign we can expand the object and see its properties, such as count or column.

Locals

VBAProject.Sheet1.LocalsWindowExample

Expression	Value	Type
Me		Sheet1/Sheet1
findMeInLocals	1	Integer
findMEInLocals2		Range/Range
AddIndent	False	Variant/Boolean
AllowEdit	True	Boolean
Application		Application/Application
Areas		Areas/Areas
Borders		Borders/Borders
Cells		Range/Range
Column	1	Long
ColumnWidth	8.43	Variant/Double
Comment	Nothing	Comment
Count	1	Long
CountLarge	1	Variant/<Unsupported variant type>
Creator	xlCreatorCode	XlCreator
CurrentArray	<No cells were found.>	Range
CurrentRegion		Range/Range
Dependents	<No cells were found.>	Range
DirectDependents	<No cells were found.>	Range
DirectPrecedents	<No cells were found.>	Range
DisplayFormat		DisplayFormat/DisplayFormat

Section 28.4: Debug.Print

To print a listing of the Error Code descriptions to the Immediate Window, pass it to the `Debug.Print` function:

```
Private Sub ListErrCodes()
    Debug.Print "List Error Code Descriptions"
    For i = 0 To 65535
        e = Error(i)
        If e <> "Application-defined or object-defined error" Then Debug.Print i & ": " & e
    Next i
End Sub
```

You can show the Immediate Window by:

- Selecting **V**iew | **I**mmediate Window from the menu bar
- Using the keyboard shortcut **Ctrl-G**

Section 28.5: Stop

The Stop command will pause the execution when called. From there, the process can be resumed or be executed step by step.

```
Sub Test()
    Dim TestVar as String
    TestVar = "Hello World"
    Stop                    'Sub will be executed to this point and then wait for the user
    MsgBox TestVar
End Sub
```

Section 28.6: Adding a Breakpoint to your code

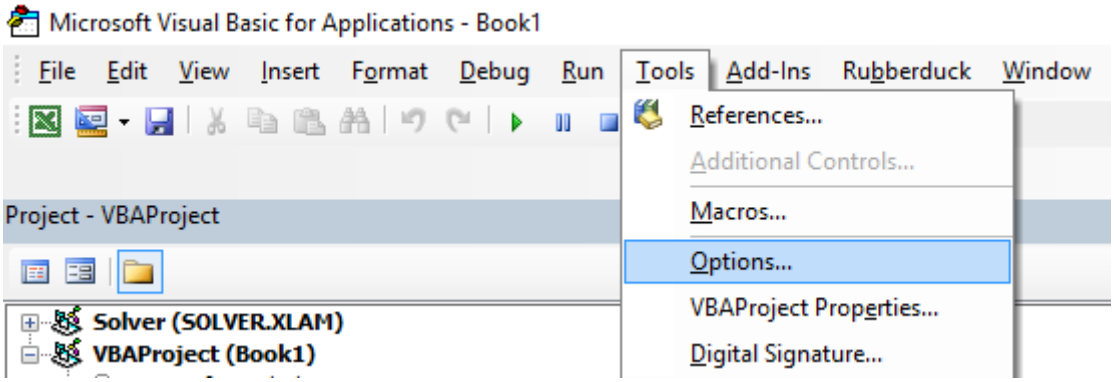
You can easily add a breakpoint to your code by clicking on the grey column to the left of the line of your VBA code where you want execution to stop. A red dot appears in the column and the breakpoint code is also highlighted in red.

You can add multiple breakpoints throughout your code and resuming execution is achieved by pressing the "play" icon in your menu bar. Not all code can be a breakpoint as variable definition lines, the first or last line of a procedure and comment lines cannot be selected as a breakpoint.

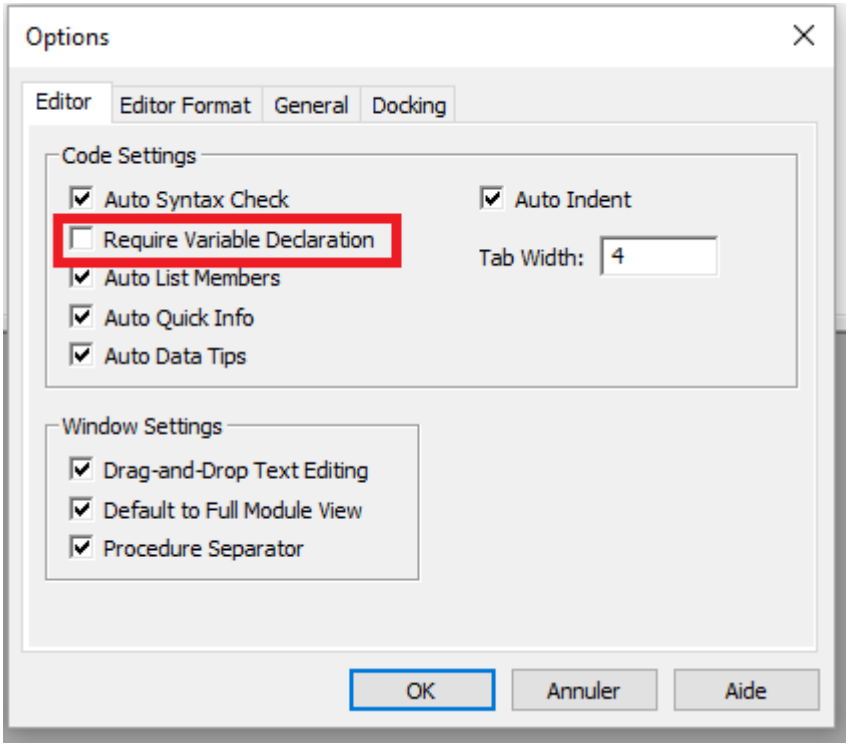
Chapter 29: VBA Best Practices

Section 29.1: ALWAYS Use "Option Explicit"

In the VBA Editor window, from the Tools menu select "Options":



Then in the "Editor" tab, make sure that "Require Variable Declaration" is checked:



Selecting this option will automatically put `Option Explicit` at the top of every VBA module.

> **Small note:** This is true for the modules, class modules, etc. that haven't been opened so far. So if you already had a look at e.g. the code of `Sheet1` before activating the option "Require Variable Declaration", `Option Explicit` will not be added!

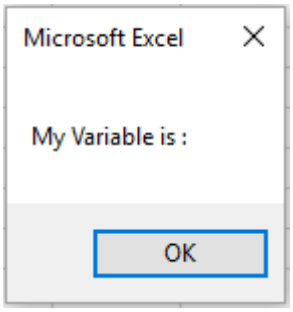
`Option Explicit` requires that every variable has to be defined before use, e.g. with a `Dim` statement. Without `Option Explicit` enabled, any unrecognized word will be assumed by the VBA compiler to be a new variable of the `Variant` type, causing extremely difficult-to-spot bugs related to typographical errors. With `Option Explicit` enabled, any unrecognized words will cause a compile error to be thrown, indicating the offending line.

Example :

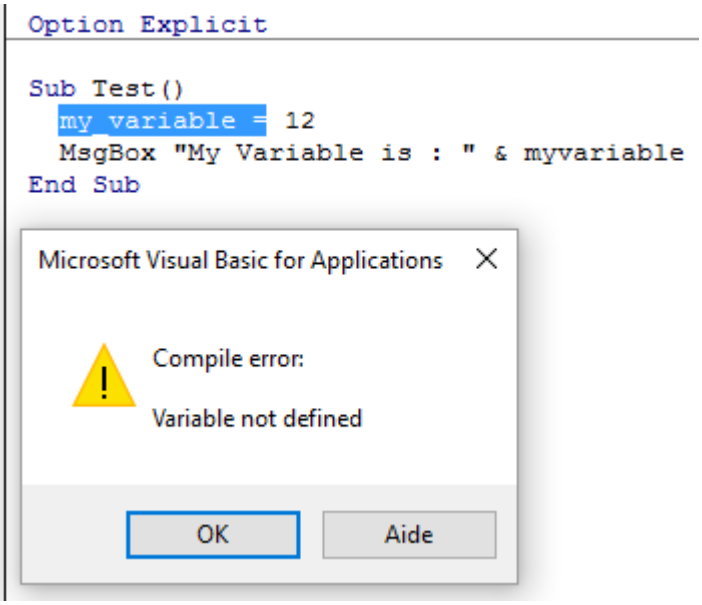
If you run the following code :

```
Sub Test()
  my_variable = 12
  MsgBox "My Variable is : " & myvariable
End Sub
```

You will get the following message :



You have made an error by writing `myvariable` instead of `my_variable`, then the message box displays an empty variable. If you use `Option Explicit` , this error is not possible because you will get a compile error message indicating the problem.

