# Introduction

## What is VBA?

- VBA stands for 'Visual Basic for Applications'
- VBA is an implementation of Microsoft's event-driven programming language Visual Basic 6 and VBA continues to be one of the most popular tool for automating repetitive tasks using Microsoft Office suite of programs
- VBA is built into most Microsoft Office applications including Microsoft Excel, Access and Outlook and the language was designed to be intuitive for users who are familiar with the Microsoft Office applications. For example, in Excel VBA, a line of code **Range("A1").AddComment("Hello")** instructs Excel to add a comment "Hello" in a cell "A1"

## Who is this course for?

This course assumes no prior knowledge of VBA and this course is for those of you, where:
- You may be already familiar with Microsoft Excel and may have recorded few macros previously but want to learn to write VBA macros from scratch
- You want to learn the fundamentals of VBA programming language and gain ability to apply VBA to real life problems
- You have tedious and repetitive tasks that you'd like to automate
- You would like to make your spreadsheets into more user friendly with user interactions such as buttons, message boxes, etc

# Introduction

| Course Objectives | By the end of this course, you will:<br>○ Know your way around the VBA development environment (Developer tab, VBA editor)<br>○ Know how to write VBA Sub-routines and Functions from scratch<br>○ Gain firm understanding of VBA programming language syntax such as IF, For Loop, Do Loop, Arrays by going through 45+ real life VBA example/exercise codes in the Examples workbook<br>○ Know how to automate tedious and repetitive tasks and processes in Excel<br>○ Know how to make your spreadsheets more interactive by using message box, input box and buttons |
|---|---|

# Curriculum

| | |
|---|---|
| **Getting Started** | o    Showing the 'Developer' tab<br>o    Finding your way around the VBA editor environment<br>o    Your first VBA program<br>o    Looking closer at your first VBA program<br>      o    Modules<br>      o    Sub-routines<br>      o    Comments<br>      o    Variables and declarations<br>      o    Operators<br>      o    Variable Scope |
| **Learn by doing: VBA fundamentals with example programs** | o    Some useful built-in functions used throughout this course<br>o    Mastering Excel 'objects' by:<br>      o    Mastering Cell Range<br>      o    Mastering Worksheet<br>      o    Mastering Workbook<br>o    More sophisticated programming with:<br>      o    Functions<br>      o    IF statements<br>      o    For loops and Do loops<br>      o    Arrays<br>o    Debugging VBA programs<br>o    Error handling: On Error GoTo statement |
| **User Interaction** | o    Message boxes and Input boxes<br>o    Programming a Button |

