Variables and Conditions

Created by:	Jan Nekola	Reviewed by:	Marek Prazak	Create Date:	03/18/2019

1.0 Purpose

- This lesson provides you with an overview of different types of variables as well as explanation of logical functions and loop statements.
- Don't despair if you won't understand all from just reading. We highly suggest to test out the content by trying to use it as you progress through the lesson as well as many parts of the lesson will start to make more sense as you'll start using them more often.
- The lesson should take approximately 1 hour to finish.

2.0 Definition

- **Module** - In broad terms, a module is the equivalent of a VBA container. In other words, it is where Excel actually stores the VBA code.
- **Variables** - Variables are specific values that are stored in a computer memory or storage system.
- **Logical functions** - includes If and Case Statements.
- **Loop Statements** - contains For/Next Loop, Do/While Loop and Do/Until Loop.
- **Sub** - When programming VBA you write sequences of VBA statements in procedures in modules. All procedures in VBA are named. In standard modules you will use Sub.

3.0 Content

3.1 Modules

In Excel VBA, there are four main types of modules:

- **Standard Code Modules**, which contain custom macros and functions.
- **Workbook and Sheet Code Modules**, which contain event procedures for the specific workbook or worksheets and for specific chart sheets.
- **User Forms**, which contain code for the controls of a UserForm object.
- **Class Modules**, which contain Property Let, Get, and Set procedures for Objects that you create.

3.1.1Standard Code Modules

- Also called simply **Code Modules** or just **Modules**, are where you put most of your VBA code. Your basic macros and your custom function (User Defined Functions) should be in these modules. For the novice programmer, **all your code will be in standard modules**. In addition to your basic procedures, the code modules should contain any Declare statements to external functions (Windows APIs or other DLLs), and custom Data Structures defined with the Type statement.

- Your workbook's **VBA Project can contain as many standard code modules** as you want. This makes it easy to **split** your **procedure** into **different modules** for organization and maintenance purposes. For example, you could put all your database procedures in a module named DataBase, and all your mathematical procedures in another module called Math. As long as a procedure isn't declared with the Private keyword, or the module isn't marked as private, you can call any procedure in any module from any other module without any additional steps.

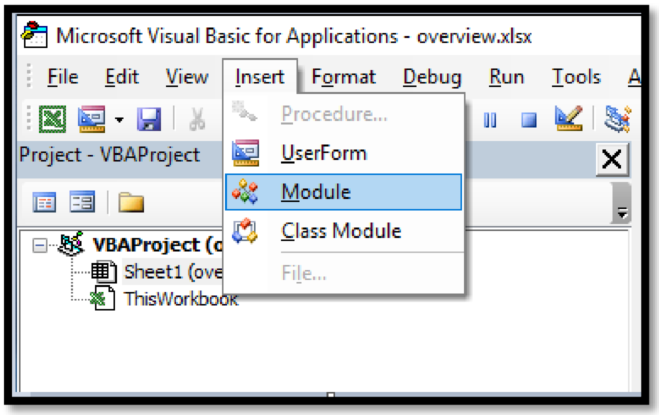


FIGURE 3.1.1 – INSERT MODULE

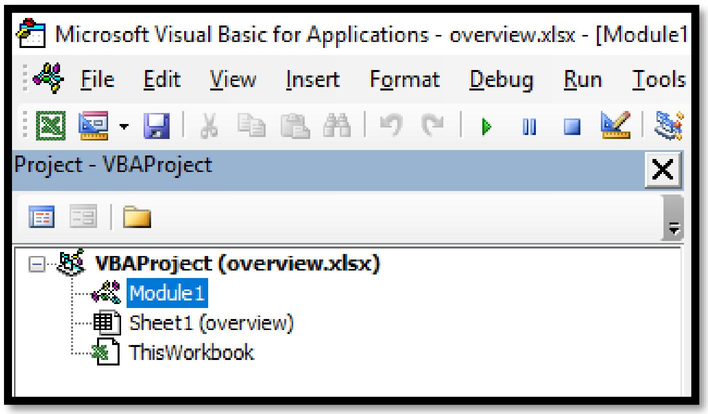
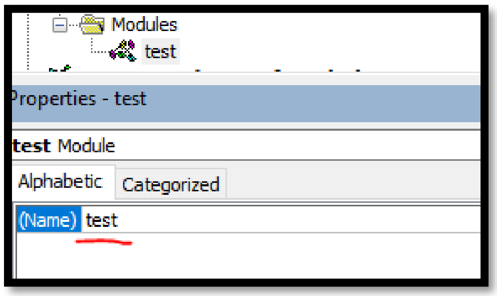

Microsoft Visual Basic for Applications - overview.xlsx - [Module1
File Edit View Insert Format Debug Run Tools
Project - VBAProject
VBAProject (overview.xlsx)
Module1
Sheet1 (overview)
ThisWorkbook

FIGURE 3.1 .2– NEW MODULE

Note: *It is good practice to name your modules as it helps with organization.*



3.1.2 Workbook And Sheet Modules

- Are modules tied directly to the Workbook object and to each Sheet object. The module for the workbook is called *ThisWorkbook*, and each Sheet module has the same name as the sheet that it is part of. These modules should contain the event procedures for the object, and that's all. If you put **the event procedures in a standard code module**, Excel won't find them, so they **won't be executed**. And if you put ordinary procedures in a workbook or sheet module, you won't be able to call them without fully qualifying the reference.