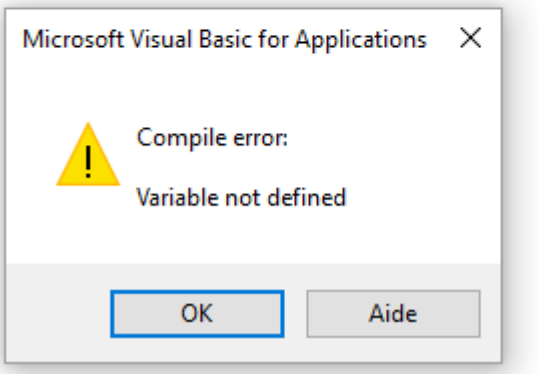


Now if you add the correct declaration :

```
Sub Test()
  Dim my_variable As Integer
  my_variable = 12
  MsgBox "My Variable is : " & myvariable
End Sub
```

You will obtain an error message indicating precisely the error with `myvariable` :

```
Option Explicit

Sub Test()
  Dim my_variable As Integer
  my_variable = 12
  MsgBox "My Variable is : " & myvariable
End Sub
```



> **Note on Option Explicit and Arrays** (Declaring a Dynamic Array):
>
> You can use the ReDim statement to declare an array implicitly within a procedure.
>
> - Be careful not to misspell the name of the array when you use the ReDim statement
> - Even if the Option Explicit statement is included in the module, a new array will be created
>
> ```
> Dim arr() as Long
>
> ReDim ar() 'creates new array "ar" - "ReDim ar()" acts like "Dim ar()"
> ```

Section 29.2: Work with Arrays, Not With Ranges

Office Blog - Excel VBA Performance Coding Best Practices

Often, best performance is achieved by avoiding the use of `Range` as much as possible. In this example we read in an entire `Range` object into an array, square each number in the array, and then return the array back to the `Range`. This accesses `Range` only twice, whereas a loop would access it 20 times for the read/writes.

```
Option Explicit
Sub WorkWithArrayExample()

Dim DataRange As Variant
Dim Irow As Long
Dim Icol As Integer
DataRange = ActiveSheet.Range("A1:A10").Value ' read all the values at once from the Excel grid, put
into an array

For Irow = LBound(DataRange,1) To UBound(DataRange, 1) ' Get the number of rows.
  For Icol = LBound(DataRange,2) To UBound(DataRange, 2) ' Get the number of columns.
    DataRange(Irow, Icol) = DataRange(Irow, Icol) * DataRange(Irow, Icol) ' cell.value^2
  Next Icol
Next Irow
```

```
    ActiveSheet.Range("A1:A10").Value = DataRange ' writes all the results back to the range at once

End Sub
```

More tips and info with timed examples can be found in Charles Williams's Writing efficient VBA UDFs (Part 1) and other articles in the series.

Section 29.3: Switch off properties during macro execution

It is best practice in any programming language to **avoid premature optimization.** However, if testing reveals that your code is running too slowly, you may gain some speed by switching off some of the application's properties while it runs. Add this code to a standard module:

```
Public Sub SpeedUp( _
    SpeedUpOn As Boolean, _
    Optional xlCalc as XlCalculation = xlCalculationAutomatic _
)
    With Application
        If SpeedUpOn Then
            .ScreenUpdating = False
            .Calculation = xlCalculationManual
            .EnableEvents = False
            .DisplayStatusBar = False 'in case you are not showing any messages
            ActiveSheet.DisplayPageBreaks = False 'note this is a sheet-level setting
        Else
            .ScreenUpdating = True
            .Calculation = xlCalc
            .EnableEvents = True
            .DisplayStatusBar = True
            ActiveSheet.DisplayPageBreaks = True
        End If
    End With
End Sub
```

More info on Office Blog - Excel VBA Performance Coding Best Practices

And just call it at beginning and end of macros:

```
Public Sub SomeMacro
    'store the initial "calculation" state
    Dim xlCalc As XlCalculation
    xlCalc = Application.Calculation

    SpeedUp True

    'code here ...

    'by giving the second argument the initial "calculation" state is restored
    'otherwise it is set to 'xlCalculationAutomatic'
    SpeedUp False, xlCalc
End Sub
```

While these can largely be considered "enhancements" for regular `Public Sub` procedures, disabling event handling with `Application.EnableEvents = False` should be considered mandatory for `Worksheet_Change` and `Workbook_SheetChange` private event macros that change values on one or more worksheets. Failure to disable event triggers will cause the event macro to recursively run on top of itself when a value changes and can lead to a "frozen" workbook. Remember to turn events back on before leaving the event macro, possibly through a "safe exit" error handler.

```
Option Explicit

Private Sub Worksheet_Change(ByVal Target As Range)
    If Not Intersect(Target, Range("A:A")) Is Nothing Then
        On Error GoTo bm_Safe_Exit
        Application.EnableEvents = False

        'code that may change a value on the worksheet goes here

    End If
bm_Safe_Exit:
    Application.EnableEvents = True
End Sub
```

One caveat: While disabling these settings will improve run time, they may make debugging your application much more difficult. If your code is *not* functioning correctly, comment out the `SpeedUp True` call until you figure out the problem.

This is particularly important if you are writing to cells in a worksheet and then reading back in calculated results from worksheet functions since the `xlCalculationManual` prevents the workbook from calculating. To get around this without disabling `SpeedUp`, you may want to include `Application.Calculate` to run a calculation at specific points.

NOTE: Since these are properties of the `Application` itself, you need to ensure that they are enabled again before your macro exits. This makes it particularly important to use error handlers and to avoid multiple exit points (i.e. `End` or `Unload Me`).

With error handling:

```
Public Sub SomeMacro()
    'store the initial "calculation" state
    Dim xlCalc As XlCalculation
    xlCalc = Application.Calculation

    On Error GoTo Handler
    SpeedUp True

    'code here ...
    i = 1 / 0
CleanExit:
    SpeedUp False, xlCalc
    Exit Sub
Handler:
    'handle error
    Resume CleanExit
End Sub
```

Section 29.4: Use VB constants when available

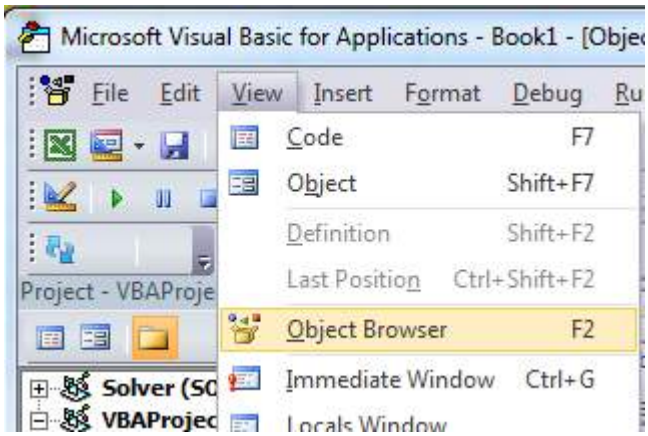
```
If MsgBox("Click OK") = vbOK Then
```

