How to Record a Macro

1. Do a test run before you press record

2. a) Click on the record macro button at the bottom left-hand side
   b) Use shortcut key Alt + T + M + R
   c) Go to **Developer** tab – Record macro
   d) Go to **View** tab – Record macro

3. Name your Macro, assign a shortcut key & decide where to store it (workbook or personal macro workbook)

4. Run through the steps the macro needs to do

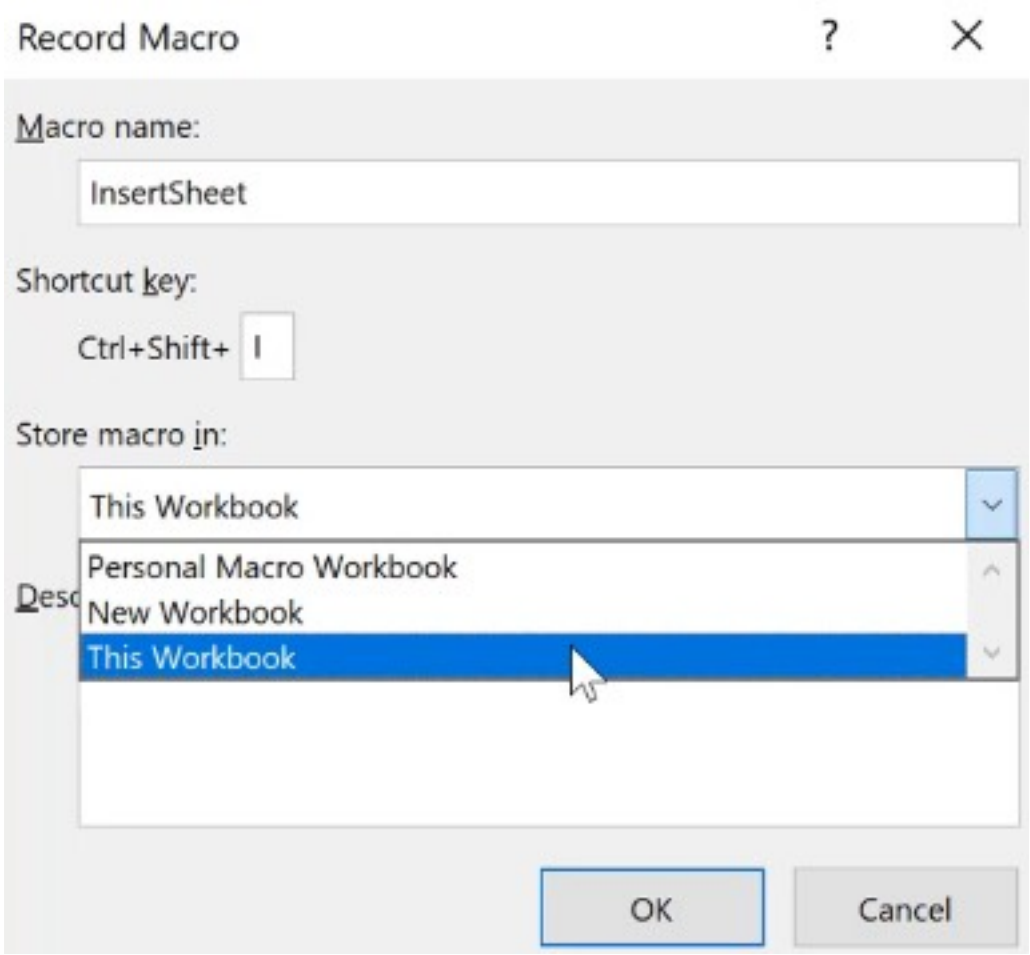
5. Stop the macro

* This Workbook implies the workbook currently active.

* New Workbook means, excel opens a board new workbook and saves the macro there.

* Personal Macro Workbook means, the excel saves the macro in a default location, which is available to all the excel workbooks. This is the XLSTART folder.

* The macro recorder only records the mouse clicks and keystrokes done on the excel workbook.



* Visual Basic Editor (VBE)

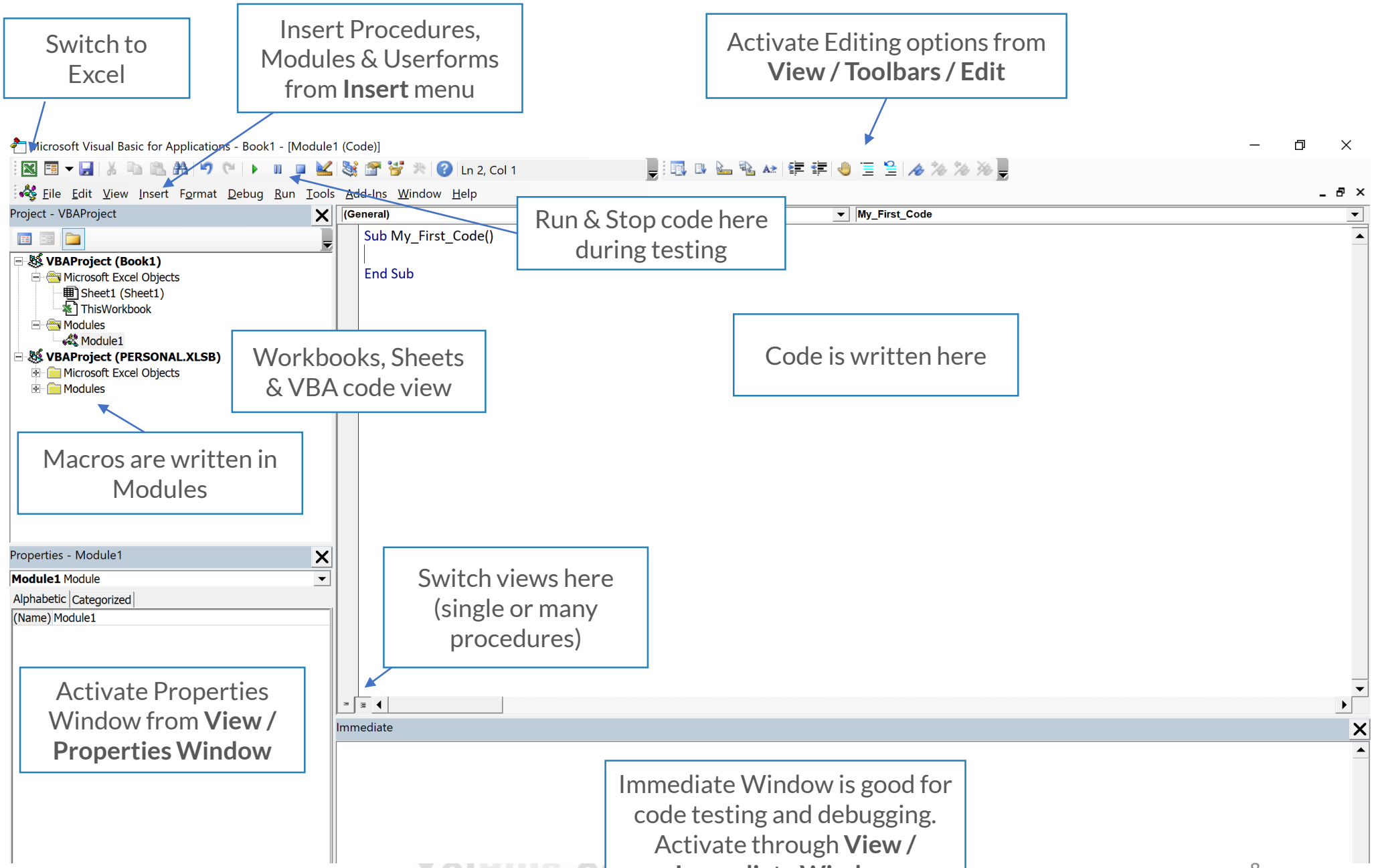
  Close VBE → Workbook stays open

  Close Workbook → VBE closes

* Code specific to a workbook is written in modules.
* Code specific to a sheet is written in that sheet (VBE).
* Actions performed using macros cannot be undone.
* One VBE (Visual Basic Editor) for all open excel file. So VBE serves as a central place. From this central place, we can write code for any workbook that is open.

Visual Basic Editor (VBE)

1. Alt + F11
2. **Developer** tab / **Visual Basic**
3. **View** / **Macros** / **View Macros**, then **Edit**
4. **Right-mouse click on sheet** / **View Code**



7 Ways to run Macros (VBA Code)

1. Use shortcut key **Alt + F8** to get to view macro and then select your macro and run

2. a) Go to Tab **Developer / Macros**
   b) Go to Tab **View / Macros**

3. Use the shortcut key you assigned

4. Add the Macro to the Quick Access Toolbar (For active workbook only)

5. Add the Macro to the Ribbon – Recommended for "general" macros that are saved in the personal macro workbook

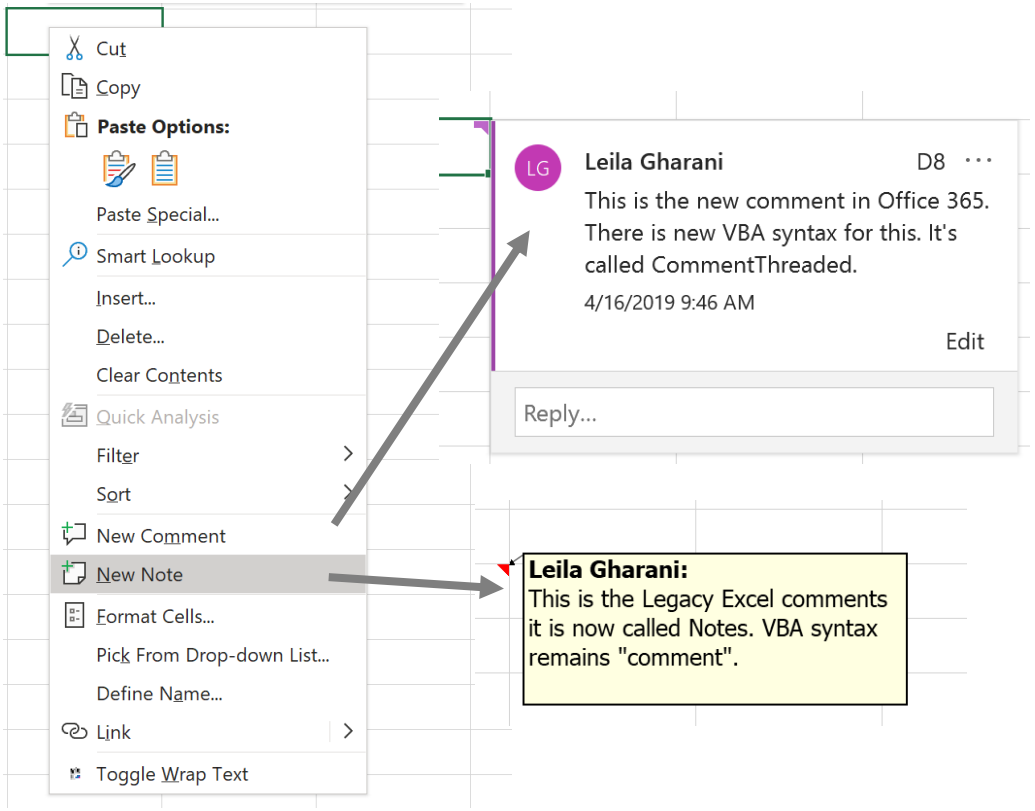
6. Insert any shape or image and assign the macro to it (**Right-mouse click, Assign Macro**)

7. Insert a Form Control button from **Developer tab, Insert, Form Controls**

Change in Comments to Notes

Working with comments & notes

In Office 365 “comments” have been renamed to “notes”.
VBA syntax has **not** changed except we have new syntax for the new comments feature.



Note: If you have Office 365 and you are following along with the lectures, when you see a comment being added to Excel, please add a “note”. VBA syntax has remained unchanged.

The new comments are called threaded comments because you can reply to them. They have a new VBA syntax called CommentThreaded.