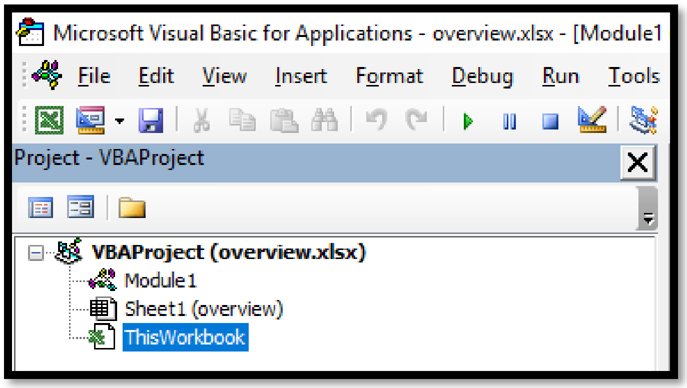


Figure 3.1 .3– ThisWorkbook

3.1.3 User Form Modules

- User Form Modules are part of the UserForm object, and contain the event procedures for the controls on that form. For example, the Click event for a command button on a UserForm is stored in that UserForm's code module. Like workbook and sheet modules, you should put only event procedures for the UserForm controls in this module.
- User Form Allows you to create forms, input forms that can be quiet useful if you want to limit the user in what type of data will they input.

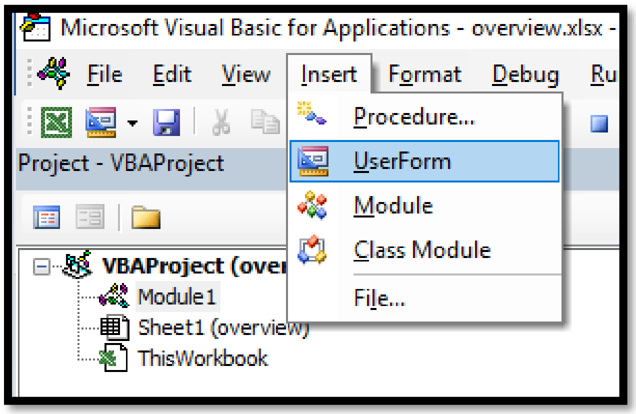


FIGURE 3.1.4 – INSERT USERFORM

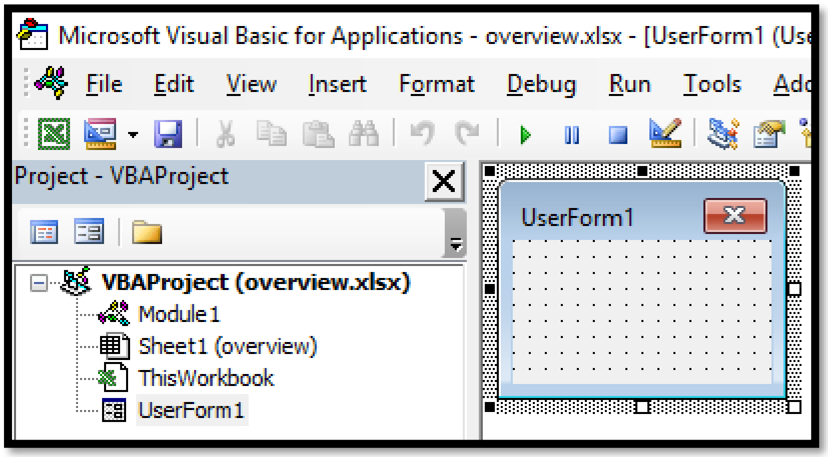
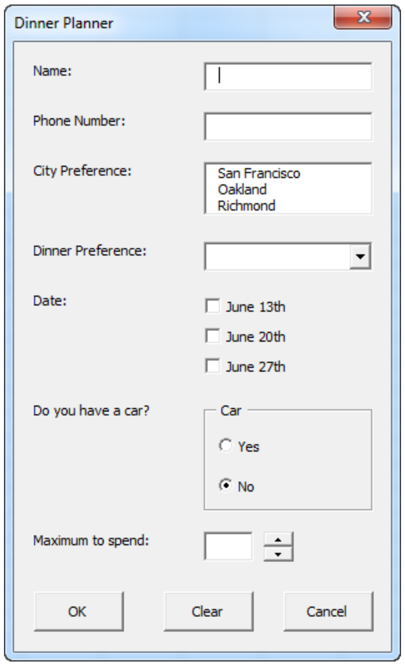


FIGURE 3.1.5 – NEW USERFORM



3.1.4 Class Modules

- Class Modules are stored in the Class Modules folder and allow us to write macros to create objects, properties, and methods that are standardly not available in excel. These custom objects or collections can be later on used in your programs. Don't worry if you don't know what objects, properties or methods mean. You're going to learn about these later in lesson. For now only remember that class modules exist and what are they, as during the course of this training you'll not be needing them.