can be used in place of

```
If MsgBox("Click OK") = 1 Then
```

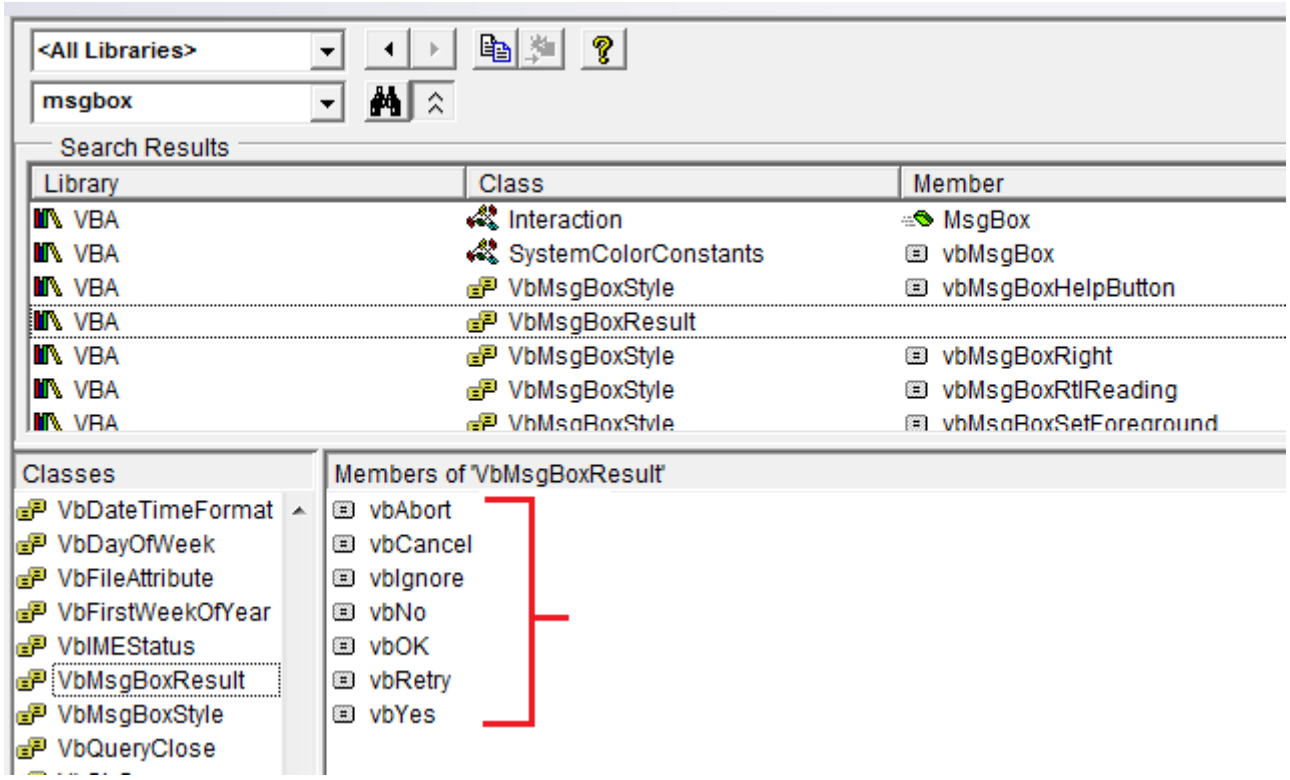
in order to improve readability.

Use *Object Browser* to find available VB constants. *View* → *Object Browser* or F2 from VB Editor.



Enter class to search

View members available



Section 29.5: Avoid using SELECT or ACTIVATE

It is **very** rare that you'll ever want to use `SELECT` or `Activate` in your code, but some Excel methods do require a worksheet or workbook to be activated before they'll work as expected.

If you're just starting to learn VBA, you'll often be suggested to record your actions using the macro recorder, then go look at the code. For example, I recorded actions taken to enter a value in cell D3 on Sheet2, and the macro code looks like this:

```
Option Explicit
Sub Macro1()
```

```
'
' Macro1 Macro
'

'
    Sheets("Sheet2").Select
    Range("D3").Select
    ActiveCell.FormulaR1C1 = "3.1415"    '(see **note below)
    Range("D4").Select
End Sub
```

Remember though, the macro recorder creates a line of code for EACH of your (user) actions. This includes clicking on the worksheet tab to select Sheet2 (`Sheets("Sheet2").Select`), clicking on cell D3 before entering the value (`Range("D3").Select`), and using the Enter key (which is effectively "selecting" the cell below the currently selected cell: `Range("D4").Select`).

There are multiple issues with using `.Select` here:

- **The worksheet is not always specified.** This happens if you don't switch worksheets while recording, and means that the code will yield different results for different active worksheets.
- **`.Select()` is slow.** Even if `Application.ScreenUpdating` is set to `False`, this is an unneccessary operation to be processed.
- **`.Select()` is unruly.** If `Application.ScreenUpdating` is left to `True`, Excel will actually select the cells, the worksheet, the form... whatever it is you're working with. This is stressful to the eyes and really unpleasant to watch.
- **`.Select()` will trigger listeners.** This is a bit advanced already, but unless worked around, functions like `Worksheet_SelectionChange()` will be triggered.

When you're coding in VBA, all of the "typing" actions (i.e. `SELECT` statements) are no longer necessary. Your code may be reduced to a single statement to put the value in the cell:

```
'--- GOOD
ActiveWorkbook.Sheets("Sheet2").Range("D3").Value = 3.1415

'--- BETTER
Dim myWB      As Workbook
Dim myWS      As Worksheet
Dim myCell    As Range

Set myWB = ThisWorkbook              '*** see NOTE2
Set myWS = myWB.Sheets("Sheet2")
Set myCell = myWS.Range("D3")

myCell.Value = 3.1415
```

(The BETTER example above shows using intermediate variables to separate different parts of the cell reference. The GOOD example will always work just fine, but can be very cumbersome in much longer code modules and more difficult to debug if one of the references is mistyped.)

**NOTE: the macro recorder makes many assumptions about the type of data you're entering, in this case entering a string value as a formula to create the value. Your code doesn't have to do this and can simply assign a numerical value directly to the cell as shown above.

**NOTE2: the recommended practice is to set your local workbook variable to `ThisWorkbook` instead of `ActiveWorkbook` (unless you explicitly need it). The reason is your macro will generally need/use resources in whatever workbook the VBA code originates and will NOT look outside of that workbook -- again, unless you

explicitly direct your code to work with another workbook. When you have multiple workbooks open in Excel, the `ActiveWorkbook` is the one with the focus *which may be different from the workbook being viewed in your VBA Editor*. So you think you're executing in a one workbook when you're really referencing another. `ThisWorkbook` refers to the workbook containing the code being executed.

Section 29.6: Always define and set references to all Workbooks and Sheets

When working with multiple open Workbooks, each of which may have multiple Sheets, it's safest to define and set reference to all Workbooks and Sheets.

Don't rely on `ActiveWorkbook` or `ActiveSheet` as they might be changed by the user.

The following code example demonstrates how to copy a range from "*Raw_Data*" sheet in the "*Data.xlsx*" workbook to "*Refined_Data*" sheet in the "*Results.xlsx*" workbook.