The Object Model

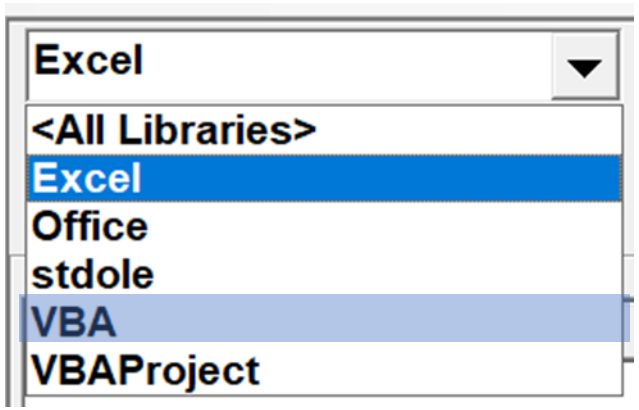
Visual Basic Essential: Understanding the “dot” in code

- # Color Guidelines & Keyboard Shortcuts
- # What the Object Model is
- # Object Properties & Methods
- # How to find the Correct Syntax (property or method)

Visual Basic for Applications (VBA)

Working with VBA (Macro) code

- Writing VBA Code (Macros) = Manipulating objects on the screen
- These objects have code words so you can easily refer to them
- These code words are kept in the reference library



Excel VBA code, generally uses a mix of VBA & Excel Object libraries

To control other applications from Excel, a good option is to activate the reference to that application's object library by going to Tools / References

VB Basics & Color Guidelines

```
Sub my_Macro()

End Sub
```

Most used VBA Procedure is the **Sub Procedure**
This consists of a set of commands the code should execute

```
Function my_Formula()

End Function
```

Function Procedures are commands that create formulas
They return one value (or an array of values)
They can be used as "normal" formulas in Excel or inside other procedures

```
Application.CutCopyMode = False
```

VBA assigns color to keywords and capitalizes code references

```
Range(Selection,
Selection.End(xlDown)).Select
```

Wrong use of code is shown as red

```
' relative Macro
```

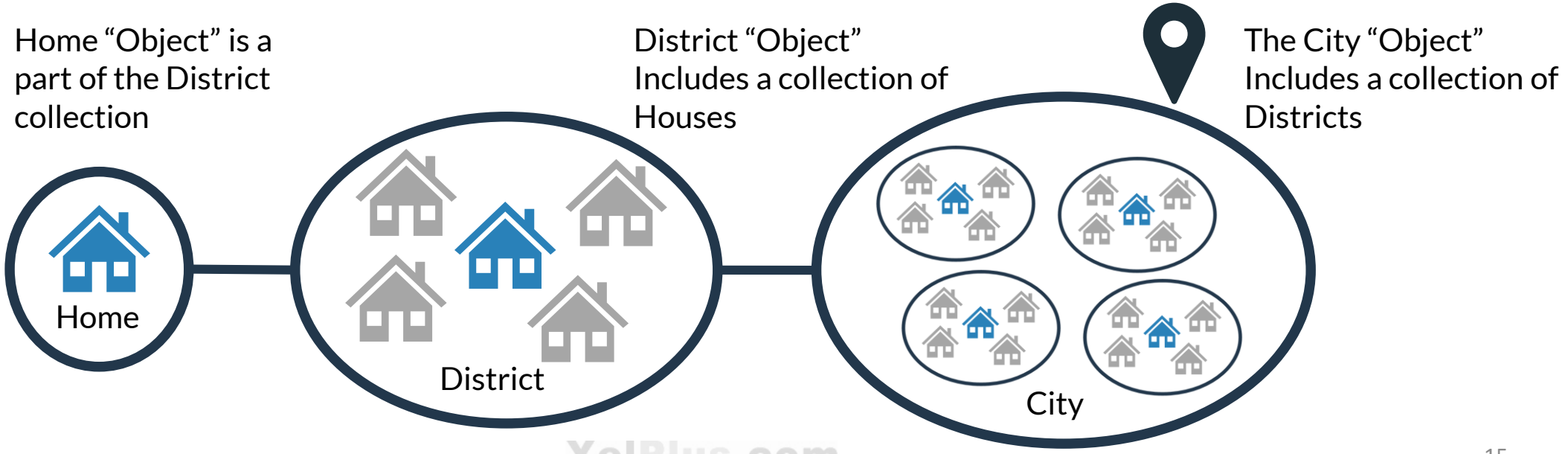
Comments are shown as green

VBE Keyboard Shortcuts

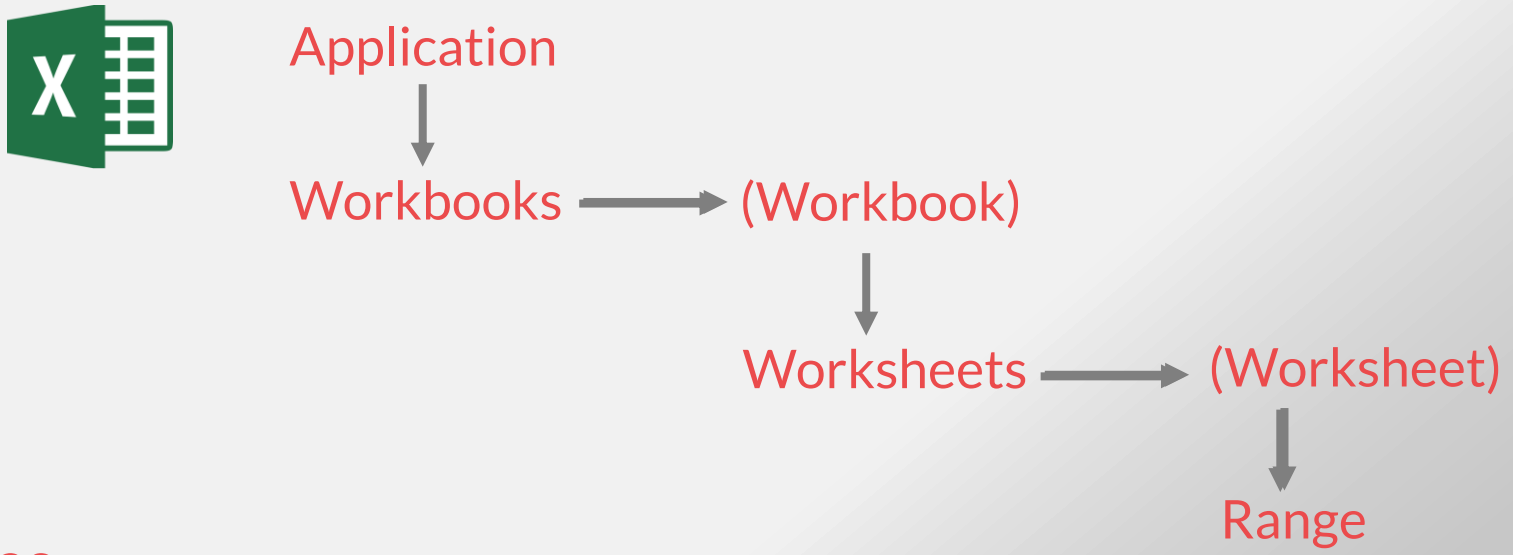
Description	Key
Indent Code	**Tab**
Remove Indent from Code	**Shift + Tab**
Complete word	**Ctrl + Space**
Open object browser	**F2**
Open Code window	**F7**
Display properties window	**F4**
Open online help	**F1**
Run project	**F5**
Step into code	**F8**
Toggle breakpoint	**F9**
Clear all breakpoints	**Ctrl + Shift + F9**

The Object Model

- VBA is Object Oriented.
  Before you perform any actions you need to specify what object to perform on
- VBA Syntax is Object First - then action
- Objects can have relationships to other objects
  This relationship is called the object hierarchy



Excel's Object Model



Examples

Application.Workbooks("Name").Worksheets("WSName").Range("A1")

Application.ThisWorkbook.Worksheets("WSName").Range("A1")

Worksheets("WSName").Range("A1") → assumes active workbook

Range("A1") → assumes active worksheet

ActiveCell → assumes the cell that is marked in the active worksheet

Properties in VBA

Property is what an object **has**

An Analogy

Examples of Property

1. Color
2. Size
3. Type
4. Engine
5. Age
6. .....

Examples of Property

1. Color
2. Size
3. Gender
4. Material
5. Season
6. .....

Properties come after the object hierarchy

Car.Color – might be too broad
Car.**Interior**.Color = Black

Shoe.Gender = Male
Shoe.Heel.Material = ?

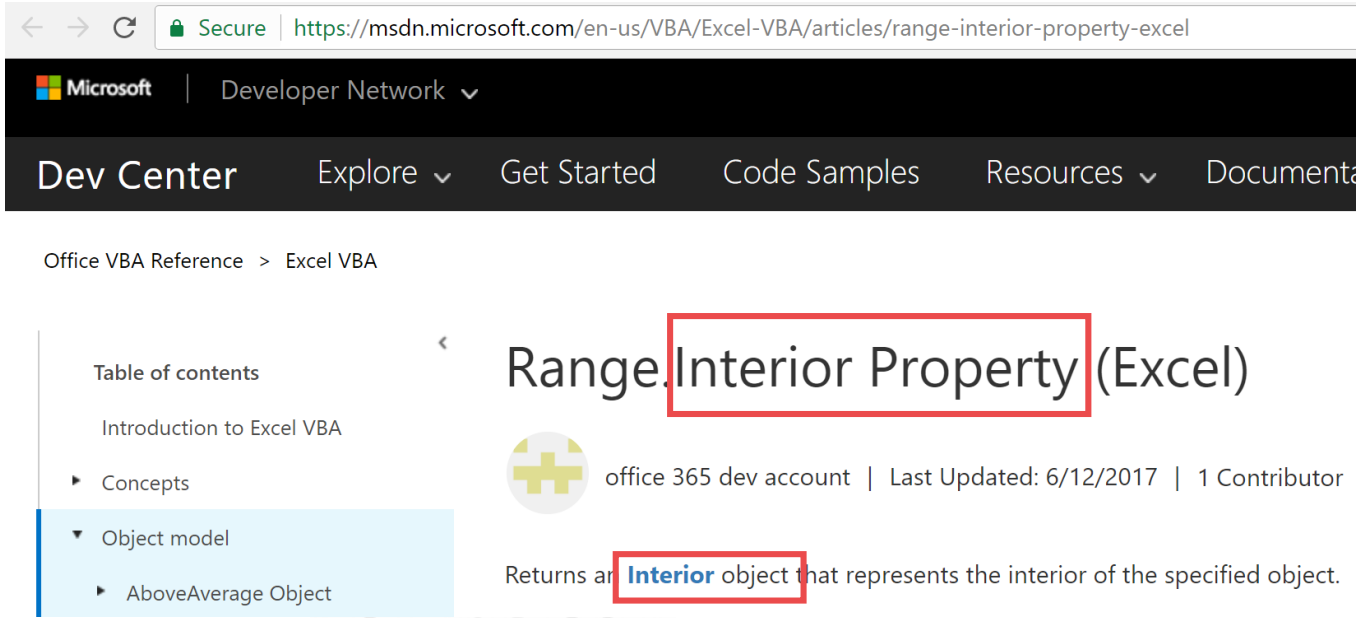
Property Details

- Some Properties don't have details

  Range("A1").Address Range("A1").Value

- Some Properties return an object

  Range("A2").**Interior**.Color Range("A2").**Font**.Color



Property Type: Read-only or Write?