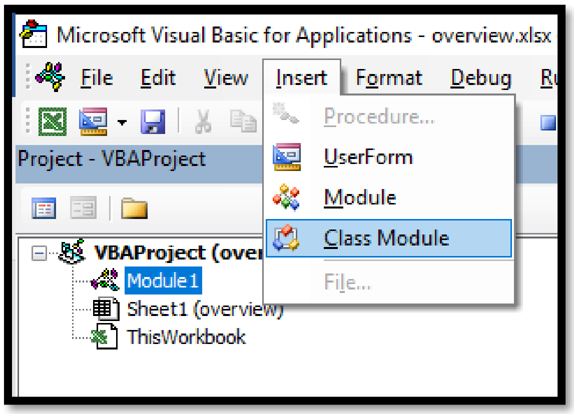


Figure 3.1.6 – Insert Class

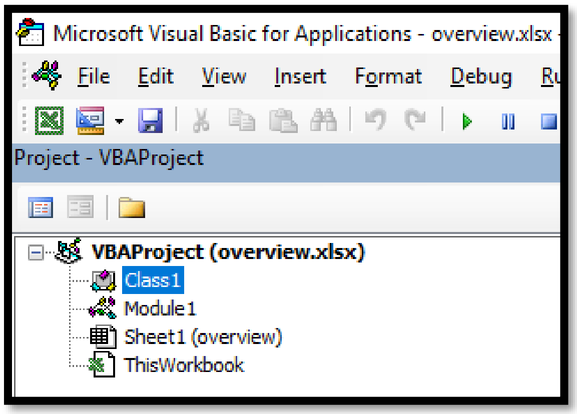
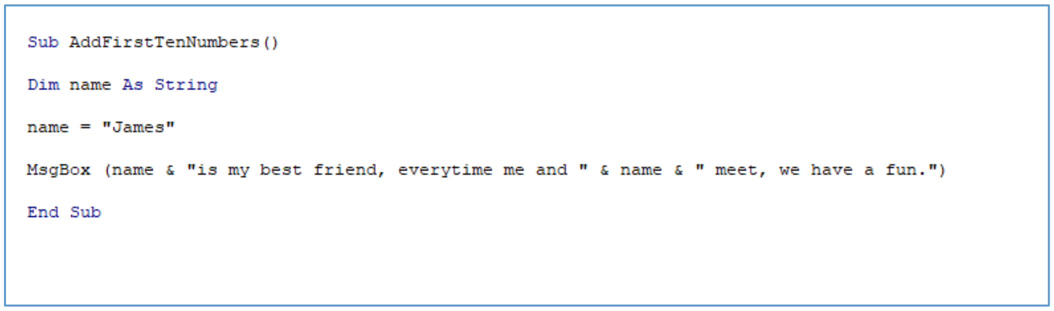


FIGURE 3.1.7– NEW CLASS

- When we double-click or right-click> View Code (keyboard shortcut: F7) on any of these objects in the Project Explorer Window, the code window opens on the right side of the VB Editor. The code window looks the same for each of the objects. It is just a big blank canvas where we can type code.

3.2 The Variables

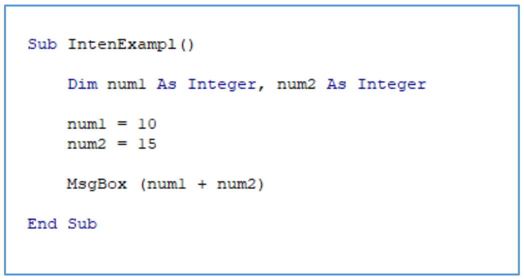
- When you write code in VBA, you will need variables that you can use to hold a value. The benefit of using a variable is that you can change the value of the variable within the code and continue to use it in the code.
- Below you'll see few of the most used Variables.
- **String**



```
Sub AddFirstTenNumbers()

Dim name As String

name = "James"

MsgBox (name & "is my best friend, everytime me and " & name & " meet, we have a fun.")

End Sub
```

  - In example, above we can see a code that shows the user a pop up message *"James is my best friend, everytime me and James meet, we have a fun."* In this syntax we've saved the name james into **string**(text) **variable** called **name** and each time we called upon it gave us the **value** we've **stored** in it. Though this case is simplistic imagine having 30 pages of contract and each page references the name or complex calculation that will have its value changed each time it has been used.

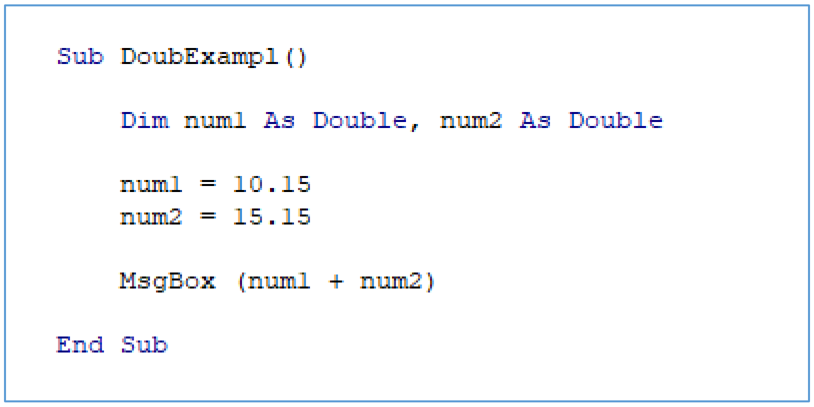
- **Integer**



```
Sub IntenExampl()

    Dim num1 As Integer, num2 As Integer

    num1 = 10
    num2 = 15

    MsgBox (num1 + num2)

End Sub
```

  - In example, above we can see a code that adds up full number 10 and full number 15. If you tried to store **text, decimal number, or more than +/-32,767** you would get an **error**.

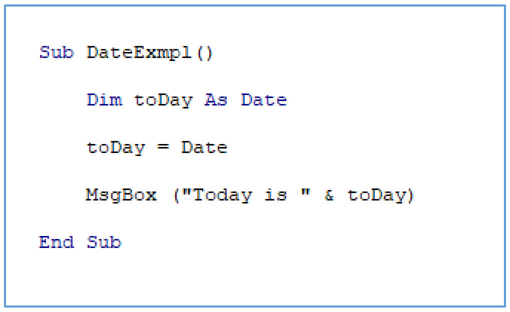
- **Double**



```
Sub DoubExampl()

    Dim num1 As Double, num2 As Double

    num1 = 10.15
    num2 = 15.15

    MsgBox (num1 + num2)

End Sub
```

- In example, above we can see a code that adds up decimal number 10.15 and decimal number 15.15. If you tried to store **text** you would get an **error**.

- **Date**



```
Sub DateExmpl()

    Dim toDay As Date

    toDay = Date

    MsgBox ("Today is " & toDay)

End Sub
```

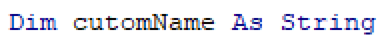
- In example, above we can see a code that stores todays date in **date variable**. If you tried to store **text, numbers, boolen** you would get an **error**.