# Getting Started – Developer Tab

## How to show the 'Developer' tab

The **Developer tab** has necessary tools, where you can write macros, run macros that you previously recorded, add some user interactivity to your spreadsheet such as buttons, checkboxes, etc:
1. Go to **File > Options**
2. Select **Customize** Ribbon
3. Under the **Main Tabs,** select the **Developer** checkbox and then choose OK
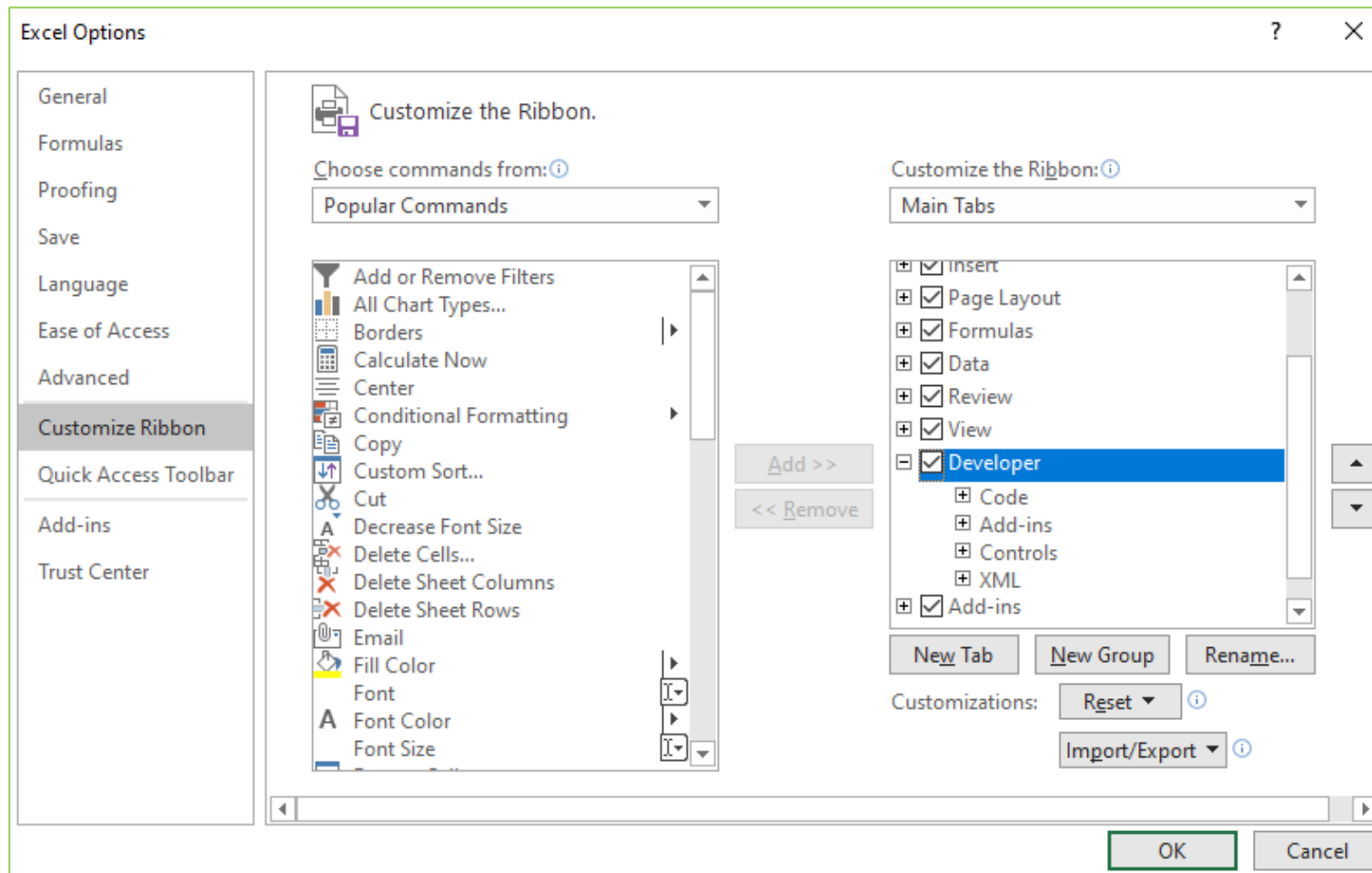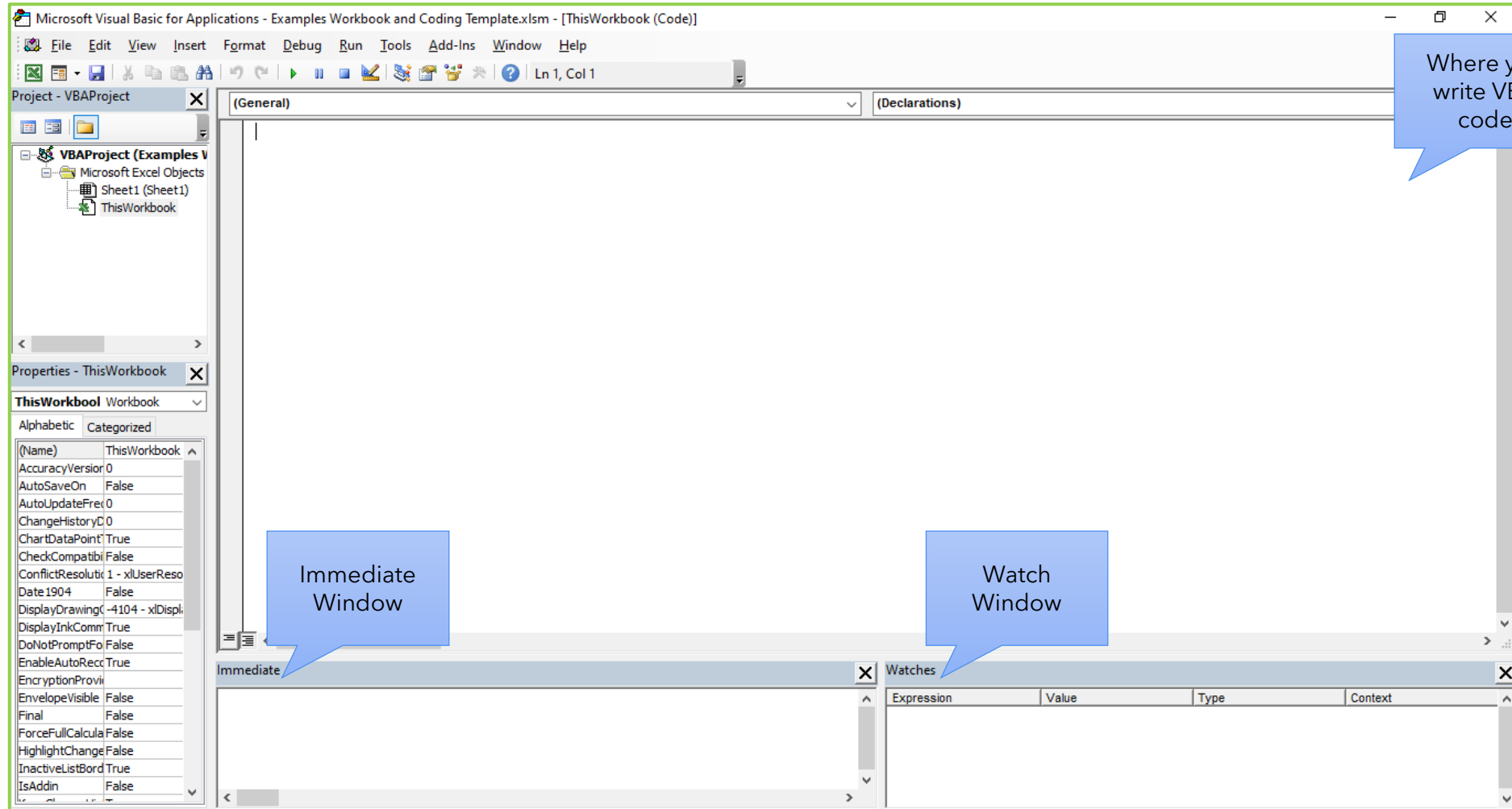
# Getting Started – VBA editor