Range("A10").Address → Read-only

Range("A1").Value → Read & Write

Range("A1").Interior.Color Range("A2").Font.Color → Read & Write

Examples

Range("A1").Value = ActiveCell.Address

Range("A1").Interior.Color = vbRed

Range("A1").Font.Color = vbBlue

Methods in VBA

Method is what an object **does**

An Analogy

Examples of Method

1. Start
2. Stop
3. Crash

Examples of Method

1. PutOn
2. TakeOff

Methods can have additional information

How to start the car? Quickly or slowly?
Car.Stop Quickly
Car.Stop StopStyle:=Quickly

Methods can change properties

The “Crash” method would change the size property of the car

Methods Can Have Arguments

Range("A2").Clear → No further arguments

Range("A2").Delete → 1 optional argument (Shift)

Sheet1.Copy → 2 optional arguments (Before, After) which are optional but exclusive

Examples

Range("A2").Delete xlShiftToLeft

Delete([*Shift*])

Range("A3").Copy Range("B3")

Copy([*Destination*])

Range("C3").PasteSpecial xlPasteValues

PasteSpecial([*Paste As* XlPasteType = xlPasteAll], [*Operation As* XlPasteSpecialOperation = xlPasteSpecialOperationNone], [*SkipBlanks*], [*Transpose*])

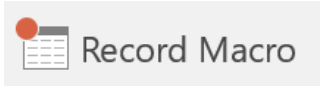
Sheet1.Copy After:=Sheet3

Sheet1.Copy , Sheet2

Copy([*Before*], [*After*])

Find the Correct Property & Method

It's impossible to remember everything…

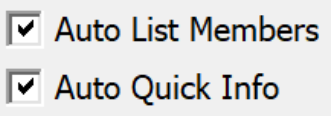


Use the **Macro Recorder** to get object names, properties and methods

Use the Object Library (**F2**)

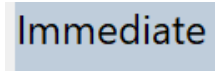
F1

Press **F1** when on an object, property or method to get help for the MSDN Microsoft Help Site



Use **IntelliSense** – Let VBA suggest the right properties & methods. Check options from Tools / Options

Use complete word to get suggestions (**Ctrl + Space**)



Use the **Immediate Window** to query (e.g. color codes) or test code

Search the **internet** & online forums

Key Takeaways: Object Model

1. **You refer to an object through its position in the object hierarchy**. The dot is used as a separator. If you do not specify the parent, Excel assumes it's the active object.
2. **You don't need to select objects to manipulate them**. The macro recorder "selects" but to write code, it's more efficient not to refer directly to objects & properties (exceptions apply).
3. **Objects have specific properties & methods.**
4. **Properties can return a reference to another object.** For example Range("A2").**Interior**.Color: The interior property returns an interior object which has the color property.
5. **Macro and VBA code is kept inside Sub Procedures** (for functions it's Function Procedures).
6. To find the correct object, property or method you can **record macros**, use the **object library** (F2), use **MSDN help** (F1), **Intellisense**, **internet** & **immediate window** to query and test code.

Referencing Ranges, Worksheets & Workbooks

Visual Basic Essential: How to work with cells, sheets & workbooks

- # Methods to Write to One or More Cells
- # Most Useful Range Properties & Methods
- # Finding the Last Row or the Used Region
- # Referencing Worksheets & Workbooks Correctly

Different Methods to Write to Cells

Using Rows, Columns & Range referencing

Range("A1").Value = "1st"

Range("A2:C2").Value = "2nd"

Range("A3:C3,E3:F3").Value = "3rd"

Range("A4,C4") = "4th"

Range("A5", "C5") = "5th"

Range("A" & 6, "C" & 6) = "6th"

Range("A4:C7").Cells(4, 2).Value = "7th"

Range("A1").Offset(7, 2).Value = "8th"

Range("A1:B1").Offset(8, 1).Value = "9th"

Range("LastOne").Value = "10th"

	A	B	C	D	E	F
1	1st					
2	2nd	2nd	2nd			
3	3rd	3rd	3rd		3rd	3rd
4	4th		4th			
5	5th	5th	5th			
6	6th	6th	6th			
7		7th				
8			8th			
9		9th	9th			
10	10th					

ActiveCell.Value = "1st"

Cells(1, 1).Value = "1st"

Range(Cells(6, 1), Cells(6, 3)).Value = "6th"

Cells(1,1).Offset(7, 2).Value = "8th"

Cell A10 is called “LastOne” in Name Manager

Referencing Entire Rows / Columns

Using Range, Cells & Offset referencing

Rows("12:14").RowHeight = 30	Rows 12, 13 & 14 have a row height of 30
Range("16:16,18:18,20:20").RowHeight = 30	Rows 16, 18 & 20 are changed. 17 & 19 are not touched.
Columns("E:F").ColumnWidth = 10	Columns E to F have a column width of 10
Range("H:H,J:J").ColumnWidth = 10	Columns H & J are changed. Column I is not touched.
Range(Columns(1), Columns(3)).ColumnWidth = 5	Columns A, B & C have column width of 5
Cells.Columns.AutoFit	All columns are adjusted by autofit

```vba
Sub ReferToCells()
    Dim i As Integer
    i = 8
    Workbooks("04-S4_ReferencingRanges_Start.xlsm").Worksheets("referencing Cells").Cells.Clear
    'Cells.Clear
    'Range("A1").Value = "1st"
    Cells(1, 1).Value = "1st"   'This can also be written as Cells(1, 1) = "1st"
    Range("A2:C2").Value = "2nd"
    Range("A3:C3,E3:F3").Value = "3rd"
    Range("A4,C4") = "4th"
    Range("A5", "C5").Value = "5th"
    Range("A" & 6, "C" & 6) = "6th"
    Range("A" & i, "C" & i) = "8th"
    Range(Cells(7, 1), Cells(7, 3)).Value = "7th"
    Range("A6:C11").Cells(4, 2).Value = "9th"
    Range("A1").Offset(9, 2).Value = "10th"
    Range("A1:B1").Offset(10, 3).Value = "11th"
    Range("Lastone").Value = "12th"


    ' Referencing entire rows and/or columns
    Rows("12:14").RowHeight = 30
    Range("16:16,18:18,20:20").RowHeight = 30

    Columns("E:F").ColumnWidth = 15
    Range("H:H,J:J").ColumnWidth = 12
    Range(Columns(1), Columns(3)).ColumnWidth = 25

    Cells.Columns.AutoFit

End Sub
```

	A	B	C	D	E	F	G	H	I	J
1	1st									
2	2nd	2nd	2nd							
3	3rd	3rd	3rd		3rd	3rd				
4	4th		4th							
5	5th	5th	5th							
6	6th	6th	6th							
7	7th	7th	7th							
8	8th	8th	8th							
9		9th								
10			10th							
11				11th	11th					
12		12th								
13										
14										
15										
16										
17										
18										
19										
20										
21										
22										

On this sheet we will practice the different methods to refer to write to cells.

Most Useful Range Properties

As found in my projects

Code Execution	Description (Click here for more)	Type
Value	Show the underlying value in a cell. This is the default property of the range object.	Read / Write
Cells	Returns a cell or range of cells within a range object	Read / Write
End	Returns the last cell of the range. Similar to Ctrl + ↓ or ↑ or → or ←	Read-only
Offset	Returns a new range based on offset row & column arguments	Read / Write
Count	Returns the number of cells in a range.	Read-only
Column / Row	Returns the column / row number of a range. If you select more than one cell, column / row returns the first occurrence in the range.	Read-only
CurrentRegion	Used with other properties such as .address returns the range of data	Read-only
EntireColumn / EntireRow	Returns a range object that represents the entire row or column	Read-only
Resize	Changes the size of the range by defining the rows & columns for resizing	
Address	Shows the range address including the $ signs.	Read-only
Font	Returns a font object that has other properties (e.g. bold)	Read / Write
Interior	Used with other properties such as .Color property to set colors	Read / Write
Formula	Places a formula in a cell. To make sure your VBA code is compatible with other languages of Excel, your VBA formulas should use the English syntax. You can record these with the macro recorder. The macro recorder uses FormulaR1C1 syntax. If you'd like to have formulas in your language of Excel, you need to use FormulaLocal.	Read / Write
NumberFormat	Define Number format (uses English version)	Read / Write
Text	Returns the data as string & includes formatting.	Read-only
HasFormula	Returns True, False or Null if the range has a mix	Read-only

Most Useful Range Methods

As found in my projects

Code Execution	Description (Click here for more)	Type
Copy	This is a practical method because it has paste destination as its argument. This means you just need one line of code.	
PasteSpecial	Allows usage of Excel's Paste Special options. To use more than one option, repeat the line of code with the new option.	
Clear	Deletes contents and cell formatting in a specified range.	
Delete	Delete the cells and shifts the cell around the area to fill up the deleted range. The delete method uses an argument to define how to shift the cells. Select XLToLeft or XLUp.	
SpecialCells	Returns a range that matches the specified cell types. This method has 2 arguments. XlCellType is required (such as cells with formulas or comments) and an optional argument to define more detail if constant and formula cell type is used in the first argument.	
Sort	Sorts a range of values	
PrintOut	Also a method of the worksheet object	
Select	Used by the macro recorder to select a cell but when writing VBA, it is not necessary to select objects. Code is faster without selecting.	

4 Methods to Find the Last Row

In these examples the results are written to a cell. Later we will store these in variables.

1 Use the End Property of the Range Object

```
Range("K6").Value = Cells(Rows.Count, 1).End(xlUp).Row
‘or
Range("K6").Value = Range(“A4”).End(xlDown).Row
‘Example for last column
Range("K8").Value = Cells(4, Columns.Count).End(xlToLeft).Column
```

2 Use the CurrentRegion Property of the Range Object

```
Range("K10").Value = Range("A4").CurrentRegion.Rows.Count
```

3 Use the SpecialCells Method of the Range Object

```
Range("K11").Value = Cells.SpecialCells(xlCellTypeLastCell).Row
```

4 Use the UsedRange Property of the Worksheet Object

```
Range("K12").Value = Application.ActiveSheet.UsedRange.Rows.Count
```

Copy & Resize Variably Sized Ranges

Copy & PasteSpecial Methods

Copy method for a variable sized range

```
Range("A4").CurrentRegion.Copy Range("J4")
```

Or for a fixed range:

```
Range("A4:E10").Copy Range("J4")
```

	A	B	C	D	E
4	**Company**	**Business Unit**	**Actual Revenue**	**Budget Revenue**	**Variance**
5	Entity A	BU_1	10,200	10,404	-2%
6	Entity B	BU_1	12,240	12,485	-2%
7	Entity C	BU_1	14,688	14,982	-2%
8	Entity D	BU_1	19,776	17,978	10%
9	Entity E	BU_2	10,300	10,506	-2%
10	Entity F	BU_2	12,360	12,607	-2%

PasteSpecial method to use Excel's Paste Special options

```
Range("A4").CurrentRegion.Copy
Range("J20").PasteSpecial xlPasteValuesAndNumberFormats
'to add more paste special options add a new line
Range("J20").PasteSpecial xlPasteComments
```

Use the Resize property to return a changed range

```
Range("A4").CurrentRegion.Offset(1, 0)
_.Resize(Range("A4").CurrentRegion.Rows.Count - 1).Copy Range("A20")
```

How to Best Reference Worksheets

Use the code name of the sheet (give your own code name in the property window)

shDest.Range("A3").Value

2 advantages

- User can be free to change the tab name
- IntelliSense works perfectly with code names (doesn't work with ActiveSheet because ActiveSheet could also be a chart sheet…)

Use the Worksheets collection and refer to the tab by name

Worksheets("Destination").Range("A3").Value

Use the index number of the worksheet (can be risky)

Worksheets(6).Range("A3").Value

ActiveSheet .Range("A3").Value

How to Best Reference Workbooks

Required if more Workbooks need to be referenced

ThisWorkBook (to reference the current workbook where macro is written)

Use the Workbooks collection and use the name of the workbook. The workbook needs to be open, otherwise you need to use the Open statement first.
Workbooks ("YourWorkBookName.xlsx")

ActiveWorkBook
Make sure the correct workbook is active

Workbooks (1)
The following is risky as it depends on the order the workbooks have been opened

Key Takeaways: Referencing Ranges

1. **Different methods to reference cells**
   Use Range(“A1”) style to reference a single cell.
   Range(“A1”, “C1”) or Range (“A1:C1”) for many cells.
   Use the Cells or Offset property

2. **Different methods to find the last row**
   Range("K6").Value = Cells(Rows.Count, 1).End(xlUp).Row
   Range("K10").Value = Range("A4").CurrentRegion.Rows.Count
   Range("K11").Value = Cells.SpecialCells(xlCellTypeLastCell).Row

3. **Copy a variably sized range with the CurrentRegion Property**

4. **Use the Code Name of the WorkSheets** to reference worksheets – alternatively use the worksheet names

5. **Use ThisWorkBook property to reference the workbook the macro is in.** For other Workbooks use the workbooks collection.