To name the variable in VBA, you need to follow the below rules.

- It must be less than 255 characters.
- No spacing is allowed.
- It must not begin with a number.
- Period is not permitted.

As you might have already noticed, to declare variable you have to follow below syntax.

- Dim, name of your choice, as, type of variable



```
Dim cutomName As String
```

The examples of valid and invalid variable names.

Valid Names	Invalid Names
My_Watch	My.Watch
NewCar1	1_NewCar (not begin with number)
EmployeeID	Employee ID (Space is not allowed)

TABLE 3.2.1– RULES TO NAME THE VARIABLE

Note: *As a best practice remember to always write **Option Explicit** before your code. This functionality will notify you if you've forgotten to declare any variable.*

3.2.1TYPE OF VARIABLES

- You might wonder **why even bother** with so many different types if you can just use variant for all. Main reason is that your **code will run more smoothly** as it will take **less memory** of your computer, certain syntaxes only **run with specific variables** and it will create **better readability**.
- To make the best use of variables, it's a good practice to specify the data type of the variable. The data type you assign to a variable will be dependent on the type of data you want that variable to hold.

Data Type	Bytes Used	Range of Values
Byte	1 byte	0 to 255
Boolean	2 bytes	True or False
Integer	2 bytes	-32,768 to 32,767
Long (long integer)	4 bytes	-2,147,483,648 to 2,147,483,647
Single	4 bytes	-3.402823E38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values
Double	8 bytes	-1.79769313486231E308 to-4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values

Currency	8 bytes	-922,337,203,685,477.5808 to 922,337,203,685,477.5807
Decimal	14 bytes	+/- 79,228,162,514,264,337,593,543,950,335 with no decimal point;+/- 7.9228162514264337593543950335 with 28 places to the right of the decimal
Date	8 bytes	January 1, 100 to December 31, 9999
Object	4 bytes	Any Object reference
String (variable-length)	10 bytes	0 to approximately 2 billion
String (fixed-length)	Length of string	1 to approximately 65,400
Variant (with numbers)	16 bytes	Any numeric value up to the range of a Double
Variant (with characters)	22 bytes	Same range as for variable-length String
User-defined	Varies	The range of each element is the same as the range of its data type.

TABLE 3.2.2– DATA TYPE OF VARIABLES

- **Workbook/Worksheet variable -** The correct way to program VBA is to explicitly assign which object you want to work with. In case of a Workbook this can be expressed in several ways.
  - In this example we've declared **wb** as workbook, then we've **assigned** the **workbook** containing the macro to this variable and we've declared **ws** as worksheet and **assigned** active **sheet** (currently selected) **from wb**. Code itself will then **copy ws.**

```
Sub CopyWs()

    Dim wb As Workbook: Set wb = ThisWorkbook
    Dim ws As Worksheet: Set ws = wb.ActiveSheet

    ws.Copy

End Sub
```

  - **ThisWorkbook** - If your macro is only intended to work on the excel file that contains it.

```
Dim wb As Workbook: Set wb = ThisWorkbook
```

  - **ActiveWorkbook -** The Active Workbook is the workbook in the active window (the window on top)

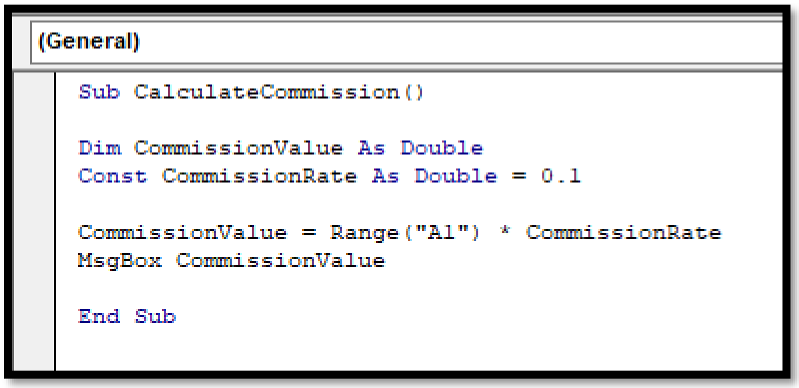
```
Dim wb As Workbook: Set wb = ActiveWorkbook
```

- **Workbooks -** The name of the active workbook appears in the Excel title bar. If a workbook is new and not yet saved, it will be assigned a temporary name.

```
Set wb = Workbooks("Book1")
```

3.2.2 DECLARING CONSTANTS

- While variables can change during the code execution, if you want to have fixed values, you can use constants.
- A constant allows you to assign a value to a named string that you can use in your code.
- While not often there might be cases where you don't want the value to change in which case constant is the ideal solution.



```
Sub CalculateCommission()

Dim CommissionValue As Double
Const CommissionRate As Double = 0.1

CommissionValue = Range("A1") * CommissionRate
MsgBox CommissionValue

End Sub
```

FIGURE 3.2.3– CONSTANTS

3.3 Logical Functions

- Before jumping to an explanation of the logical functions, it would be beneficial to review logical operators as they play important role in the logical functions.

3.3.1 Logical Operators

- VBA includes several built-in operators and functions, which can be used for building expressions or performing tasks in your VBA code.
- Logical operators return a logical (True or False) result. The main logical operators are listed in the table below.

Operator	**Action**
And	If the expression A **AND** B is true(expected result) at the same time then the syntax will return Boolean value TRUE, if either A or B is not true the syntax will return value FALSE.
Or	If either the expression A **OR** B is true(expected result) then the syntax will return Boolean value TRUE, if neither A **OR** B is not true the syntax will return value FALSE.
Not	Negates an evaluation. The expression A is not true(correct) then the syntax returns TRUE. Else if A is true(correct) then syntax returns False if A is true.