The procedure also demonstrates how to copy and paste without using the **`SELECT`** method.

```
Option Explicit

Sub CopyRanges_BetweenShts()


    Dim wbSrc                           As Workbook
    Dim wbDest                          As Workbook
    Dim shtCopy                         As Worksheet
    Dim shtPaste                        As Worksheet

    ' set reference to all workbooks by name, don't rely on ActiveWorkbook
    Set wbSrc = Workbooks("Data.xlsx")
    Set wbDest = Workbooks("Results.xlsx")

    ' set reference to all sheets by name, don't rely on ActiveSheet
    Set shtCopy = wbSrc.Sheet1 '// "Raw_Data" sheet
    Set shtPaste = wbDest.Sheet2 '// "Refined_Data") sheet

    ' copy range from "Data" workbook to "Results" workbook without using Select
    shtCopy.Range("A1:C10").Copy _
    Destination:=shtPaste.Range("A1")

End Sub
```

Section 29.7: Use descriptive variable naming

Descriptive names and structure in your code help make comments unnecessary

```
Dim ductWidth  As Double
Dim ductHeight As Double
Dim ductArea   As Double

ductArea = ductWidth * ductHeight
```

is better than

```
Dim a, w, h

a = w * h
```

This is especially helpful when you are copying data from one place to another, whether it's a cell, range, worksheet, or workbook. Help yourself by using names such as these:

```
Dim myWB As Workbook
Dim srcWS As Worksheet
Dim destWS As Worksheet
Dim srcData As Range
Dim destData As Range

Set myWB = ActiveWorkbook
Set srcWS = myWB.Sheets("Sheet1")
Set destWS = myWB.Sheets("Sheet2")
Set srcData = srcWS.Range("A1:A10")
Set destData = destWS.Range("B11:B20")
destData = srcData
```

If you declare multiple variables in one line make sure to specify a type for *every* variable like:

```
Dim ductWidth As Double, ductHeight As Double, ductArea As Double
```

The following will only declare the last variable and the first ones will remain `Variant`:

```
Dim ductWidth, ductHeight, ductArea As Double
```

Section 29.8: Document Your Work

It's good practice to document your work for later use, especially if you are coding for a dynamic workload. Good comments should explain why the code is doing something, not what the code is doing.

```
Function Bonus(EmployeeTitle as String) as Double
    If EmployeeTitle = "Sales" Then
        Bonus = 0    'Sales representatives receive commission instead of a bonus
    Else
        Bonus = .10
    End If
End Function
```

If your code is so obscure that it requires comments to explain what it is doing, consider rewriting it to be more clear instead of explaining it through comments. For example, instead of:

```
Sub CopySalesNumbers
    Dim IncludeWeekends as Boolean

    'Boolean values can be evaluated as an integer, -1 for True, 0 for False.
    'This is used here to adjust the range from 5 to 7 rows if including weekends.
    Range("A1:A" & 5 - (IncludeWeekends * 2)).Copy
    Range("B1").PasteSpecial
End Sub
```

Clarify the code to be easier to follow, such as:

```
Sub CopySalesNumbers
    Dim IncludeWeekends as Boolean
    Dim DaysinWeek as Integer

    If IncludeWeekends Then
        DaysinWeek = 7
```

```
    Else
        DaysinWeek = 5
    End If
    Range("A1:A" & DaysinWeek).Copy
    Range("B1").PasteSpecial
End Sub
```

Section 29.9: Error Handling

Good error handling prevents end users from seeing VBA runtime errors and helps the developer easily diagnose and correct errors.

There are three main methods of Error Handling in VBA, two of which should be avoided for distributed programs unless specifically required in the code.

```
On Error GoTo 0 'Avoid using
```

or

```
On Error Resume Next 'Avoid using
```

Prefer using:

```
On Error GoTo <line> 'Prefer using
```

On Error GoTo 0

If no error handling is set in your code, `On Error GoTo 0` is the default error handler. In this mode, any runtime errors will launch the typical VBA error message, allowing you to either end the code or enter `debug` mode, identifying the source. While writing code, this method is the simplest and most useful, but it should always be avoided for code that is distributed to end users, as this method is very unsightly and difficult for end users to understand.

On Error Resume Next

`On Error Resume Next` will cause VBA to ignore any errors that are thrown at runtime for all lines following the error call until the error handler has been changed. In very specific instances, this line can be useful, but it should be avoided outside of these cases. For example, when launching a separate program from an Excel Macro, the `On Error Resume Next` call can be useful if you are unsure whether or not the program is already open:

```
'In this example, we open an instance of Powerpoint using the On Error Resume Next call
Dim PPApp As PowerPoint.Application
Dim PPPres As PowerPoint.Presentation
Dim PPSlide As PowerPoint.Slide

'Open PPT if not running, otherwise select active instance
On Error Resume Next
Set PPApp = GetObject(, "PowerPoint.Application")
On Error GoTo ErrHandler
If PPApp Is Nothing Then
    'Open PowerPoint
    Set PPApp = CreateObject("PowerPoint.Application")
    PPApp.Visible = True
End If
```

Had we not used the `On Error Resume Next` call and the Powerpoint application was not already open, the `GetObject` method would throw an error. Thus, `On Error Resume Next` was necessary to avoid creating two instances of the application.

Note: It is also a best practice to *immediately* reset the error handler as soon as you no longer need the `On Error Resume Next` call

On Error GoTo <line>

This method of error handling is recommended for all code that is distributed to other users. This allows the programmer to control exactly how VBA handles an error by sending the code to the specified line. The tag can be filled with any string (including numeric strings), and will send the code to the corresponding string that is followed by a colon. Multiple error handling blocks can be used by making different calls of `On Error GoTo <line>`. The subroutine below demonstrates the syntax of an `On Error GoTo <line>` call.

Note: It is essential that the `Exit Sub` line is placed above the first error handler and before every subsequent error handler to prevent the code from naturally progressing into the block *without* an error being called. Thus, it is best practice for function and readability to place error handlers at the end of a code block.

```
Sub YourMethodName()
    On Error GoTo errorHandler
    ' Insert code here
    On Error GoTo secondErrorHandler

    Exit Sub 'The exit sub line is essential, as the code will otherwise
             'continue running into the error handling block, likely causing an error

errorHandler:
    MsgBox "Error " & Err.Number & ": " & Err.Description & " in " & _
        VBE.ActiveCodePane.CodeModule, vbOKOnly, "Error"
    Exit Sub

secondErrorHandler:
    If Err.Number = 424 Then 'Object not found error (purely for illustration)
        Application.ScreenUpdating = True
        Application.EnableEvents = True
        Exit Sub
    Else
        MsgBox "Error " & Err.Number & ": " & Err.Desctription
        Application.ScreenUpdating = True
        Application.EnableEvents = True
        Exit Sub
    End If
    Exit Sub

End Sub
```

If you exit your method with your error handling code, ensure that you clean up:

- Undo anything that is partially completed
- Close files
- Reset screen updating
- Reset calculation mode
- Reset events
- Reset mouse pointer
- Call unload method on instances of objects, that persist after the `End Sub`

- Reset status bar

Section 29.10: Never Assume The Worksheet

Even when all your work is directed at a single worksheet, it's still a very good practice to explicitly specify the worksheet in your code. This habit makes it much easier to expand your code later, or to lift parts (or all) of a `Sub` or `Function` to be re-used someplace else. Many developers establish a habit of (re)using the same local variable name for a worksheet in their code, making re-use of that code even more straightforward.

As an example, the following code is ambiguous -- but works! -- as long the developer doesn't activate or change to a different worksheet:

```
Option Explicit
Sub ShowTheTime()
    '--- displays the current time and date in cell A1 on the worksheet
    Cells(1, 1).Value = Now()   ' don't refer to Cells without a sheet reference!
End Sub
```

If `Sheet1` is active, then cell A1 on Sheet1 will be filled with the current date and time. But if the user changes worksheets for any reason, then the code will update whatever the worksheet is currently active. The destination worksheet is ambiguous.

The best practice is to always identify which worksheet to which your code refers:

```
Option Explicit
Sub ShowTheTime()
    '--- displays the current time and date in cell A1 on the worksheet
    Dim myWB As Workbook
    Set myWB = ThisWorkbook
    Dim timestampSH As Worksheet
    Set timestampSH = myWB.Sheets("Sheet1")
    timestampSH.Cells(1, 1).Value = Now()
End Sub
```

The code above is clear in identifying both the workbook and the worksheet. While it may seem like overkill, creating a good habit concerning target references will save you from future problems.

Section 29.11: Avoid re-purposing the names of Properties or Methods as your variables

It is generally not considered 'best practice' to re-purpose the reserved names of Properties or Methods as the name(s) of your own procedures and variables.