Understanding Variables & Data Types

Visual Basic Essential: Working with Variables

- # What are Variables and Why Use Them
- # Data Types & Best Practice
- # Working with Object Variables
- # Re-using Variables in Other Procedures

What are Variables?

- Variables are nick-names that store a value
- Names are assigned with the equal sign

myTitle = Range("A1").value

Why use Variables?

- Variables make your code easy to read
- Variables simplify code maintenance and allow for complex coding

myTitle = Range("A1").value

If you change the title to cell "B1", you just have to update the myTitle reference once in the code instead of every single place the reference to "A1" is made.

Data Types for Variables

Variables can accommodate different data types

Data Type	Memory used	Range
Byte	1 byte	0 to 255
Integer	2 bytes	-32,768 to 32,768
Long	4 bytes	-2,147,483,648 to 2,147,483,647
Boolean	2 bytes	True / False
Double	8 bytes	Very large negative to positive range with high precision (also used for %)
String	1 byte per char	Depends on length
Object	4 bytes	Any object
Date	8 bytes	01,1,0100 to 12,31,9999
Currency	8 bytes	Very large negative to positive range up to 4 decimal places
Variant	16 bytes (more with characters)	Any value – can also hold values such as “Empty”, “Nothing” and “Null”

Declaring Variables, Arrays & Constants

…& the role of Option Explicit

1 Declaring variables by defining a suitable data type has the outcome:

1. Your code will run faster
2. Your code will be less prone to mistakes

```
Dim myText As String
```

```
Dim LastRow As Long, FirstRow as Long
```

2 A group of similar variables that has one name can be declared as an Array

```
Dim myMonth(1 To 12) As String
```

3 If you need to refer to a value that never changes, you can use Constants

```
Const myScenario As String = "Actual"
```

```
Const ProfitCen As Long = 9999
```

Use Option Explicit (to minimize mistakes)
In VBE, add a check-mark: Tools / Options / Require Variable Declaration

Using Object Variables

Set Statement in VBA

Variables can also hold objects. Common objects are:

1. Workbook Object — Dim NewBook As Workbook
2. Worksheet Object — Dim NewSheet As Worksheet
3. Range Object — Dim NewRange As Range

To assign variables to objects, you need to use the SET statement

```
Set NewBook = Workbooks.Add
```

All "usual" VBA assignments are actually done with the LET statement. However this is an optional keyword. That's why it's usually not used. In case you come across it, you know what it does.

LastRow = Rows.Count– is the same as:

```
Let LastRow = Rows.Count
```

Variable Scope

Is the variable reusable in other procedures?

Procedure: Variables exists only when the procedure runs. Dim is inside the procedure. Memory is freed after procedure ends.

```
Sub Defining_Variables ()
  Dim LastRow As Long, FirstRow as Long
  '---code
End Sub
```

Module: Variable exists for ALL Procedures within the Module. Dim is outside any procedure. Typically right below Option Explicit. Value is kept in memory after procedure completes.

```
Option Explicit
Dim LastRow As Long, FirstRow as Long

Sub Defining_Variables ()
  '---code
End Sub
```

All Modules & Procedures: Variable exists for ALL Modules and ALL Procedures. Use the keyword "Public" to declare these variables. Can be declared in any Module before the first procedure. Value is kept in memory after procedure completes.

```
Option Explicit
Public LastRow As Long, FirstRow as Long

Sub Defining_Variables ()
  '---code
End Sub
```

Key Takeaways: Variables & Data Types

1. **Use Option Explicit** (to minimize mistakes)
   In VBE, add a check-mark: Tools / Options / Require Variable Declaration

2. **Pick the data type which uses the least bytes** but can still handle the data you want to store in memory for example

   Byte: worksheet numbers, months, column headers (smaller data sets)
   Long: To loop through rows
   Integer: to loop through smaller data sets – e.g. master data
   Double: Decimals, very large numbers and numbers where precision is required

3. **Think about the scope of your variables.** Procedure, Module or Project? Declare these accordingly.

4. **Use the SET statement to assign variables to objects.** Data type assignments do not need a statement but they could use the optional keyword "LET".

Looping Through Collections & Making Decisions

A VBA Essential for Efficient Macros

- # With…End With Constructs for Easier Coding
- # Looping Through Collections, i.e. sheets, cells in One Go
- # Making Decisions with IF & Select Case Constructs
- # Change the Program Flow with GoTo Statements

With…End With for Easier coding

The benefits of the With…End With construct are:

- Faster code **writing**
- Easier code **maintenance**
- Faster code **execution**

Before

```
Set myRange = Range("A10", "A" & Cells(Rows.Count, 1).End(xlUp).Row)
myRange.Font .Name = “Arial”
myRange.Font.Size = 12
myRange.Font.Bold = True
```

After

```
Set myRange = Range("A10", "A" & Cells(Rows.Count, 1).End(xlUp).Row)
  With myRange.Font
     .Name = “Arial”
     .Size = 12
     .Bold = True
  End With
```

The less dots you have, the faster your code will run.

For…Each to Loop Through Collections

Looping through Worksheets, Ranges etc. in One Go

VBA provides an easy method to loop through a collection of similar objects. For example:

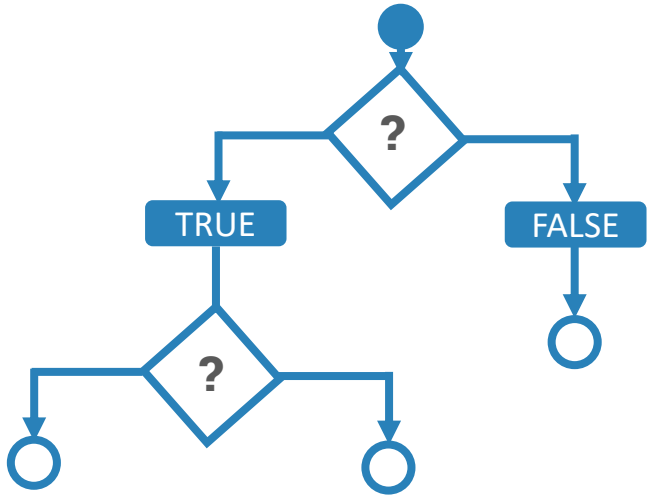
- Execute code for all **Worksheets in a Workbook**
- Follow instructions for all **Cells inside a Range**
- **Run code for** each **Comment inside the Worksheet Comments collection**

```
Dim Sh As Worksheet
 For Each Sh In ThisWorkbook.Worksheets
   Sh.Protect "test“
 Next Sh
```

```
Dim cell As Range
 For Each cell In ActiveSheet.UsedRange
   ‘[Instructions – you can use IF here to check whether a condition
   ‘ is met – also to exclude parts of the collection]
 Next cell
```

IF…Then Statements for Conditional Outcomes

Decide which way your code should run



- Similar to Excel formulas, you can use **IF…Then** in VBA
- You can handle as **many conditions** as you see fit
- You can also use **AND, OR** statements

Simple IF: 1 Line

```
If Range("B3").Value <> "" Then Range("C3").Value = Range("B3").Value
```

Simple IF: More Lines

```
If Range("B3").Value <> "" Then
   Range("C3").Value = Range("B3").Value
End If
```

IF…Then…ElseIf…Else

```
Dim Sh As Worksheet
  For Each Sh In ThisWorkbook.Worksheets
    If Sh.Name = "Purpose" Then
      'don't allow formatting etc...
       Sh.Protect " test"
    ElseIf Sh.CodeName = "shWith" Then
       'protect without password
        Sh.Protect
    Else
       'allow formatting for all other sheets
       Sh.Protect "test", , , , , True, True, True
    End If
  Next Sh
```

Select Case, as Alternative for Many IFs

In case your IF..Then construct gets too complex

Select Case is a good alternative to IF…ElseIf…Else Constructs:

- You have a better **overview**
- Easier code **maintenance**
- Faster code **execution.** VBA leaves the Select Case construct the moment it arrives at a TRUE. Place the likely cases at the top of the code.

```
Select Case Range("B3").Value
  Case 1 To 200
    Range("C3").Value = "Good"
  Case 0
    Range("C3").Value = ""
  Case Is > 200
    Range("C3").Value = "Excellent"
  Case Else
    Range("C3").Value = "Bad"
End Select
```

To select different specific values, you can also write:

```
Case 1,3,7
```

Goto Statement to Change Program Flow

You can skip code lines with the Goto Statement. Why would you want this?

- Mainly for **error handling**
- Execute a different part of code depending on a **condition**

To use GoTo

- Type the name of the label with colon (or number without colon)
- You might need to Exit sub before the label if you have a message box or another VBA statement

```
Sub Simple_GoTo ()
  Range("D3").Value = ""
   If IsError(Range("B3")) Then GoTo GetOut
   Range("C3").Value = Range("B3").Value
   Exit Sub
GetOut:
   Range("D3").Value = "You have an error in the cell“
End Sub
```

Use this mainly for Error Handling. Otherwise your code can become too confusing to understand.

Key Takeaways: Collections & Decisions

1 **Use With … End With Construct**
To optimize code writing and execution

2 **Use For…Each Loop**
To loop through members of a collection.

3 **Use Select Case Statement once IF gets too Complex**
Put the most likely cases on top

4 **Use GoTo Statement to Jump to Different Sections of Code**
Use labels to put “bookmarks” in part of code, such as “Leave:” then use GoTo Leave to jump to that section of code.

Useful Built-in Functions

Become familiar with common VBA & Worksheet Functions

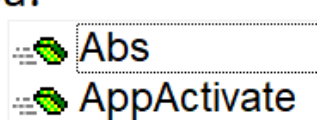
- Most Useful VBA Functions
- Display a Message Box (also with Yes, No buttons)
- Display an Input Box (VBA InputBox function)
- Input Box that Selects Ranges (Excel InputBox method)

VBA & Worksheet Functions

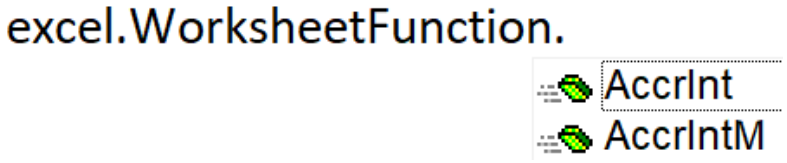
Similar to worksheet formulas…

VBA has built-in functions to help you run calculations.
Excel has built-in worksheet functions to help you with your calculations.

VBA Functions: To see a list of all functions, type VBA followed by "."



Worksheet Functions: To see a list of all functions, type Excel or Application followed by "."



If VBA has an equivalent function to Excel, only the VBA function is available for use!