TABLE 3.3.1– LOGICAL OPERATORS

- To give you little bit more clarification on above, look at following example.
  A = 1
  B = 2
  Question: Is **A = 2** ?? **AND** Is **B = 2**??
  Answer: **False AND True**
  Conclusion: Result of this exercise is **FALSE** as **A is not true**

3.3.2 If Then Else Statement

- **If Then Else Statement** allows you the check for a condition and perform an action accordingly.

- **The syntax of IF:**

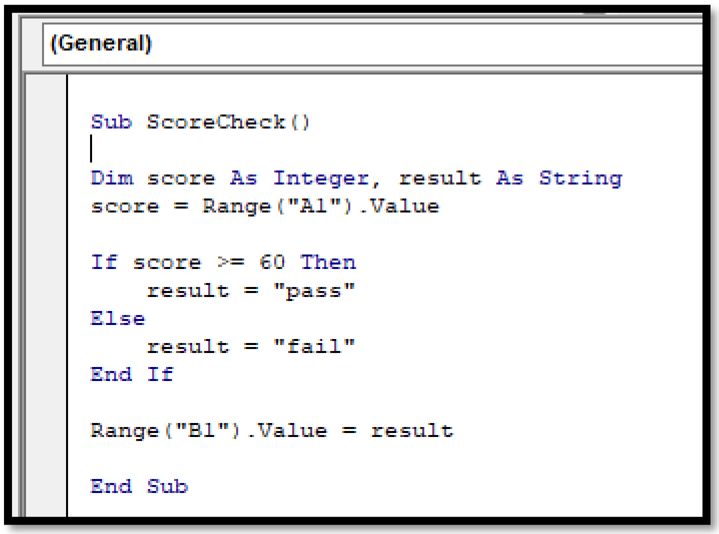
```
IF [ condition is true] Then
        [statements]
End If
```

Note: *To make your code more readable it is good practice to indent the line between the If Then and End If statements.*

- Now let's see **the syntax** of **IF Else statement** in VBA:

```
IF [ condition] Then
        [statements if condition is true]
Else
        [statements if condition is false]
End If
```

Example below checks the value in the cell A1 and if the cell value is greater or equal to 60, then function returns "pass" the cell B1. If the value is lower, the function will write "fail" in cell B1. This



```
Sub ScoreCheck()

Dim score As Integer, result As String
score = Range("A1").Value

If score >= 60 Then
    result = "pass"
Else
    result = "fail"
End If

Range("B1").Value = result

End Sub
```

FIGURE 3.3.1– IF STATEMENT

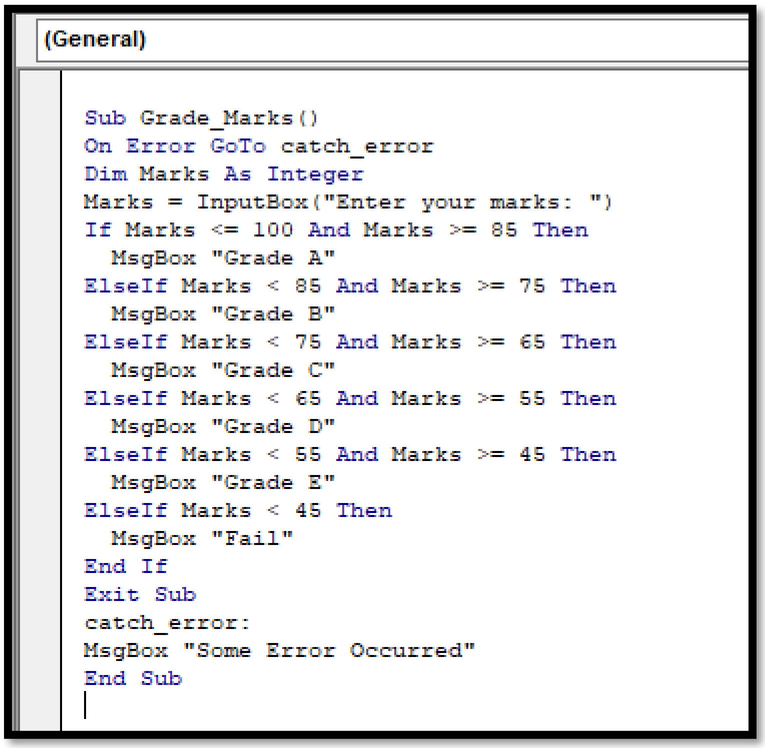
3.3.3 USING IF STATEMENT WITH LOGICAL OPERATORS

- Logical operators make it possible for you to check multiple condition at a time, inside a single IF statement. There are many logical operators in VBA like: **And**, **Or**, **Not**, **AndAlso**, **OrElse**, and **Xor** but in most cases we only deal with the first three.
- Logical operators:

A **>** B	A is **larger than** B
A **<** B	A is **smaller than** B
A = B	A is **equal** to B
A **<>** B	A does **not equal** to B
A **>=** B	A is **larger or equal** to B
A **<=** B	A is **smaller or equal** to B

- In the below table we have a Grade Table. Our task is to write a program that accepts Marks from user and displays the corresponding Grade.