## How to show the VBA editor

The **VBA editor** is where you write your VBA program or view previously recorded macros. You can also design your own User forms which can enhance user experience for your spreadsheet application. To show the VBA editor:

- On the **Developer tab,** click **Visual Basic**



Project Explorer

Properties Window

Where you write VBA code

Immediate Window

Watch Window

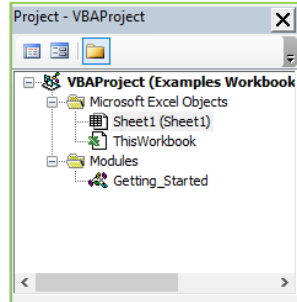# Getting Started – Your First VBA Program

**Run your first VBA program**

1. **Open** the Example workbook
2. Show **'Developer' tab** if you haven't done so per the previous slides
3. Show **VBA editor**
4. Double click on the 'Getting Started' module – **Do not worry about what the program does yet!**
5. Click on the ▶ button to run the program
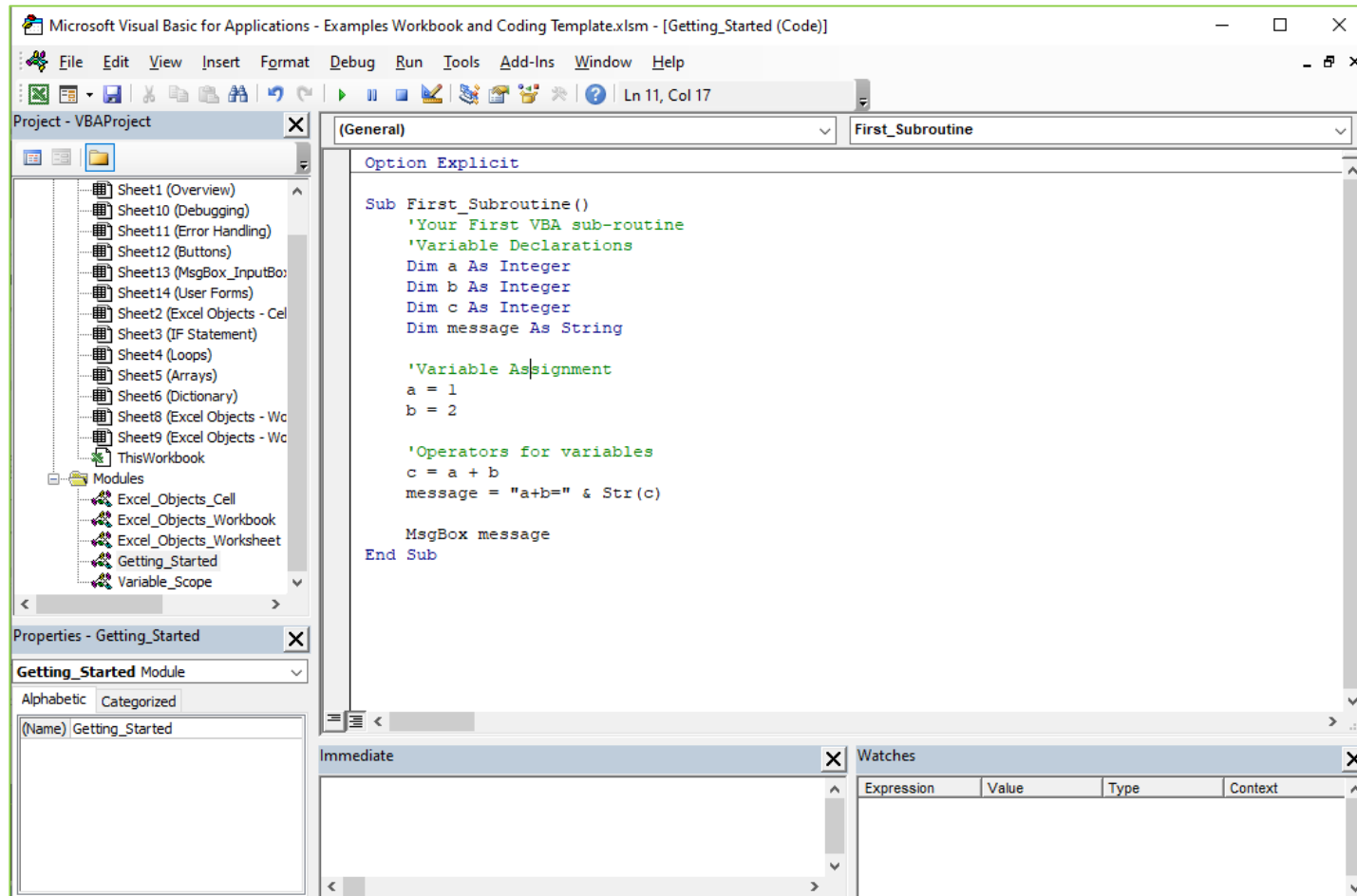
# Getting Started – Looking Closer At Your First VBA Program

**Module**

- In an Excel VBA project for a given Excel macro-enabled workbook (*.xlsm), you can either write your VBA code in "placeholders" within the 'ThisWorkbook' object, a Worksheet object or a Module.