Excel	VBA Equivalent
ABS ()	Abs function
ISBLANK ()	ISEMPTY function
LEN ()	LEN function
LOWER ()	LCASE function
RAND ()	RND function
TODAY ()	DATE function
TYPE ()	TYPENAME function
UPPER ()	UCASE function

Useful VBA Functions

Text Handling
(String manipulation)

Text handling	Description
InStr	Returns the position of one string within another
LCase	Converts string to lowercase
Left	Returns the left hand side of a string based on specified number of characters
Len	Returns the number of characters in a string
Mid	Returns a specified number of characters from a string
Replace	Returns a string and replaces a subset of the string with another string
Right	Returns the right hand side of a string based on specified number of characters
Space	Returns a string the contains spaces
StrConv	returns a string converted to uppercase, lowercase, proper case or Unicode.
Trim	Returns a string without leading or trailing spaces
UCase	Converts string to uppercase

Constants	Description
VbNewLine	Creates a new line in message box
VbNullString	Returns a zero length string
VbTab	Adds a tab space in message box

Useful VBA Functions

Date Handling

Date handling	Description
Date	Returns the current date
Day	Returns the day of the month of a date
Hour	Returns the hour of a time value
IsDate	Returns true if variable is a Date type
Minute	Returns the minute of a time value
Month	Returns the month of a date
MonthName	Returns a string for the month of a date (abbreviation possible)
Now	Returns the current date and time
Second	Returns the seconds portion of time
Time	Returns the current system time
Timer	Returns the number of seconds since midnight (good for adding a timer in code)
Weekday	Returns a number for the day of the week
WeekdayName	Returns a string for the day of the week
Year	Returns the year of a date

Useful VBA Functions

Formatting, Converting & Number Handling

Formatting & Converting	Description
CDate	Converts expression to Date data type
CInt	Converts expression to Integer data type
CLng	Converts expression to Long data type
CStr	Converts expression to String data type
Format	Shows an expression in the desired format
Str	Returns a string from a number
Val	Returns a number (double) from a string

Number handling	Description
Abs	Absolute value of a number
FormatNumber	Formats an expression as number
FormatPercent	Formats an expression as percentage
IsNumeric	Returns True if a cell or value is numeric
Round	Rounds a number

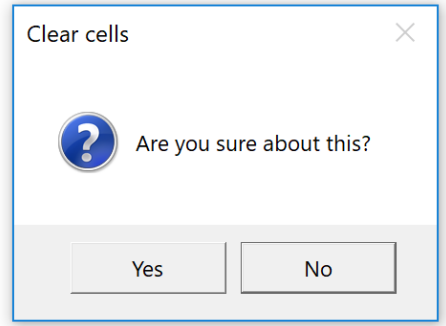
Useful VBA Functions

Text Files & Other Useful Functions

Text files & Dir	Description
CurDir	Returns the current path
Dir	Returns the name of the file or directory
EOF	Equals True if end of text file is reached
FileDateTime	Date and time when a file was last modified
FileLen	Number of bytes in a file
FreeFile	Returns the next available file number for text files

Other useful functions	Description
DoEvents	Control is given back to the operating system to process other events
InputBox	Displays a dialog box for user to input
IsEmpty	Returns True if a cell is blank or a variable hasn't been initialized
LBound	Returns the smallest value in an array
MsgBox	Displays a dialog box
RGB	Returns a number that reflects the RGB color
TypeName	Returns a string that specifies the data type of a variable
UBound	Returns the largest element of an array

VBA's MsgBox Function



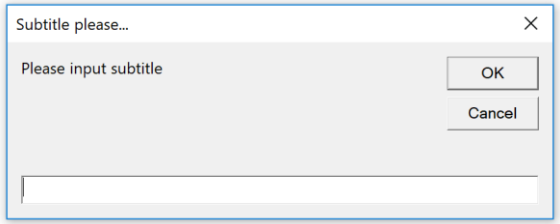
- Easily create info dialog boxes to communicate with the user & get simple responses such as Ok, Yes, No or Cancel.
- The message box is a function, i.e. your macro is paused until the user provides a response.
- You can use the MsgBox function by itself (don't use brackets) if you don't need a response from the user. If you need a response, the message box will return a result which you can capture in a variable (use brackets).

```
MsgBox "Hello " & Excel.Application.UserName, , "Welcome!"
```

```
Dim Answer As VbMsgBoxResult
Answer = MsgBox("Are you sure about this?", vbYesNo + vbQuestion + vbDefaultButton2, "Clear cells")
  If Answer = vbYes Then
            Range("A7:B9").Clear
  Else
            Exit Sub
  End If
```

Tip: use + to add more constants (symbols, buttons…)

VBA's InputBox Function



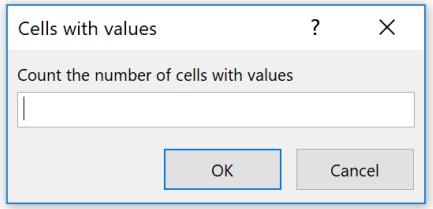
Easily create input dialog boxes to capture user input.

Input Box always returns a string. If you write the result of the input box to a cell, Excel will automatically convert the answer to the correct type. For example if you input a number, it will be recognized as a number in the cell.

If you'd like to use the answer in your code, and the answer should be a number, you need to convert it to a number with a conversion function. For example the Val function. To validate if the input is a number, you can use IsNumeric function

```
Dim myInp As String
  myInp = VBA.InputBox("Please input subtitle", "Subtitle please...")
  If myInp = "" Then Exit Sub
  Range("A2").Value = Excel.WorksheetFunction.Proper(myInp)
```

Excel's InputBox Method



3 benefits of using the Input Box Method

1. You can specify the data type for input (not restricted to string)
2. Excel performs a validation automatically
3. You can select ranges

```
CName = VBA.InputBox _
  ("Please input new Customer Name", "Customer Master")
```

```
Set cRange = Application.InputBox _
  ("Count the number of cells with values", "Cells with values", , , , , , 8)
```

Code	Description
0	Formula
1	Number
2	String
4	Boolean
8	Range
16	Error value
64	Array of values
1+2 = 3	Number + String

Key Takeaways: Built-in Functions

1 **Review the List for Useful VBA Functions**
Take some time and review the list and test some of the functions you think could come in handy for your projects.

2 **MsgBox Function**
Use the message box to inform the user or get a “Yes”, “No” answer from them.
You can also use the message box for debugging your code.

3 **VBA InputBox Function**
VBA’s Input box function is a quick way to allow the user to interact with your code. If you’d like to add some basic validation checks – i.e. for numbers or text, use Excel’s InputBox method instead.

4 **Excel’s InputBox Method**
Excel offers its own version of the input box. It has the additional feature of allowing the user to select a range or do basic validation checks.

Debugging, Error Handling & Procedure Scope

For Faster & Efficient Macros

- # Methods to Debug Code
- # How to Handle Errors
- # Writing Code that's Faster & Efficient
- # Procedures Scope & Passing Arguments

Methods to Debug Your Code

To help you Pinpoint Errors… in addition to the MsgBox

1. **F8** or select **Debug** from the menu and then **Step Into**.
2. Add breakpoints with **F9** or **Debug** menu / **Toggle Breakpoint**. You can play the code until a certain place and then you can step in with F8. Add as many as you need.
3. **Hover your mouse** on a variable to see its result.
4. Use the **Immediate Window.** Type "Debug.Print" followed by your variable name in the code. This writes the result to the immediate window. You can also type in code directly in this window. Use the "?" to get the answer. Activate it from the View tab.
5. Use the **Locals Window** to see the value and characteristics associated with each variable. Activate it from the View tab.
6. Use the **Watch window** to keep your eye on certain variables. Right-mouse click on a variable or statement and "add to watch". Alternatively drag to the window. You can change the variable property directly by typing it in. Activate watch from the View tab.

Error Handling: Different Methods

Common error handling examples

1

```
Sub Jump_to_End()
 On Error GoTo Leave

  [code instructions]

Leave:
End Sub
```

Uses a label to jump to the end of the code the moment an error occurs.

2

```
Sub Resume_Then_Normal()

 On Error Resume Next

  [code instructions]

 On Error GoTo 0

  [code instructions]

End Sub
```

Suppresses certain errors and then resumes normal error handling for the remaining instructions.

3

```
Sub Handle_Based_on_Error_Type()
 On Error GoTo ErrorHandle

  [code instructions]

  Exit Sub

ErrorHandle:
 Select Case Err.Number
  Case 424
   Exit Sub
  Case Else
   MsgBox "An error has occurred."
  End Select
End Sub
```

More detailed error handing by type. Add “Exit sub” to leave the macro if error-free.

Faster & Efficient VBA Code

Suppressing Pop-ups & flickering screen

Suppress

```
With Application
    .StatusBar = "Wait“
    .ScreenUpdating = False
   .Calculation = xlCalculationManual
   .DisplayAlerts = False
End With
```

- Use the Status bar to let the user know the macro is running
- Screen flickering
- Formula calculations
- Excel alerts – for example when deleting worksheets or closing a workbook.

Restore

```
With Application
    .ScreenUpdating = True
   .Calculation = xlCalculationAutomatic
   .DisplayAlerts = True
   .CutCopyMode = False      (In case you used PasteSpecial)
   .StatusBar = ""
End With
```

Use this block of code when writing to cells, working with different worksheets or workbooks & other longer tasks.

Procedure Scope & Passing Arguments

- Procedures are Public by Default.
- Private Procedures are accessible to other procedures in the same module. They are also not shown in the Macro list.
- To execute one procedure from another procedure you can use the optional keyword **CALL**. If Procedures have arguments you need brackets if you use the CALL keyword.

```
Call myCalc (i, c)
MyCalc i, c
```

ByRef is the default way of passing arguments (passes the variable itself). Variable's value could change once it's back. E.g. Sub myCalc(ByRef GetValue, myPercent)

ByVal only processes a copy of the variable and does not change the variable's value. Variable keeps it's value once it's back.
E.g. Sub myCalc(ByVal GetValue, myPercent)

```
Sub GetMyValue()
  [Instructions]
  Call myCalc(myValue, p)
End Sub
```

```
Private Sub myCalc(GetValue, myPercent)
  GetValue = GetValue * myPercent
End Sub
```

Key Takeaways: Debug & Handle Errors

1 **Keep the Debug Options List Handy**
You might prefer one way over another when debugging. I personally prefer to use F8 to step through the code → the immediate window for testing results and for more complex checks → the watch window. I use the Locals window for testing Arrays.

2 **Error Handling**
Implement some type error handling if you are planning to share your tool with others. Try to catch what you can from your side but allow others to test your code and provide feedback.

3 **Suppressing Certain Events**
For longer codes make sure you suppress screen updating or display of alerts. This makes your code run smoother and faster.

4 **Procedure Scope**
Procedures are Public by default. Sub Procedures that are declared as Private do not show up in the Macros dialog box. They can be used inside Public Sub procedures.

Looping in VBA

Controlling the Flow of Code

- For...Next Counter Loops
- Do Until / Do While Loop
- Find Method for Quicker Results
- Time your Code (to pick the faster version)

For Next Counter Loops

Simple Looping construct that's based on a counter.
Exit For statement leaves the loop.

```
Sub Simple_For()
    Dim i As Long
    Dim myValue As Double
LastRow = ActiveSheet.UsedRange.Cells(ActiveSheet.UsedRange.Rows.Count, 1).Row
For i = 4 To LastRow
       myValue = Range("F" & i).Value
      If myValue > 400 Then Range("F" & i).Value = myValue + 10
      If myValue < 0 Then Exit For
    Next i
End Sub
```

```
Sub Delete_Hidden_Filtered_Rows()
    Dim r As Long
    With ActiveSheet
       LastRow = .UsedRange.Cells(.UsedRange.Rows.Count, 1).Row
       For r = LastRow To 4 Step -1
          If .Rows(r).Hidden = True Then
             .Range("H" & r) = "X"
         '     .Rows(r).Delete
          End If
       Next r
    End With
End Sub
```

Do…Until / Do…While Loop

Do…While Loop runs **as long** as a specified condition is met
Do…Until Loop runs **until** a specified condition is met
Exit Do command leaves the loop immediately

```
Sub Simple_Do_Until_V1()
Startcell = 8
   Do Until ActiveSheet.Range("A" & Startcell).Value = ""
      Range("B" & Startcell).Value = Range("A" & Startcell).Value + 10
       Startcell = Startcell + 1
   Loop
End Sub
```

You can also use Do…Loop (without While or Until)

While Wend is similar to Do loop and is still included in VBA for compatibility purposes.

```
Sub Simple_Do_While()
Startcell = 8
   Do While ActiveSheet.Range("A" & Startcell).Value <> ""
      Range("C" & Startcell).Value = Range("A" & Startcell).Value + 10
       Startcell = Startcell + 1
   Loop
End Sub
```

```
Sub Input_Number_Only()
   Dim myAnswer As String
   Do While IsNumeric(myAnswer) = False
      myAnswer = VBA.InputBox("Please input Quantity")
      If IsNumeric(myAnswer) Then MsgBox "Well Done!"
   Loop
End Sub
```

Find Method for Quicker Results

FIND method provides a quick way to find the answer without Looping

expression . **Find**(***What*** , ***After*** , ***LookIn*** , ***LookAt*** , ***SearchOrder*** , ***SearchDirection*** , ***MatchCase*** , ***MatchByte*** , ***SearchFormat***)

expression A variable that represents a **Range** object.