Marks Range	Grade
>=85	Grade A
< 85 to >= 75	Grade B
< 75 to >= 65	Grade C
< 65 to >= 55	Grade D
< 55 to >= 45	Grade E
< 45	Fail



```
Sub Grade_Marks()
On Error GoTo catch_error
Dim Marks As Integer
Marks = InputBox("Enter your marks: ")
If Marks <= 100 And Marks >= 85 Then
  MsgBox "Grade A"
ElseIf Marks < 85 And Marks >= 75 Then
  MsgBox "Grade B"
ElseIf Marks < 75 And Marks >= 65 Then
  MsgBox "Grade C"
ElseIf Marks < 65 And Marks >= 55 Then
  MsgBox "Grade D"
ElseIf Marks < 55 And Marks >= 45 Then
  MsgBox "Grade E"
ElseIf Marks < 45 Then
  MsgBox "Fail"
End If
Exit Sub
catch_error:
MsgBox "Some Error Occurred"
End Sub
```

FIGURE 3.3.2– IF ELSE STATEMENT

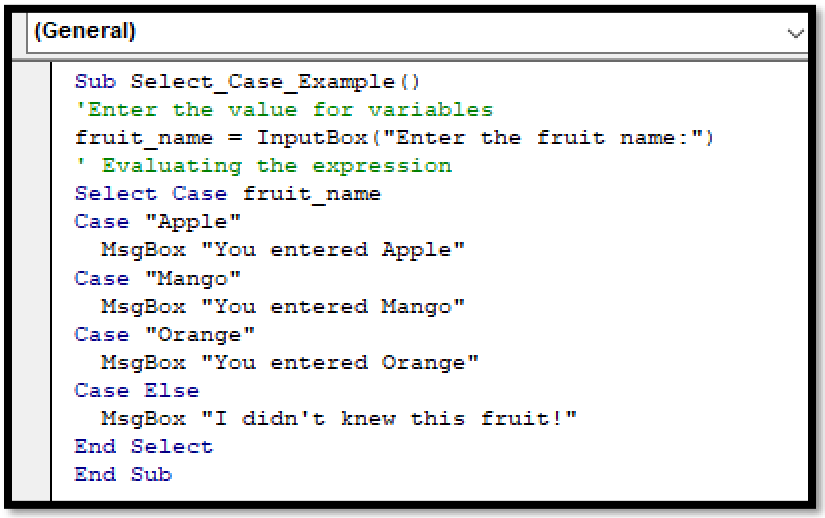
- As the best practice, it is always nice to use **Select Case** statements instead of writing multiple **ELSEIF** statements (just like we have seen in above example). Select Case statements execute faster and look cleaner than IF THEN ELSE.

3.3.4SELECT CASE STATEMENT

- **Select Case** can be used instead of complex Excel **Nested If statements**. This makes the VBA code faster to execute and easier to understand.
- Statement checks a variable or an expression for different cases (values). If anyone of the case becomes true, then only that case is executed and the program ignores all other cases.

```
'The Syntax is as under:

Select Case Condition
    Case value_1
        'Code to Execute When Condition = value_1
    Case value_2
        'Code to Execute When Condition = value_2
    Case value_3
        'Code to Execute When Condition = value_3
    Case Else
        'Code to Execute When all the other cases are False
End Select
```

- Here, **Condition** refers to the variable or the expression that is to be tested and based on which anyone of the code segments will be executed. **value_1**, **value_2** and **value_3** are the possible outcomes of the **Condition**. Whenever anyone of these values matches the **Condition** then its corresponding **Case** block will execute. **Else** is a kind of default case value, which will only execute when all the above Case statements result into **False**. **Else** case is optional but generally it is considered a good practice to use it.
- **Example** of Case Statement to check Text Strings



```
Sub Select_Case_Example()
'Enter the value for variables
fruit_name = InputBox("Enter the fruit name:")
' Evaluating the expression
Select Case fruit_name
Case "Apple"
  MsgBox "You entered Apple"
Case "Mango"
  MsgBox "You entered Mango"
Case "Orange"
  MsgBox "You entered Orange"
Case Else
  MsgBox "I didn't knew this fruit!"
End Select
End Sub
```

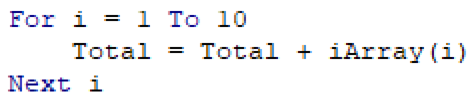
FIGURE 3.3.2– CASE STATEMENT

3.4 LOOP STATEMENTS

- **Looping** is one of the **most powerful** programming techniques. A loop in Excel VBA enables you to loop through a range of cells with just a few codes lines. You might encounter such technique for example in the instance where you work with a list of employees and you'd like to loop through all the date for each employee. Without the Loop statement you'd have to write identical code for each employee and the macro would get very long, compared to that the loop will only replace the value with each new employee.

3.4.1The For … Next Loop

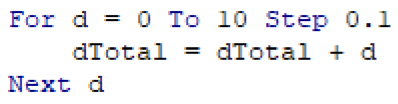
- The **For … Next loop** uses a variable, which cycles through a series of values within a specified range. The VBA code inside the loop is then executed for each value. This is best explained by way of a simple example:



```
For i = 1 To 10
    Total = Total + iArray(i)
Next i
```

- The above, **For ... Next loop** sets the variable **i** to have the values **1, 2, 3, ..., 10** and **for each of these values**, runs through the **VBA code inside the loop**. Therefore, in the above example, the loop **adds** each of the members of the array iArray **to the** variable, **Total**.

- In the above example, no **step size** is specified, so the loop uses **the default step size of 1**, when looping from 1 to 10. However, you may sometimes want to step through a loop using different sized steps. This can be done using the Step keyword, as shown in the following simple example.



```
For d = 0 To 10 Step 0.1
    dTotal = dTotal + d
Next d
```

- In the above **For loop**, because the **step size** is **specified as 0.1**, the value of the variable **d** is set to the values **0.0, 0.1, 0.2, 0.3, ..., 9.9, 10.0** for each execution of the VBA code inside the loop.