Bad Form - While the following is (strictly speaking) legal, working code the re-purposing of the Find method as well as the Row, Column and Address properties can cause problems/conflicts with name ambiguity and is just plain confusing in general.

```
Option Explicit

Sub find()
    Dim row As Long, column As Long
    Dim find As String, address As Range

    find = "something"

    With ThisWorkbook.Worksheets("Sheet1").Cells
```

```
        Set address = .SpecialCells(xlCellTypeLastCell)
        row = .find(what:=find, after:=address).row        '< note .row not capitalized
        column = .find(what:=find, after:=address).column  '< note .column not capitalized

        Debug.Print "The first 'something' is in " & .Cells(row, column).address(0, 0)
    End With
End Sub
```

Good Form - With all of the reserved words renamed into close but unique approximations of the originals, any potential naming conflicts have been avoided.

```
Option Explicit

Sub myFind()
    Dim rw As Long, col As Long
    Dim wht As String, lastCell As Range

    wht = "something"

    With ThisWorkbook.Worksheets("Sheet1").Cells
        Set lastCell = .SpecialCells(xlCellTypeLastCell)
        rw = .Find(What:=wht, After:=lastCell).Row         '◂ note .Find and .Row
        col = .Find(What:=wht, After:=lastCell).Column     '◂ .Find and .Column

        Debug.Print "The first 'something' is in " & .Cells(rw, col).Address(0, 0)
    End With
End Sub
```

While there may come a time when you want to intentionally rewrite a standard method or property to your own specifications, those situations are few and far between. For the most part, stay away from reusing reserved names for your own constructs.

Section 29.12: Avoid using ActiveCell or ActiveSheet in Excel

Using `ActiveCell` or `ActiveSheet` can be source of mistakes if (for any reason) the code is executed in the wrong place.

```
ActiveCell.Value = "Hello"
'will place "Hello" in the cell that is currently selected
Cells(1, 1).Value = "Hello"
'will always place "Hello" in A1 of the currently selected sheet

ActiveSheet.Cells(1, 1).Value = "Hello"
'will place "Hello" in A1 of the currently selected sheet
Sheets("MySheetName").Cells(1, 1).Value = "Hello"
'will always place "Hello" in A1 of the sheet named "MySheetName"
```

- The use of `Active*` can create problems in long running macros if your user gets bored and clicks on another worksheet or opens another workbook.
- It can create problems if your code opens or creates another workbook.
- It can create problems if your code uses `Sheets("MyOtherSheet").Select` and you've forgotten which sheet you were on before you start reading from or writing to it.

Section 29.13: WorksheetFunction object executes faster than a UDF equivalent

VBA is compiled in run-time, which has a huge negative impact on it's performance, everything built-in will be faster, try to use them.

As an example I'm comparing SUM and COUNTIF functions, but you can use if for anything you can solve with WorkSheetFunctions.

A first attempt for those would be to loop through the range and process it cell by cell (using a range):

```
Sub UseRange()
    Dim rng as Range
    Dim Total As Double
    Dim CountLessThan01 As Long

    Total = 0
    CountLessThan01 = 0
    For Each rng in Sheets(1).Range("A1:A100")
        Total = Total + rng.Value2
        If rng.Value < 0.1 Then
            CountLessThan01 = CountLessThan01 + 1
        End If
    Next rng
    Debug.Print Total & ", " & CountLessThan01
End Sub
```

One improvement can be to store the range values in an array and process that:

```
Sub UseArray()
    Dim DataToSummarize As Variant
    Dim i As Long
    Dim Total As Double
    Dim CountLessThan01 As Long

    DataToSummarize = Sheets(1).Range("A1:A100").Value2 'faster than .Value
    Total = 0
    CountLessThan01 = 0
    For i = 1 To 100
        Total = Total + DataToSummarize(i, 1)
        If DataToSummarize(i, 1) < 0.1 Then
            CountLessThan01 = CountLessThan01 + 1
        End If
    Next i
    Debug.Print Total & ", " & CountLessThan01
End Sub
```

But instead of writing any loop you can use `Application.Worksheetfunction` which is very handy for executing simple formulas:

```
Sub UseWorksheetFunction()
    Dim Total As Double
    Dim CountLessThan01 As Long

    With Application.WorksheetFunction
        Total = .Sum(Sheets(1).Range("A1:A100"))
        CountLessThan01 = .CountIf(Sheets(1).Range("A1:A100"), "<0.1")
    End With
```

```
    Debug.Print Total & ", " & CountLessThan01
End Sub
```

Or, for more complex calculations you can even use `Application.Evaluate`:

```
Sub UseEvaluate()
    Dim Total As Double
    Dim CountLessThan01 As Long

    With Application
        Total = .Evaluate("SUM(" & Sheet1.Range("A1:A100").Address( _
            external:=True) & ")")
        CountLessThan01 = .Evaluate("COUNTIF('Sheet1'!A1:A100,""<0.1"")")
    End With

    Debug.Print Total & ", " & CountLessThan01
End Sub
```

And finally, running above Subs 25,000 times each, here is the average (5 tests) time in milliseconds (of course it'll be different on each pc, but compared to each other they'll behave similarly):

1. UseWorksheetFunction: 2156 ms
2. UseArray: 2219 ms (+ 3 %)
3. UseEvaluate: 4693 ms (+ 118 %)
4. UseRange: 6530 ms (+ 203 %)

Chapter 30: Excel VBA Tips and Tricks

Section 30.1: Using xlVeryHidden Sheets

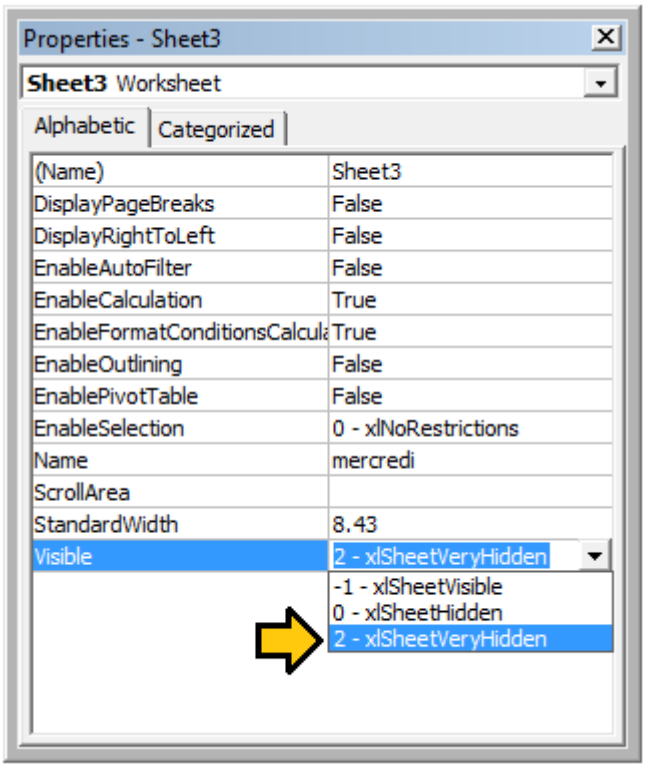
Worksheets in excel have three options for the `Visible` property. These options are represented by constants in the `xlSheetVisibility` enumeration and are as follows:

1. `xlVisible` or `xlSheetVisible` value: `-1` (the default for new sheets)
2. `xlHidden` or `xlSheetHidden` value: `0`
3. `xlVeryHidden` `xlSheetVeryHidden` value: 2

Visible sheets represent the default visibility for sheets. They are visible in the sheet tab bar and can be freely selected and viewed. Hidden sheets are hidden from the sheet tab bar and are thus not selectable. However, hidden sheets can be unhidden from the excel window by right clicking on the sheet tabs and selecting "Unhide"

Very Hidden sheets, on the other hand, are *only* accessible through the Visual Basic Editor. This makes them an incredibly useful tool for storing data across instances of excel as well as storing data that should be hidden from end users. The sheets can be accessed by named reference within VBA code, allowing easy use of the stored data.

To manually change a worksheet's .Visible property to xlSheetVeryHidden, open the VBE's Properties window (F4), select the worksheet you want to change and use the drop-down in the thirteenth row to make your selection.