From MSDN.Microsoft.com

Parameters

Name	Required/Optional	Data Type	Description
What	Required	**Variant**	The data to search for. Can be a string or any Microsoft Excel data type.
After	Optional	**Variant**	The cell after which you want the search to begin. This corresponds to the position of the active cell when a search is done from the user interface. Notice that *After* must be a single cell in the range. Remember that the search begins after this cell; the specified cell isn't searched until the method wraps back around to this cell. If you do no specify this argument, the search starts after the cell in the upper-left corner of the range.
LookIn	Optional	**Variant**	Can be one of the following **XlFindLookIn** constants: **xlFormulas** , **xlValues** , or **xlNotes** .
LookAt	Optional	**Variant**	Can be one of the following **XlLookAt** constants: **xlWhole** or **xlPart** .
SearchOrder	Optional	**Variant**	Can be one of the following **XlSearchOrder** constants: **xlByRows** or **xlByColumns** .
SearchDirection	Optional	XlSearchDirection	The search direction.
MatchCase	Optional	**Variant**	**True** to make the search case sensitive. The default value is **False** .
MatchByte	Optional	**Variant**	Used only if you have selected or installed double-byte language support. **True** to have double-byte characters match only double-byte characters. **False** to have double-byte characters match their single-byte equivalents.
SearchFormat	Optional	**Variant**	The search format.

FIND with DO LOOP for Many Matches

```
Sub Many_Finds()
   Dim CompID As Range, FirstMatch As Variant, i As Long
   Dim Range("D3:D6").ClearContents
   i = 3
   Set CompID = Range("A:A").Find(What:=Range("B3").Value, _
   LookIn:=xlValues, LookAt:=xlWhole)
   If Not CompID Is Nothing Then
      Range("D" & i).Value = CompID.Offset(, 4).Value
      FirstMatch = CompID.Address
      Do
         Set CompID = Range("A:A").FindNext(After:=CompID)
         If CompID.Address = FirstMatch Then Exit Do
         i = i + 1
         Range("D" & i).Value = CompID.Offset(, 4).Value
      Loop
   End If
End Sub
```

Add a Timer to Test & Speech to Inform

Adding a timer (VBA function) helps you test different versions of code for the same task

```
Dim Start
Start = Timer 'seconds since midnight

    [code instructions]

Debug.Print Timer - Start
```

1. Write your code
2. Add a Timer
3. Comment out the current code
4. Write new code and test
5. Use the code version that runs fastest
6. Remove the Timer or comment out

Allowing your computer to “speak” is a good way of informing the user that a long procedure is now completed. Speakers should be turned on.

```
    [code instructions]
Application.Speech.Speak "Job Done!"
End Sub
```

Useful VBA Statements

Common Statements

General	Description
Const	Declare a constant
Dim	Declares variables
End	Exits program – also ends procedures, with statements, etc.
Function	Declare a function procedure
Kill	Deletes a file
Let	Assigns a variable to an expression (is optional - can be omitted)
Like	Returns True if one string can be matched with another
Load	Loads an object (like a userform) but doesn't show it
Mid	Replaces characters in a string with other characters
Option Explicit	Forces variable declaration
Public	Declares a variable to be used in all procedures
ReDim	Change the dimension of the array
Set	Assign an object to a variable
Sub	Declares the name of the sub procedure

Useful VBA Statements

Controlling code execution

Code Execution	Description
Call	Go to another procedure
Do Until – Loop	Loops through instructions until a condition becomes true
Do While – Loop	Loops through instructions while a condition is true
Exit Do	Exit Do-Loop code
Exit For	Exit For-Next loop
Exit Sub	Exit subroutine
For – Each -Next	Loops through a collection
For - Next	Loops through instructions for a specified number of times
GoTo	Jump to a specific place in the code
If – Then - Else	Check for conditions
On Error	Give specific instructions for the case when an error occurs
Resume	Resumes code execution
Select Case – End Select	Check for conditions
While - Wend	Loops through instructions as long as a condition is true (included for compatibility purposes)

Useful VBA Statements

Text files & Directories

Text file & Dir	Description
Close	Close a text file
Get	Read data from a text file
Input #	Read data from a text file
Line Input #	Reads a line of data from text file
MkDir	Creates a new directory
Name	Renames a file or directory
Print #	Writes data to a text file
Open	Opens a Text file
Write #	Writes data to a text file

Key Takeaways: LOOPING IN VBA

1 **For … Next Counter Loop**
Very powerful and flexible way of looping through cells. It's safer to use than the Do loop: The loop only runs for a specific number of times depending on the lower and upper values of the control variable.

2 **Do Loop**
Variations include Do…Until, Do…While and just Do (with a check for a condition when the loop can be exited). These come in handy when you don't know the number of times the loop should run. Tip → Use F8 first before running the code to make sure it works properly.

3 **Find Method**
Find method can be faster than the For…Next and Do Loop methods when looking for one or many matches.

4 **Use a Timer & Speech to Inform**
Use VBA's Timer function to test different variations of code, in case you aren't sure which one is more efficient. Speech.Speak method is a good way of vocally informing the user the macro has finished running.

Working with Arrays

For Faster Procedures

- One Dimensional Fixed Arrays
- Working with Dynamic Arrays (& preserving values)
- Two Dimensional Arrays
- Variant Arrays

How to Declare & Fill Arrays

Declaring Arrays	VBA Code
Declaring a Fixed Array	Dim MonthArray(1 To 12) As String
Declaring a Dynamic Array	Dim MonthArray() As String
Declaring a two Dimensional Fixed Array	Dim MonthInfo(1 To 12, 1 To 2) As Variant
Referring to the Lowest / Highest Index in the array	LBound(MonthArray) UBound(MonthArray)
Define the size of the dynamic array	ReDim MonthArray(1 To Cnt)
Keep the existing values inside the array	ReDim Preserve Cust(1 To 3)

Fill a 1 Dimensional Array

```
Dim MonthArray(1 To 12) As String
  Dim i As Byte
  'Fill up the month array
  For i = 1 To 12
    MonthArray(i) = Range("mymonths").Cells(i, 1).Value
  Next i
```

Fill a 2 Dimensional Array

```
Dim MonthInfo(1 To 12, 1 To 2) As Variant
  Dim r As Long
  Dim c As Long
    'fill the rows
  For r = 1 To 12
    'fill the columns
    For c = 1 To 2
    MonthInfo(r, c) = Cells(r + 4, c).Value
    Next c
  Next r
```

Working with Variant Arrays

- Similar to working with arrays in Excel
- Keep values of many cells inside one variable (compartment)
- Manipulate the members of this variable
- Write back to the range in one go

Looping inside arrays is faster than looping in the cells.

Example of Variant Array

```
Sub Write_to_Variant_Array()
    'notice brackets are not required for variant arrays
    Dim QuantityValue As Variant
    Dim r As Long
   'QuantityTbl is a range of cells defined in name manager
    QuantityValue = Range("QuantityTbl").Value
    For r = 1 To UBound(QuantityValue, 1)
    'Add 10 to the existing value
    QuantityValue(r, 1) = QuantityValue(r, 1) + 10
    Next r
    Range("QuantityTbl") = QuantityValue
End Sub
```

Key Takeaways: Arrays

1 **One Dimensional Array**
This is similar to highlighting one row or one column in Excel. Fixed one-dimensional array is defined during the DIM statement: Dim MonthArray(1 To 12) As String.

2 **Two Dimensional Array**
This is similar to highlighting a matrix which includes a few columns and rows in Excel. Think of the rows as the 1st dimension in an array and the columns as the 2nd dimension.

3 **Dynamic Array**
The exact size of the array is defined during code execution. Use the REDIM statement to define the size and then fill the array.

4 **Variant Array**
This is similar to Excel arrays. A range of cells is defined as a variant. It can be directly manipulated with a loop and values written back to cells in one go. This is a faster method than using a For…Next loop.

Working with Files, Folders & Text Files

Provide Flexibility & Create Data Exports

- Check if a File / Folder Exists (DIR)
- Allow the User to Select a File or Folder
- Create CSV Files
- Work with Text Files (for more control)

Check if File or Folder Exist (DIR)

DIR Function uses a path argument and returns the name of the file or folder. If path name is not found, DIR returns a zero length string (“”).

Dir [(path [, attributes])
]

Constant	Value	Description
vbNormal	0	(Default) Files with no attributes
vbReadOnly	1	Read-only files
vbHidden	2	Hidden files
VbSystem	4	System files
vbDirectory	16	Directories or folders

```
Sub File_Exists()
Dim FileName As String
   FileName = Dir("C:\Users\LG\Desktop\VBA\S2_*.xls*")
If FileName = VBA.Constants.vbNullString Then
      MsgBox "file does not exist"
   Else
      MsgBox FileName
   End If
End Sub
```

Note: Wildcards like * and ?
Can be used in the path string.

User Selects File(s): GetOpenFileName

GetOpenFileName Method to allow the user to select one or more files

GetOpenFileName (1 File)

```
Sub Get_Data_From_File()
  Dim FileToOpen As Variant
  Dim OpenBook As Workbook
   FileToOpen = Application.GetOpenFilename _
   (Title:="Browse for your File", _
    FileFilter:="Excel Files (*.xls*),*.xls*")
   If FileToOpen <> False Then
      Set OpenBook = Application.Workbooks.Open(FileToOpen)
      '[Instructions]
      OpenBook.Close False
   End If
End Sub
```

GetOpenFileName (Many Files)

```
Sub Select_Many_Files()
  Dim FileToOpen As Variant
  Dim FileCnt As Byte
  Dim SelectedBook As Workbook
   'Pick the files to import - allow multiselect
   FileToOpen = Application.GetOpenFilename _
   (Filefilter:="Excel Files (*.xlsx), *.xlsx", _
   Title:="Select Workbook to Import", MultiSelect:=True)

  If IsArray(FileToOpen) Then
    For FileCnt = 1 To UBound(FileToOpen)
     Set SelectedBook = Workbooks.Open
           _(FileName:=FileToOpen(FileCnt))
     ' [Instructions]
     SelectedBook.Close
    Next FileCnt
   End If
End Sub
```

User Selects a Folder (to Loop Through)

Get Folder Name

```
Sub Loop_Inside_Folder()
  Dim FileDir As String
  Dim FiletoList As String
  With Application.FileDialog(msoFileDialogFolderPicker)
      .Title = "Please select a folder"
      .ButtonName = "Pick Folder"
      'Cancel show value = 0, -1 there was a selection
      If .Show = 0 Then
        Exit Sub
      Else
         FileDir = .SelectedItems(1) & "\"
      End If
   End With
  FiletoList = Dir(FileDir & "*xls*")
  Do Until FiletoList = ""
    FiletoList = Dir
  Loop
End Sub
```

FileDialog Property provides an easy way to allow the user to select a folder. You can write a loop to go through each file inside the folder.

Calling the Dir function again inside the loop without any arguments moves on to the next file in the folder.

Export Sheets as CSV

Export as CSV

```
Sub Save_as_CSV()

    Dim NewBook As Workbook
    Dim FileName As String
    Application.ScreenUpdating = False
    Application.DisplayAlerts = False
   FileName = Application.ThisWorkbook.Path & "\TestTextCSV.csv"
    Set NewBook = Workbooks.Add
    ShCSV.Copy before:=NewBook.Sheets(1)
    With NewBook
        .SaveAs FileName:=FileName, FileFormat:=Excel.xlCSV
        .Close
    End With
    Application.ScreenUpdating = True
    Application.DisplayAlerts = True
    MsgBox "Your CSV file was exported.", vbInformation

End Sub
```

To export a sheet as CSV, first copy the sheet to a new Workbook (make any adjustments necessary) and the save this workbook as a csv file. Make sure to turn off display alerts.