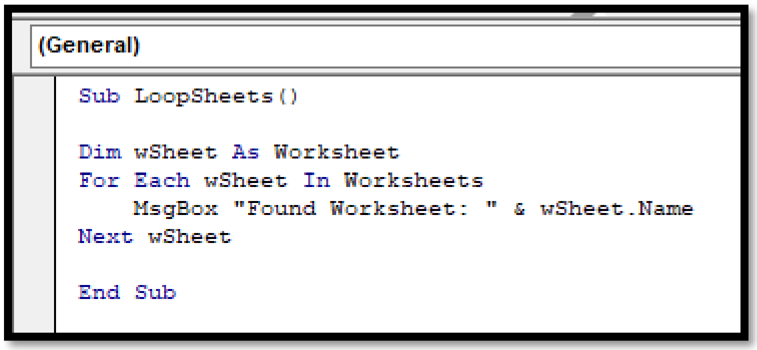
- You can also use **negative step sizes** in the VBA For loop, as is illustrated below:

```
For i = 10 To 1 Step -1
    iArray(i) = i
Next i
```

- In this example, the **step size** is specified as **-1**, and so the loop sets the variable **i** to have the values **10, 9, 8, ..., 1**.

3.4.2 The For Each Loop

- **The For Each loop** is **similar** to the **For ... Next loop** but, instead of running through a set of values for a variable, the For Each loop **runs through every object** within a set of objects. For example, the following code shows the For Each loop used to list every Worksheet in the current Excel Workbook:



```
Sub LoopSheets()

Dim wSheet As Worksheet
For Each wSheet In Worksheets
    MsgBox "Found Worksheet: " & wSheet.Name
Next wSheet

End Sub
```

Figure 3.3.3– For Each Loop

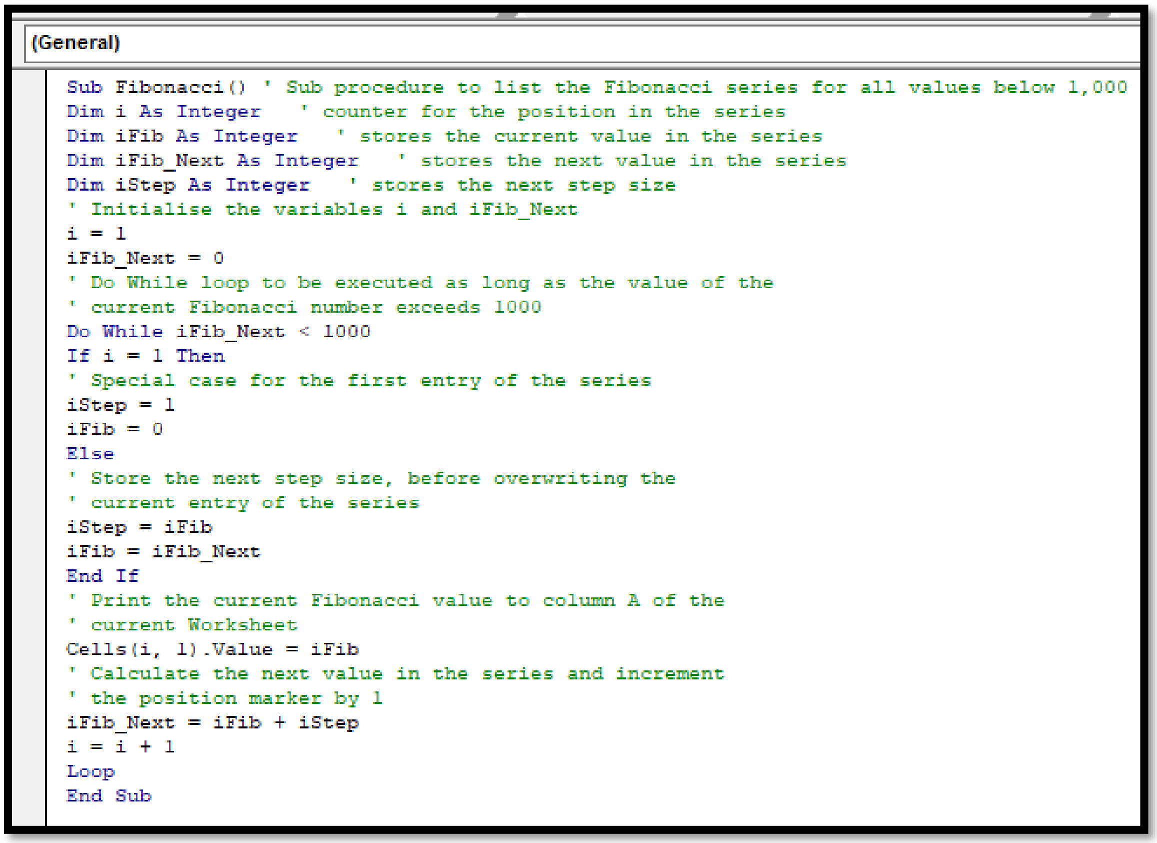
3.4.3 The Exit For Statement

- If, **you want to exit** a 'For' **Loop early**, you can use the **Exit For statement**. This statement causes VBA to jump out of the loop and continue with the next line of code outside of the loop. For example, when searching for a particular value in an array, you could use a loop to check each entry of the array. However, once you have found the value you are looking for, there is no need to continue searching, so you exit the loop early.
- The Exit For statement is illustrated in the following example, which **loops** through **100 array** entries, comparing each to the value **'dVal'**. The loop is **exited early if** dVal is **found** in the array:

```
For i = 1 To 100
    If dValues(i) = dVal Then
        IndexVal = i
    Exit For
    End If
Next i
```

3.4.4 THE DO WHILE LOOP

- The Do While loop **repeatedly executes** a section of code **while** a specified **condition continues** to evaluate to True. This is shown in the following Sub procedure, where a **Do While** loop is used to **print out** all values of the Fibonacci Sequence **until** the current **value** is **greater than 1,000**:

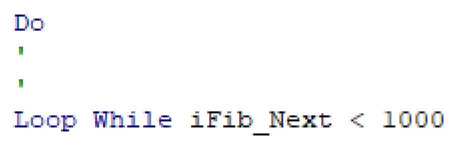

(General)

```
Sub Fibonacci() ' Sub procedure to list the Fibonacci series for all values below 1,000
Dim i As Integer   ' counter for the position in the series
Dim iFib As Integer   ' stores the current value in the series
Dim iFib_Next As Integer   ' stores the next value in the series
Dim iStep As Integer   ' stores the next step size
' Initialise the variables i and iFib_Next
i = 1
iFib_Next = 0
' Do While loop to be executed as long as the value of the
' current Fibonacci number exceeds 1000
Do While iFib_Next < 1000
If i = 1 Then
' Special case for the first entry of the series
iStep = 1
iFib = 0
Else
' Store the next step size, before overwriting the
' current entry of the series
iStep = iFib
iFib = iFib_Next
End If
' Print the current Fibonacci value to column A of the
' current Worksheet
Cells(i, 1).Value = iFib
' Calculate the next value in the series and increment
' the position marker by 1
iFib_Next = iFib + iStep
i = i + 1
Loop
End Sub
```

FIGURE 3.3.4–DO WHILE

- It can be seen that, in the above example, the condition **iFib_Next < 1000** is tested at the start of the loop. Therefore, **if** the **first** value **of iFib_Next** were **greater** than **1,000**, the loop would **not be executed** at all.

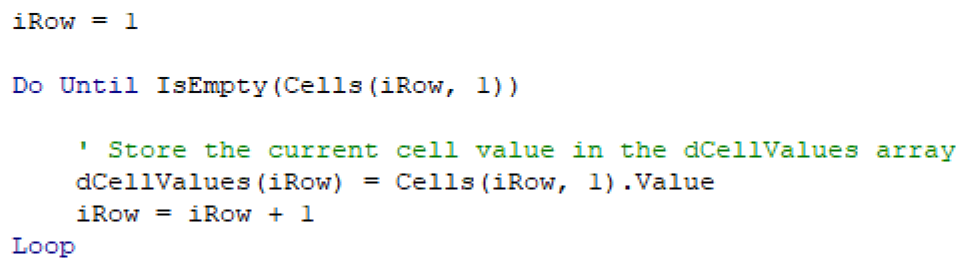
- Another way that you can implement the **Do While loop** is to place the **condition at the end of the loop** instead of at the beginning. This causes the loop to be executed at least once, regardless of whether or not the condition initially evaluates to True.



```
Do
'
'
Loop While iFib_Next < 1000
```

3.4.5The Do Until Loop

- The **Do Until loop** is very **similar** to the **Do While loop**. The loop repeatedly executes a section of code until a specified condition evaluates to True. This is shown in the following sub procedure, where a **Do Until loop** is used to extract the values from all cells in Column A of a Worksheet, **until it encounters an empty cell**:



```
iRow = 1

Do Until IsEmpty(Cells(iRow, 1))

    ' Store the current cell value in the dCellValues array
    dCellValues(iRow) = Cells(iRow, 1).Value
    iRow = iRow + 1
Loop
```

- In the above example, as the condition **IsEmpty(Cells(iRow, 1))** is **at the start** of the Do Until loop, the loop will **only be entered** if the **first cell** encountered is **non-blank**.

- However, as illustrated in the **Do While loop**, you may sometimes **want to enter the loop at least once**, regardless of the initial condition. In this case, the **condition** can be placed **at the end** of the loop, as follows:

```
Do
'
'
'
Loop Until IsEmpty(Cells(iRow, 1))
```

4.0 Closing Exercise

Following example is unnecessary complicated for such simple instruction, but it is designed this way for the sake of this exercise.

- Use source excel

1) create 5 modules with following names:
    i. Main
    ii. OverviewTable
    iii. HighlighSalary
    iv. ProtectSheets
    v. AddCount
2) Main - Use this subroutine to call OverviewTable, HighlighSalary and ProtectSheets
3) OverviewTable – macro will create new sheet named “Overview” and stores unique states. The header should contain “state” and “Number of Employees”. Call subroutine “AddCount from OverviewTable that sums up how many employees occur in the particular states. (Use COUNTIF formula in Do Until IsEmpty loop).
4) HighlighSalary – In the sheet “Sheet1” mark the salary, which is lower than or equal to 29 000 with color, RGB(200, 255, 200). The salary which is higher than 29 000 color yellow, RGB(255, 255, 0). Use loop function with if else statement.
5) ProtectSheets – Create loop statement that protects each sheet with password. Password=HRSTransformation.
6) Extra Task. Protect the file during opening with password =HRSTransformation.

5.0 Additional Knowledge

- http://www.cpearson.com/excel/codemods.htm
- https://www.guru99.com/vba-data-types-variables-constant.html
- https://trumpexcel.com/vba-data-types-variables-constants/
- https://www.techonthenet.com/excel/formulas/case.php
- https://www.excelfunctions.net/vba-loops.html
- https://www.excel-easy.com/vba/loop.html
- https://corporatefinanceinstitute.com/resources/excel/study/vba-do-loop/
- https://www.excelcampus.com/vba/code-modules-event-procedures/
- https://www.excelfunctions.net/vba-operators-and-functions.html
- https://www.techonthenet.com/excel/formulas/case.php
- https://www.excel-easy.com/vba/examples/select-case.html
- https://excelmacromastery.com/vba-if/
- https://www.excel-easy.com/vba/if-then-statement.html
- https://www.exceltrick.com/formulas_macros/vba-if-statement/
- https://www.exceltrick.com/formulas_macros/vba-select-case-statement/