To change a worksheet's .Visible property to xlSheetVeryHidden[1] in code, similarly access the .Visible property and assign a new value.

```
with Sheet3
    .Visible = xlSheetVeryHidden
end with
```

[1] Both **xlVeryHidden** and **xlSheetVeryHidden** return a numerical value of **2** (they are interchangeable).

Section 30.2: Using Strings with Delimiters in Place of Dynamic Arrays

Using Dynamic Arrays in VBA can be quite clunky and time intensive over very large data sets. When storing simple data types in a dynamic array (Strings, Numbers, Booleans etc.), one can avoid the `ReDim Preserve` statements required of dynamic arrays in VBA by using the `Split()` function with some clever string procedures. For example, we will look at a loop that adds a series of values from a range to a string based on some conditions, then uses that string to populate the values of a ListBox.

```
Private Sub UserForm_Initialize()

Dim Count As Long, DataString As String, Delimiter As String

For Count = 1 To ActiveSheet.UsedRows.Count
    If ActiveSheet.Range("A" & Count).Value <> "Your Condition" Then
        RowString = RowString & Delimiter & ActiveSheet.Range("A" & Count).Value
        Delimiter = "><" 'By setting the delimiter here in the loop, you prevent an extra occurance
of the delimiter within the string
    End If
Next Count

ListBox1.List = Split(DataString, Delimiter)

End Sub
```

The `Delimiter` string itself can be set to any value, but it is prudent to choose a value which will not naturally occur within the set. Say, for example, you were processing a column of dates. In that case, using `.`, `-`, or `/` would be unwise as delimiters, as the dates could be formatted to use any one of these, generating more data points than you anticipated.

Note: There are limitations to using this method (namely the maximum length of strings), so it should be used with caution in cases of very large datasets. This is not necessarily the fastest or most effective method for creating dynamic arrays in VBA, but it is a viable alternative.

Section 30.3: Worksheet .Name, .Index or .CodeName

We know that 'best practise' dictates that a range object should have its parent worksheet explicitly referenced. A worksheet can be referred to by its .Name property, numerical .Index property or its .CodeName property but a user can reorder the worksheet queue by simply dragging a name tab or rename the worksheet with a double-click on the same tab and some typing in an unprotected workbook.

Consider a standard three worksheet. You have renamed the three worksheets Monday, Tuesday and Wednesday in that order and coded VBA sub procedures that reference these. Now consider that one user comes along and decides that Monday belongs at the end of the worksheet queue then another comes along and decides that the worksheet names look better in French. You now have a workbook with a worksheet name tab queue that looks something like the following.

If you had used either of the following worksheet reference methods, your code would now be broken.

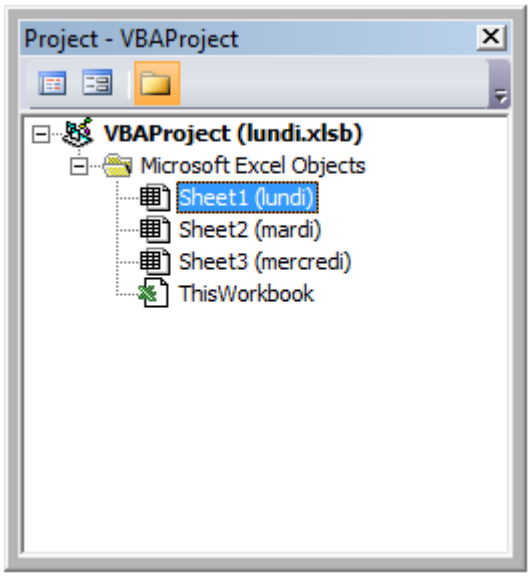
```
'reference worksheet by .Name
with worksheets("Monday")
```

```
    'operation code here; for example:
    .Range(.Cells(2, "A"), .Cells(.Rows.Count, "A").End(xlUp)) = 1
end with

'reference worksheet by ordinal .Index
with worksheets(1)
    'operation code here; for example:
    .Range(.Cells(2, "A"), .Cells(.Rows.Count, "A").End(xlUp)) = 1
end with
```

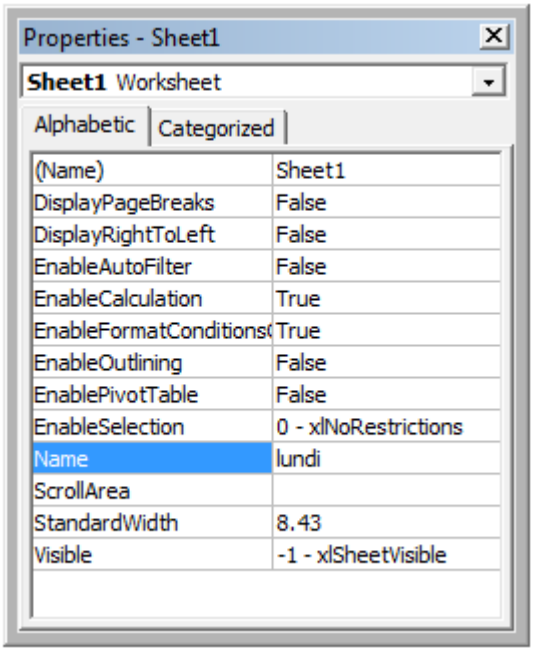
Both the original order and the original worksheet name have been compromised. However, if you had used the worksheet's .CodeName property, your sub procedure would still be operational

```
with Sheet1
    'operation code here; for example:
    .Range(.Cells(2, "A"), .Cells(.Rows.Count, "A").End(xlUp)) = 1
end with
```

The following image shows the VBA Project window ([Ctrl]+R) which lists the worksheets by .CodeName then by .Name (in brackets). The order they are displayed does not change; the ordinal .Index is taken by the order they are displayed in the name tab queue in the worksheet window.



While it is uncommon to rename a .CodeName, it is not impossible. Simply open the VBE's Properties window ([F4]).



The worksheet .CodeName is in the first row. The worksheet's .Name is in the tenth. Both are editable.

Section 30.4: Double Click Event for Excel Shapes

By default, Shapes in Excel do not have a specific way to handle single vs. double clicks, containing only the "OnAction" property to allow you to handle clicks. However, there may be instances where your code requires you to act differently (or exclusively) on a double click. The following subroutine can be added into your VBA project and, when set as the `OnAction` routine for your shape, allow you to act on double clicks.

```
Public Const DOUBLECLICK_WAIT as Double = 0.25 'Modify to adjust click delay
Public LastClickObj As String, LastClickTime As Date

Sub ShapeDoubleClick()

    If LastClickObj = "" Then
        LastClickObj = Application.Caller
        LastClickTime = CDbl(Timer)
    Else
        If CDbl(Timer) - LastClickTime > DOUBLECLICK_WAIT Then
            LastClickObj = Application.Caller
            LastClickTime = CDbl(Timer)
        Else
            If LastClickObj = Application.Caller Then
                'Your desired Double Click code here
                LastClickObj = ""
            Else
                LastClickObj = Application.Caller
                LastClickTime = CDbl(Timer)
            End If
        End If
    End If

End Sub
```

This routine will cause the shape to functionally ignore the first click, only running your desired code on the second click within the specified time span.

Section 30.5: Open File Dialog - Multiple Files

This subroutine is a quick example on how to allow a user to select multiple files and then do something with those file paths, such as get the file names and send it to the console via debug.print.

```
Option Explicit

Sub OpenMultipleFiles()
    Dim fd As FileDialog
    Dim fileChosen As Integer
    Dim i As Integer
    Dim basename As String
    Dim fso As Variant
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set fd = Application.FileDialog(msoFileDialogFilePicker)
    basename = fso.getBaseName(ActiveWorkbook.Name)
    fd.InitialFileName = ActiveWorkbook.Path ' Set Default Location to the Active Workbook Path
    fd.InitialView = msoFileDialogViewList
    fd.AllowMultiSelect = True

    fileChosen = fd.Show
    If fileChosen = -1 Then
        'open each of the files chosen
        For i = 1 To fd.SelectedItems.Count
            Debug.Print (fd.SelectedItems(i))
            Dim fileName As String
            ' do something with the files.
            fileName = fso.getFileName(fd.SelectedItems(i))
            Debug.Print (fileName)
        Next i
    End If

End Sub
```

Chapter 31: Common Mistakes

Section 31.1: Qualifying References

When referring to a `worksheet`, a `range` or individual `cells`, it is important to fully qualify the reference.