Writing and Reading a Text File

VBA Open Statement (not the Open Method of Workbook) opens a file for reading or writing. This gives your more control over the layout (for example the delimiter)

```
Open pathname For mode [ Access access ] [ lock ] As [ # ] filenumber [ Len = reclength ]
```

Required statements are:

- **Pathname** – path and name of file to be opened
- **Mode** – Input (read the file only), Output (write or read the file), Append (add to the bottom of file)
- **Filenumber** – the next available file number. Use #1 if this is the first file opened or use the FreeFile Function to get the next available file number

Write - This writes a line of text to the file surrounding it with quotations
Print - This writes a line of text to the file without quotations

```
FileName = Application.ThisWorkbook.Path & "\ProjectActivity.csv"
   Open FileName For Output As #1
    ' [Instructions to loop through range]
Print #1, myValue
Close #1
```

Key Takeaways: Files & Folders

1. **DIR Function**
   Use DIR to check if a file or folder exists. Remember you can use wildcards * ? In the path.

2. **GetOpenFileName Method**
   Use this method to allow the user to browse and pick a file. Set the filter using the correct extension if you'd like to restrict them to certain file types.

3. **FileDialog Property**
   Use this property to allow the user to browse for and select a folder (you can also use it to select files). Use DIR to get the name of the first file inside the folder and then DIR again without any arguments to loop through the folder, i.e. move on to the next file.

4. **Export Sheet as CSV**
   Export the Sheet to a new Workbook first and then save the workbook as a csv file format.

5. **Export Data as Text** (with delimiter of your choice)
   Use Open, Print and Close statements to create a text file in the directory. Then loop through your data set and write each cell to the file.

Excel Tables, Formulas & Pivot Tables

A VBA Essential for Efficient Macros

- # Using Excel Formulas in VBA
- # Excel Tables - Use the Right Syntax
- # Pivot Tables - What you Need to Know

Using Excel Formulas In VBA

To ensure compatibility with different Excel languages, **use the macro recorder** to record formulas. The formulas are always recorded with the default English reference library. These work on all the different languages of Excel.

FormulaR1C1 property uses the row and column numbering to refer to cells. **Formula** property uses the A1 Type of referencing (although it can interpret R1C1 referencing as well).

To display formulas in local language, use the **FormulaLocal** property. E.g: Range("F20").FormulaLocal instead of Range("F20").Formula

```
Dim LastRow As Long
LastRow = Range("A" & Cells.Rows.Count).End(xlUp).Row
Range("F9").FormulaR1C1 = "=IF(VALUE(LEFT(RC[-4],1))=8,RC[-1]-50,"""")"
Range("F9").AutoFill Destination:=Range("F9:F" & LastRow), Type:=xlFillDefault
Application.Calculate
```

Working with Excel Tables

Table Task	VBA Code
Declaring Tables	Dim myTable As ListObject Set myTable =Activesheet.ListObjects(“Table1")
Whole Table	myTable.Range.Select
All Data – No Headers	myTable.DataBodyRange.Select
Count Rows	myTable.Range.Rows.Count
Specific Row	myTable.ListRows(5).Range.Select
Specific Column exclude Headers	myTable.ListColumns(“Quantity”).DataBodyRange.Select
Header Only	myTable.HeaderRowRange.Select
Add a Row to the bottom	myTable.ListRows.Add , False
Add a 2nd Row	myTable.ListRows.Add 2
Add a Column to the end	myTable.ListColumns.Add
Add a 2nd Column	myTable.ListColumns.Add 2
Add a Column to the end	myTable.ListColumns.Add
Rename the last Column	myTable.ListColumns(myTable.ListColumns.Count).Name = “NEW"
Add Formula to Row 1 Column 6	myTable.DataBodyRange(1, 6).FormulaR1C1 = "=(“put formula”)"

Working with Pivot Tables

PivotTable Task	VBA Code
Declaring Pivot Tables	Dim PT As PivotTable Set PT = ActiveSheet.PivotTables("PivotTable1")
Refresh Pivot Table	PT.PivotCache.Refresh
Refresh All Pivot Tables	ActiveWorkbook.RefreshAll
Data Range of Pivot	PT.DataBodyRange.Font.FontStyle = "Arial"
Specific Pivot Field	PT.PivotFields("Sum of Quantity").NumberFormat = "#,##0"
Count number of Pivot Cache in Workbook	ActiveWorkbook.PivotCaches.Count
Memory used for PivotCache 1 in bytes	ActiveWorkbook.PivotCaches(1).MemoryUsed
Cache Index number of a Pivot Table	ActiveSheet.PivotTables("PivotTable2").CacheIndex
Change Pivot Cache of one Pivot to another Pivot	ActiveSheet.PivotTables("PivotTable2").ChangePivotCache ("PivotTable1")
Change Pivot Cache to another Table	PT.ChangePivotCache ActiveWorkbook.PivotCaches.Create(SourceType:=xlDatabase, SourceData:="compnew")

Key Takeaways: Formulas & (Pivot) Tables

1 **Excel Formulas in VBA**
Use the macro recorder to record your formula and then copy it to your code. This makes sure the English language reference is used which will ensure your formula will be compatible with other Excel language packs (the English reference library is available by default).

2 **Excel Tables**
Working with tables, ensures more flexibility for your macros. Use the ListObject class to declare a table variable. Refer to the hand-out for the remaining statements.

3 **Pivot Tables**
Pivot Cache is the brain of the Pivot Table. When creating a Pivot Table, a Pivot Cache is created first. Make sure you don't have duplicate Pivot Caches in your reports (unless you need them).

CONGRATULATIONS

You've completed the second major milestone of the course!

I hope you've learnt new techniques which you can apply to your projects and I hope you're enjoying the process as much as I am!

Interacting with Other Applications

(Word, PowerPoint, Adobe etc.)

- # PDF: Save Specific Tabs as a PDF File
- # Email: Create Personalized Emails (with Attachments)
- # Word: Change Specific Text in Microsoft Word
- # PowerPoint: Add Slides with Excel Content

Excel's In-built Methods / Dialogs

For Easy Interaction

ExportAsFixedFormat Method to create PDF files

You can loop through many tabs and export each to a separate PDF file or export all tabs as one PDF file

```
ActiveSheet.ExportAsFixedFormat xlTypePDF, [FileName]
```

xlDialogSendMail to create E-mails with Attachments
Uses the default email client

You can loop through different cells to get individual email addresses, email subject information and respective Excel sheets to be sent as attachment

```
Application.Dialogs(xlDialogSendMail).Show [email], [subject]
```

Early Versus Late Binding

To work with other applications you need to create an instance of the object.
You have two options:

1. Early Binding
2. Late Binding

Early Binding: Go to Tools / References / Place a check mark on the application
Advantage: You have access to the Object Reference Library of this application
Disadvantage: The reference is version specific

```
Dim PowerPointApp as PowerPoint.Application
Set PowerPointApp = New PowerPoint.Application
```

Late Binding: You create the object at runtime.
Advantage: It's version independent.
Disadvantage: You don't have access to the object library of this application

```
Dim PowerPointApp as Object
Set PowerPointApp = CreateObject ("PowerPoint.Application")
```

CreateObject & GetObject

CreateObject → Creates a new Instance of the application (starts a separate copy of the program)

```
Dim PowerPointApp as Object
Set PowerPointApp = CreateObject (“PowerPoint.Application”)
PowerPointApp.Visible = True
[code]
PowerPointApp.Quit
Set PowerPointApp = Nothing
```

GetObject → Uses an existing instance of the application or starts the application with a file loaded (you need to specify the path)

```
Set WordApp = GetObject(, "Word.Application")
```

Key Takeaways: Other Applications

1. Use **ExportAsFixedFormat** method to create PDF files

2. Use **xlDialogSendMail** to create simple E-mails (no need to activate the reference library)

3. Use **Early Binding** to connect to other applications if you'd like to get IntelliSense.
   Do this by activating the Object Reference Library of the respective application.

4. Use **CreateObject** to create a new Instance of the application

5. Use **GetObject** to use an existing instance of the application

Worksheet & Workbook Events

Use the Power of Events to Automatically Run Macros

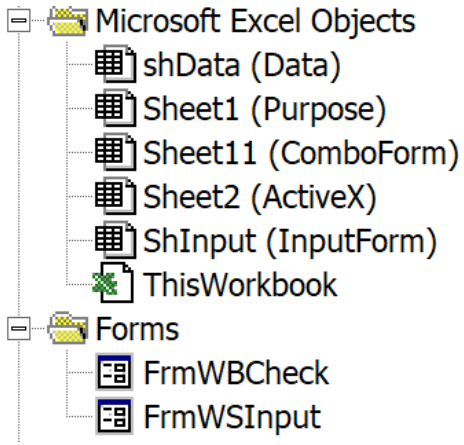
- # Understanding Workbook Events (e.g. Open, Deactivate…)
- # Worksheet Events such as Selection Change & Change
- # Practical Examples: Auto Refresh Pivot Tables
- # Resetting a Dependent Drop-down (data validation)

What are Events?

Excel is programmed to monitor various events:

Workbook Events → Events that occur in a Workbook. For example, opening a Workbook triggers the Open event. Closing a Workbook, triggers the Before_Close event.

Worksheet Events → Events that occur in a Worksheet. For example, changing a cell value triggers the Change event. Selecting a cell, triggers the Selection_Change event.



In addition to these, we have Chart events & UserForm events.

Event procedures need to be placed in the right place. Otherwise they will not execute.

Useful Workbook Events

Common Workbook Events	How it's triggered
Open & BeforeClose	Workbook is opened & workbook is about to be closed
Activate & Deactivate	Workbook is activated & deactivated
SheetActivate & Sheet Deactivate	Any worksheet is activated & deactivated
BeforeSave & AfterSave	A workbook is about to be saved & after it is saved
BeforePrint	A workbook or parts of the workbook are about to be printed
NewSheet	A new sheet is created in the workbook
SheetChange	The contents of any worksheet are changed
SheetCalculate	Any worksheet in the workbook is calculated

```
Private Sub Workbook_Open()
   Sheets(1).Select
   MsgBox "Welcome to this Lesson on Events"
   Sheets(1).ScrollArea = "A1:K15"
End Sub
```

Useful Worksheet Events

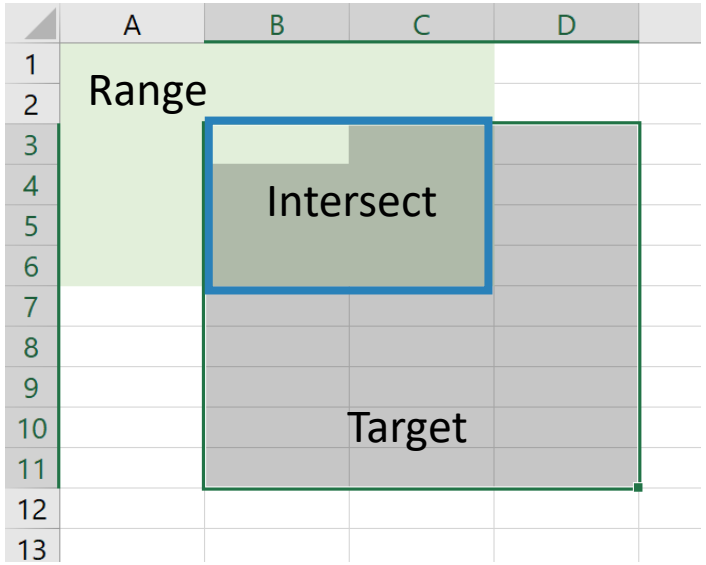
Common Worksheet Events	How it's triggered
SelectionChange	The selection in the sheet is changed
Change	A cell input value is changed (does not apply to formula results)
Calculate	The worksheet is calculated
Activate & Deactivate	The worksheet is activated & deactivated
BeforeDelete	The worksheet is about to be deleted
BeforeRightClick	The worksheet is right clicked
BeforeDoubleClick	The worksheet is double clicked

```
Private Sub Worksheet_SelectionChange(ByVal Target As Range)
  Cells.Font.Color = VBA.ColorConstants.vbBlack
  Target.Font.Color = VBA.ColorConstants.vbBlue
End Sub
```

2 Useful Tips When Using Events

1 **Disabling Events** → Application.EnableEvents = False turns off any worksheet or workbook events. You might need this to prevent an infinite loop. E.g. You might need to change a value in a cell inside the Worksheet_Change event. This would re-trigger the event. To make sure it doesn't, turn off events before this line of code and then turn it on again after the line of code Application.EnableEvents = True.