- You can think of Modules as placeholders, where you can write VBA code, which is accessible by any part of the Excel application the module belongs to. In the example VBA program we have created, a module called 'Getting_Started' has been added to the workbook by going to VBA Editor > Project window > Right click on the 'VBAProject' > Insert > Module.



Module 'Getting_Started'
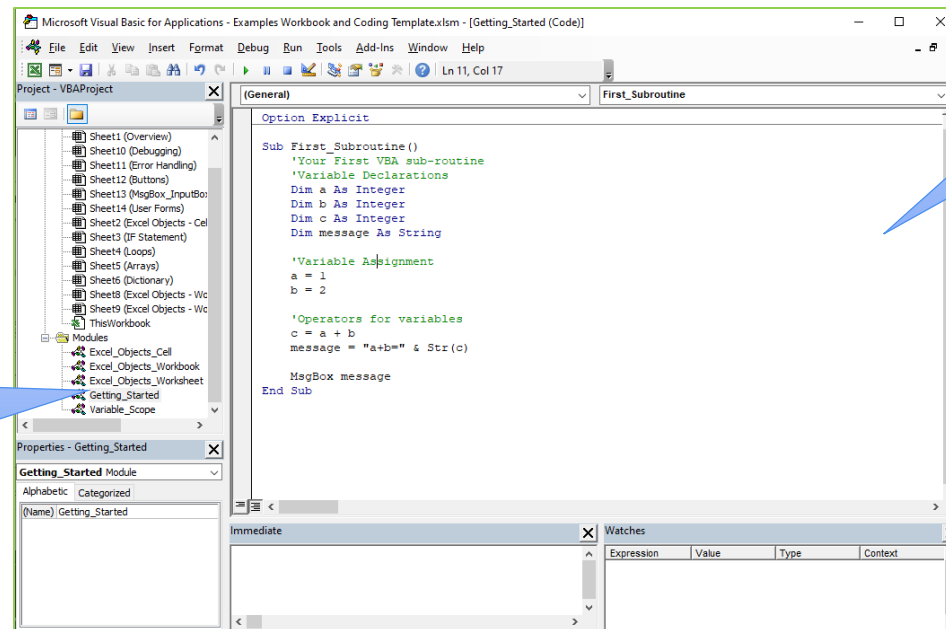
VBA code written in the Module

# Getting Started – Looking Closer At Your First VBA Program

**Sub-routines**

o   A **Sub-routine** is a piece of code that performs a specific task and does NOT return a result. **Sub-routines** are used to break down large pieces code into small manageable parts, which can be called repeatedly.

```vba
Option Explicit

Sub First_Subroutine()
    'Your First VBA sub-routine
    'Variable Declarations
    Dim a As Integer
    Dim b As Integer
    Dim c As Integer
    Dim message As String

    'Variable Assignment
    a = 1
    b = 2

    'Operators for variables
    c = a + b
    message = "a+b=" & Str(c)

    MsgBox message
End Sub
```

Beginning of a Sub-routine

Code that gets run when the Sub-routine gets executed

End of a Sub-routine

# Getting Started – Looking Closer At Your First VBA Program

**Comments**

o   Here is what you see in the first section of the subroutine 'First_Subroutine()' . The apostrophe character/symbol **'** at the beginning signify a single-line comment in excel VBA.

```
'Your First VBA sub-routine
'Variable Declarations
```

o   Comments are written in plain English to provide an explanation or comment on the code. All comments written after an apostrophe symbol (per above) are completely ignored by Excel when the VBA program is being executed. Writing succinct and well explained comments on what a specific piece of code does is a best practice shared among many programmers, as this helps one to understand how the code works, especially as one begins to write large complex programs.