For example:

```
ThisWorkbook.Worksheets("Sheet1").Range(Cells(1, 2), Cells(2, 3)).Copy
```

Is not fully qualified: The `Cells` references do not have a workbook and worksheet associated with them. Without an explicit reference, Cells refers to the `ActiveSheet` by default. So this code will fail (produce incorrect results) if a worksheet other than `Sheet1` is the current `ActiveSheet`.

The easiest way to correct this is to use a **`With`** statement as follows:

```
With ThisWorkbook.Worksheets("Sheet1")
    .Range(.Cells(1, 2), .Cells(2, 3)).Copy
End With
```

Alternatively, you can use a Worksheet variable. (This will most likely be preferred method if your code needs to reference multiple Worksheets, like copying data from one sheet to another.)

```
Dim ws1 As Worksheet
Set ws1 = ThisWorkbook.Worksheets("Sheet1")
ws1.Range(ws1.Cells(1, 2), ws1.Cells(2, 3)).Copy
```

Another frequent problem is referencing the Worksheets collection without qualifying the Workbook. For example:

```
Worksheets("Sheet1").Copy
```

The worksheet `Sheet1` is not fully qualified, and lacks a workbook. This could fail if multiple workbooks are referenced in the code. Instead, use one of the following:

```
ThisWorkbook.Worksheets("Sheet1")       '<--ThisWorkbook refers to the workbook containing
                                        'the running VBA code
Workbooks("Book1").Worksheets("Sheet1") '<--Where Book1 is the workbook containing Sheet1
```

However, avoid using the following:

```
ActiveWorkbook.Worksheets("Sheet1")     '<--Valid, but if another workbook is activated
                                        'the reference will be changed
```

Similarly for `range` objects, if not explicitly qualified, the `range` will refer to the currently active sheet:

```
Range("a1")
```

Is the same as:

```
ActiveSheet.Range("a1")
```

Section 31.2: Deleting rows or columns in a loop

If you want to delete rows (or columns) in a loop, you should always loop starting from the end of range and move back in every step. In case of using the code:

```
Dim i As Long
With Workbooks("Book1").Worksheets("Sheet1")
    For i = 1 To 4
        If IsEmpty(.Cells(i, 1)) Then .Rows(i).Delete
    Next i
End With
```

You will miss some rows. For example, if the code deletes row 3, then row 4 becomes row 3. However, variable `i` will change to 4. So, in this case the code will miss one row and check another, which wasn't in range previously.

The right code would be

```
Dim i As Long
With Workbooks("Book1").Worksheets("Sheet1")
    For i = 4 To 1 Step -1
        If IsEmpty(.Cells(i, 1)) Then .Rows(i).Delete
    Next i
End With
```

Section 31.3: ActiveWorkbook vs. ThisWorkbook

`ActiveWorkbook` and `ThisWorkbook` sometimes get used interchangeably by new users of VBA without fully understanding which each object relates to, this can cause undesired behaviour at run-time. Both of these objects belong to the Application Object

The `ActiveWorkbook` object refers to the workbook that is currently in the top-most view of the Excel application object at the time of execution. *(e.g. The workbook that you can see and interact with at the point when this object is referenced)*

```
Sub ActiveWorkbookExample()

'// Let's assume that 'Other Workbook.xlsx' has "Bar" written in A1.

    ActiveWorkbook.ActiveSheet.Range("A1").Value = "Foo"
    Debug.Print ActiveWorkbook.ActiveSheet.Range("A1").Value '// Prints "Foo"

    Workbooks.Open("C:\Users\BloggsJ\Other Workbook.xlsx")
    Debug.Print ActiveWorkbook.ActiveSheet.Range("A1").Value '// Prints "Bar"

    Workbooks.Add 1
    Debug.Print ActiveWorkbook.ActiveSheet.Range("A1").Value '// Prints nothing

End Sub
```

The `ThisWorkbook` object refers to the workbook in which the code belongs to at the time it is being executed.

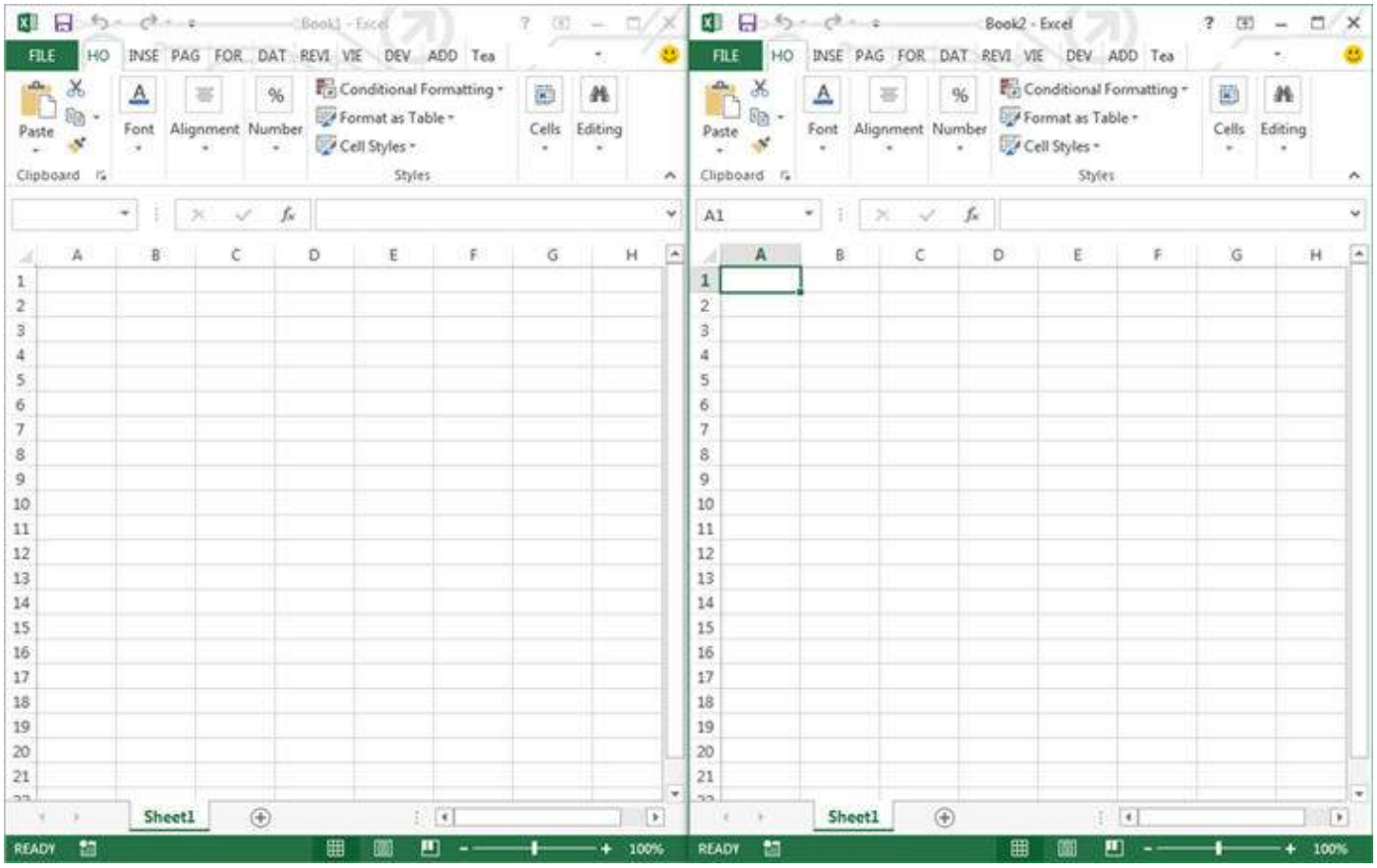
```
Sub ThisWorkbookExample()

'// Let's assume to begin that this code is in the same workbook that is currently active

    ActiveWorkbook.Sheet1.Range("A1").Value = "Foo"
    Workbooks.Add 1
```

```
    ActiveWorkbook.ActiveSheet.Range("A1").Value = "Bar"

    Debug.Print ActiveWorkbook.ActiveSheet.Range("A1").Value '// Prints "Bar"
    Debug.Print ThisWorkbook.Sheet1.Range("A1").Value '// Prints "Foo"

End Sub
```