2 **Intersect Method** → Returns a range object which represents the intersection of different ranges. If ranges don't intersect, it returns nothing.

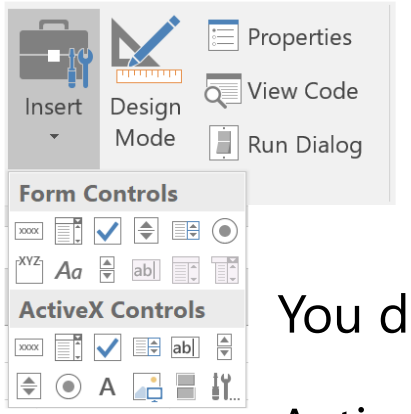



```
Private Sub Worksheet_Change(ByVal Target As Range)
   If Not Intersect(Target, Range("A1:C6")) Is Nothing Then
      [Instructions]
   End If
End Sub
```

Working with UserForms & Form Controls

For Interactive & User-Friendly tools

- # How to Use ActiveX Controls
- # ActiveX Controls: Practical Examples
- # UserForm Basics (Order of Events, Checklist)
- # UserForms: Practical Examples



ActiveX Controls

Embed UserForm controls directly in your Worksheet

You don't necessarily need to use Macros: **You can link to values to cells**.

ActiveX controls are **more flexible** than Form Controls.

Design mode: You need to switch to design mode to adjust properties of ActiveX controls

Use **properties window** to adjust control name, fill range and linked cell

ActiveX controls have **event-handler procedures** which is kept in the code window of the sheet the ActiveX is in.

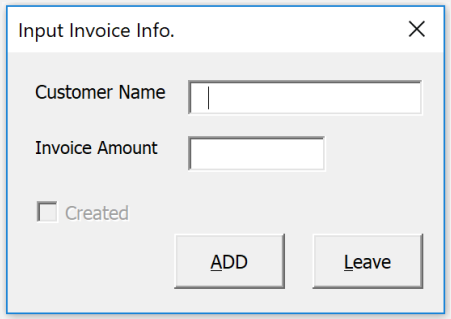
Add a new Sheet

```
Private Sub btInsert_Click()
    Sheets.Add after:=Sheets(Sheets.Count)
End Sub
```

Steps to Your UserForm

1. Insert a UserForm (right-mouse click in Project View and Insert → UserForm)
2. Add Controls to the Form (use copy/paste, align options etc.)
3. Adjust Properties for the UserForm and each Control in the Properties Window
4. Write procedures for the controls in the code window of the UserForm (use keyword ME when referring to the UserForm (instead of referencing the form's name)
5. Create a Sub Procedure in a Module to show the UserForm
6. Unload UserForm once it's no longer needed

Order of UserForm Events



Load → Initialize

Show → Initialize → Activate

UnLoad → QueryClose → Terminate

Hide

Close Button → UnLoad

Each Control has specific events & a default event

Filling up List Boxes or Combo Boxes:

- in the control property (RowSource) of properties window
- Before the UserForm is Shown (during Initialization)

UserForm Checklist

- Does your UserForm have a simple & easy to read layout?
- Is text spelled correctly?
- Is the tab order set correctly?
- Can controls be accessed with a hot key?
- Can the User Cancel or Close the form?
- Do the controls behave as required?

UserForms can be exported and imported to other projects.
Right-mouse click on form → Export File & File → Import File to load the form

Using a ListBox or ComboBox

- You can add items to a ListBox or ComboBox by specifying a range (or a named range) in the RowSource property at design time

Current Master Data

Gary Miller	WenCaL	2120 Eagle Lane	Underwood	MN 56586		20%	Gary.Miller@WenCaL.com
James Willard	Blend	2748 Haul Road	Eagan	MN 55121			James.Willard@Blend.com
Richard Elliot	Voltage	2529 Paradise Lane	Pomona	CA 91766			Richard.Elliot@Voltage.com
Robert Spear	Inkly	1792 Parrish Avenue	Santa Clara	CA 95054			Robert.Spear@Inkly.com
Roger Mun	Sleops	123 Parrish Avenue	Santa Clara	CA 95054			Roger.Mun@Sleops.com
Paul Garza	Kind Ape	67 Parrish Avenue	Santa Clara	CA 95054			Paul.Garza@Kind.com
Robert Marquez	Pet Feed	523 Confederate Drive	Amber	NY 13110			Robert.Marquez@Pet.com
Natalie Porter	Right App Learning	605 Old Dear Lane	New York	NY 10013			Natalie.Porter@Right.com
Kim West	Right App Play	605 Old Dear Lane	New York	NY 10013			Kim.West@Right.com
Stevie Bridge	Hackrr	4372 Salina Street	Salinas	CA 93901			Stevie.Bridge@Hackrr.com
Andre Cooper	Robert	30 Copthorne Way	CB5 0HP CAMBRIDGE	United Kingdom	GB718186591		Andre.Cooper@Perino.co.uk
Crystal Doyle	Silvrr	Wolfensbergstrasse 109	3656 Ringoldswil	Switzerland			Crystal.Doyle@Silvrr.com
Robert Musser	Dasring	Hauptstrasse 14	3587 Salzburg	Austria	ATU56754789	20%	Rmusse@Dasring.com
Daniel Garrett	Rehire	Ganggasse 142	1250 Salzburg	Austria	ATU56755475	20%	Daniel.Garrett@Rehire.at

- You can link the value of a ListBox or ComboBox to a cell using the ControlSource property or use the Value property in VBA code
- You can add items to a ListBox and ComboBox at runtime using the AddItem method
- To get the Value for a ListBox (single) use Value or ListIndex property. For a Multiselect ListBox use Selected() property
- You can allow MutiSelect in a ListBox and display the items with CheckBoxes or OptionButtions (for single selection)
- If no items in a ListBox are selected, ListIndex = -1
  The first item in the ListBox has a ListIndex of 0

CONGRATULATIONS

You've completed the LAST major milestone of the course!

I hope you've learnt new techniques which you can apply to your projects and I hope you're enjoying the process as much as I am!

Creating Customized Functions / Formulas

Make your own formulas in Excel

- # How Function Procedures Work
- # Functions that Get only Text or only Number from cell
- # Functions in Sub Procedures
- # Functions that Sum Based on Cell Color

How Function Procedures Work

You can use Function Procedures in:

- **Excel Formulas:** Use your created functions as normal formulas
- **Other Sub Procedures:** Use them in Sub Procedures

When are Function Procedures Recalculated?

- Custom Functions are recalculated **just like other Excel formulas** – i.e, when one of its arguments change.

  You can force calculation whenever a value in a cell is changed by using **Application.Volatile True**

  To force re-calculation of all formulas use **Ctrl + Alt + F9**

Example of a Simple Function Procedure

```
Function YearsSince(myDate) As Long
    YearsSince = Year(Date) - Year(myDate)
End Function
```

Key Takeaways Function Procedures

Use Arguments instead of cell references

Use Function Procedures (Private or Public) to make coding much simpler

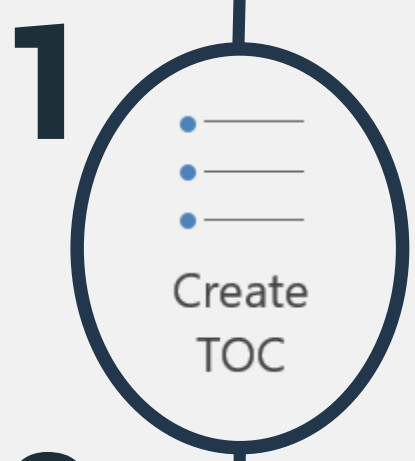
You can use optional arguments in Function procedures by using the keyword “Optional”

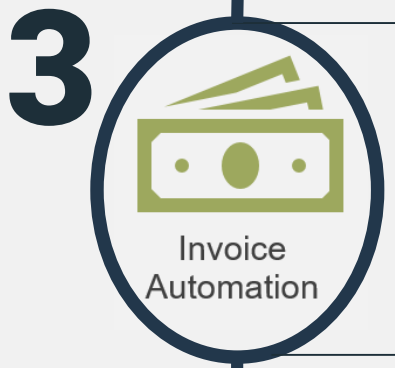
If your formula returns #Value you probably have a mistake in the VBA code To debug functions:

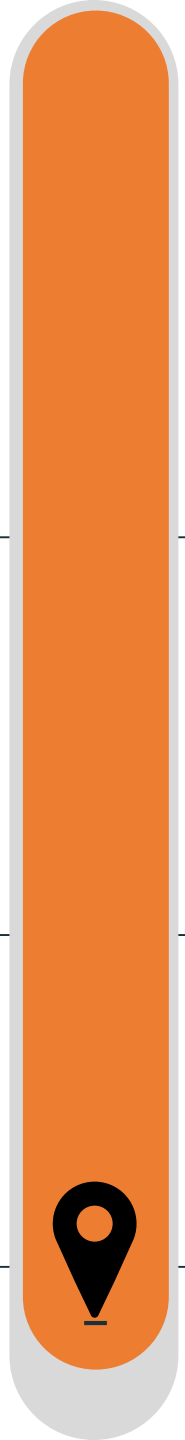
- Add a breakpoint
- Add a message box / use immediate window

Example of a Function Procedure with Optional Arguments

```
Function YearsSinceOp(myDate, Optional Txt As String) As Variant
    If IsMissing(Txt) Then
        YearsSinceOp = Year(Date) - Year(myDate)
    Else
        YearsSinceOp = Year(Date) - Year(myDate) & " " & Txt
    End If
End Function
```







Become familiar with Macros, VBE & the Object Model

How to Reference Ranges, Worksheets & Workbooks

Variables, Data Types

Working with Collections & Making Decisions

Useful VBA Functions

Debugging code, Handling Errors & Procedure Scope

Looping in VBA to Control the Flow of Code

Working with Arrays

Working with Files, Folders & Text Files

Excel Tables, Formulas & Pivot Tables

Interacting with Other Applications (Word, PowerPoint, Outlook, PDF)

Worksheet & Workbook Events

ActiveX Form Controls & VBA UserForms

Function Procedures & Working with Charts

More...

Working with Charts

- Understand the Chart Object Model
- Create Custom Charts Automatically
- Animated Charts
- Embed Charts in UserForms

VBA Basics for Charts

Embedded Chart

Application
↓
Workbook
↓
Worksheet
↓
ChartObject
↓
Chart
↓
(Chart Element)

Embedded Charts are part of the ChartObjects & the Shapes Collection

Chart Sheet

Application
↓
Workbook
↓
Chart
↓
(Chart Element)

AddChart2 Method to create an Embedded Chart and **Add2** Method for a new Chart Sheet (From Excel 2013 & above)

Examples for embedded charts

ActiveSheet.Shapes.AddChart2 , xlColumnClustered

Set myChart=ActiveSheet.Shapes.AddChart2 (, xlColumnClustered).Chart

More Learning…

If you'd like to improve your Excel knowledge, check out my other courses:

- Visually Effective Excel Dashboards
- Advanced Excel: Top Excel Tips and Formulas
- Excel Charts: Visualization Secrets for Impressive Reports
- Ultimate Excel Waterfall Chart Course

I share free content on YouTube every Thursday

Msdn.Microsoft.com Website (Visual Basic for Applications)

Power Programming with VBA (Excel 2013): By John Walkenbach. This is one of my absolute favorite VBA books.

👍 THANK YOU!

Please take a few seconds to leave a review for the course. Your support is very much appreciated.