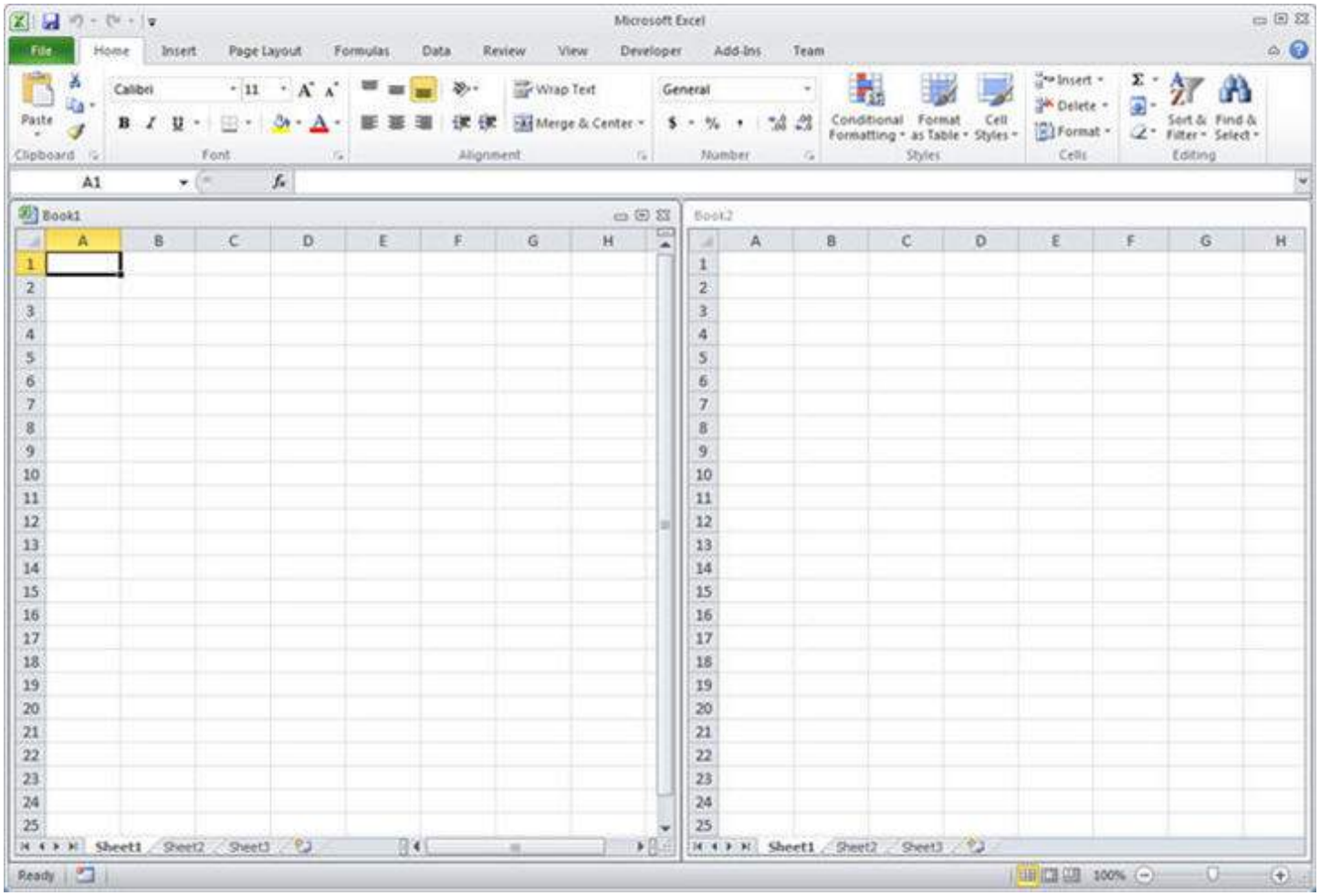
Section 31.4: Single Document Interface Versus Multiple Document Interfaces

> Be aware that Microsoft Excel 2013 (and higher) uses Single Document Interface (SDI) and that Excel 2010 (And below) uses Multiple Document Interfaces (MDI).

This implies that for Excel 2013 (SDI), each workbook in a single instance of Excel contains its **own** ribbon UI:



Conversely for Excel 2010, each workbook in a single instance of Excel utilized a **common** ribbon UI (MDI):



This raise some important issues if you want to migrate a VBA code (2010 <->2013) that interact with the Ribbon.

> A procedure has to be created to update ribbon UI controls in the same state across all workbooks for Excel 2013 and Higher.

Note that :

1. All Excel application-level window methods, events, and properties remain unaffected. (`Application.ActiveWindow`, `Application.Windows` ...)
2. In Excel 2013 and higher (SDI) all of the workbook-level window methods, events, and properties now operate on the top level window. It is possible to retrieve the handle of this top level window with `Application.Hwnd`

To get more details, see the source of this example: MSDN.

This also causes some trouble with modeless userforms. See Here for a solution.

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

Adam	Chapter 4
Alexis Olson	Chapter 29
Alon Adler	Chapter 2
Andy Terra	Chapters 5 and 30
Branislav Kollár	Chapters 1, 4 and 29
Byron Wall	Chapter 21
Captain Grumpy	Chapters 18 and 20
Chel	Chapters 27 and 29
Cody G.	Chapters 1, 28, 29 and 30
Comintern	Chapter 29
curious	Chapter 14
Doug Coats	Chapters 1, 4 and 12
EEM	Chapter 1
Egan Wolf	Chapter 31
Etheur	Chapter 28
Excel Developers	Chapter 11
FreeMan	Chapter 29
genespos	Chapter 29
Gordon Bell	Chapters 1 and 31
Gregor y	Chapter 28
Hubisan	Chapters 2, 14 and 29
Jeeped	Chapters 8, 18, 29 and 30
jlookup	Chapter 18
Joel Spolsky	Chapters 1, 4 and 20
Julian Kuchlbauer	Chapter 28
Kaz	Chapter 1
Kyle	Chapters 28 and 29
Máté Juhász	Chapter 29
Macro Man	Chapters 1, 29, 30 and 31
Malick	Chapters 1, 8, 18, 29 and 31
Masoud	Chapter 26
Miguel_Ryu	Chapter 2
Mike	Chapter 24
Miqi180	Chapter 14
Munkeeface	Chapter 29
paul bica	Chapters 14, 26 and 29
PeterT	Chapters 10, 17 and 29
Portland Runner	Chapters 5 and 29
P□н	Chapter 29
quadrature	Chapters 7 and 15
R3uK	Chapters 6 and 14
RGA	Chapters 1, 14, 23, 28, 29 and 30
Robby	Chapter 24
Ron McMahon	Chapter 28
Sgdva	Chapter 19
Shahin	Chapter 2
Shai Rado	Chapters 1, 12, 14 and 29

Slai	Chapters 8 and 14
Stefan Pinnow	Chapter 29
SteveES	Chapter 3
Steven Schroeder	Chapters 28 and 29
SWa	Chapter 31
T.M.	Chapters 7, 22 and 26
TheGuyThatDoesn'tKnowMuch	Chapter 27
ThunderFrame	Chapter 29
Toast	Chapters 1, 28 and 31
user3561813	Chapter 8
Vegard	Chapters 4 and 8
Verzweifler	Chapter 29
Vityata	Chapter 13
Zsmaster	Chapters 9, 16 and 25

You may also like