# Getting Started – Looking Closer At Your First VBA Program

## Variables and Declarations

- Variables are used in VBA to store values for later use in the program
- It is a best practice to declare variables before usage within the program and declaration of a variable involves specifying its name and characteristics (e.g. data type, scope of the variable). Typing the 'Option Explicit' command at the very beginning of every module enforces declaration of variables.

```
Option Explicit
```

- In our first example VBA program, 5 variables have been declared per below. The basic syntax for declaring a variable is **Dim** <<*variable name goes here…*>> **As** <<*variable data type goes here…*>>

```
Dim a As Integer
Dim b As Integer
Dim c As Integer
Dim message As String
```

- There are many variable data types and I will not be listing them all here but common examples, which will be used in this course include the following.
    - String: text variable e.g. "hello world"
    - Integer: an integer (whole number) variable with a range from -32,768 to 32,767
    - Long: an integer variable with a range from -2,147,483,648 to 2,147,483,647
    - Double: a decimal numerical variable
    - Boolean: a variable holding True or False values
    - Excel objects such as Range, Worksheet, Workbook
    - Arrays

- Variable assignments means storing some value in the variable declared. In our example code, below are examples of storing numerical values in the variables a and b.

```
'Variable Assignment
a = 1
b = 2
```

## Operators

o 'Operators' can be used with variables as 'operands' to compute a relevant result. For example, in our example program, we have used an arithmetic operator '+' to add two numerical variables **a** and **b**. We also have used a concatenation operator '&' to concatenate two strings "a+b=" and the results of adding **a** and **b** converted to a String using the **Str()** function.

```
'Operators for variables
c = a + b
message = "a+b=" & Str(c)
```
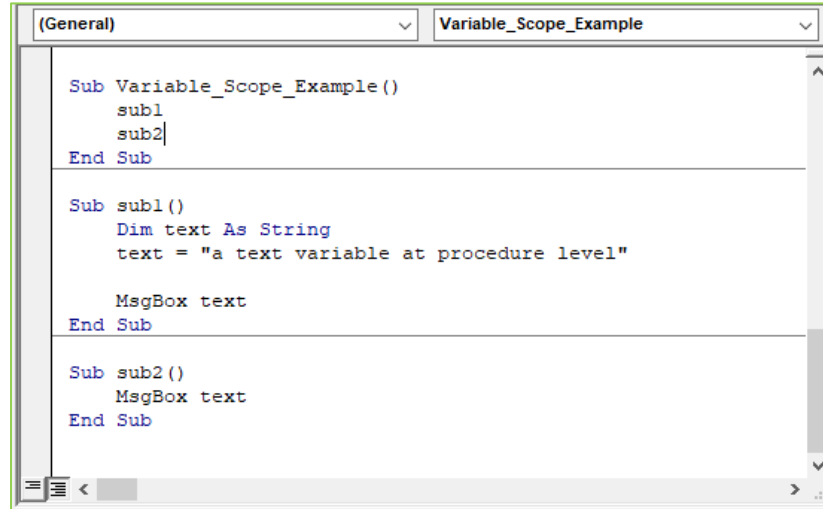
o There are four different types of commonly used 'operators' in VBA. We will only cover the most commonly used operators here but there are many resources on the web which provides a full list of operators in VBA.

| Type | Operator | What does it do? |
|---|---|---|
| Arithmetic Operator | + | Adds the two operands (e.g. 1 + 2 returns 3) |
| | - | Subtracts the second operand from the first one |
| | * | Multiplies both the operands |
| | / | Divides the numerator by the denominator |
| | Mod | Returns the remainder of a division operation (e.g. 3 Mod 2 = 1, 5 Mod 3 = 2) |
| Comparison Operator | = | Returns True if two operands are the same (e.g. 1 = 1 is True) |
| | <> | Returns True if two operands are NOT the same (e.g. 1 <> 2 is True) |
| | > | Returns True if the first operand is greater than the second (e.g. 2 > 1 is True) |
| | < | Returns True if the first operand is less than the second |
| | >= | Returns True if the first operand is greater than or equal to the second |
| | <= | Returns True if the first operand is less than or equal to the second |
| Boolean/Logical Operator | AND | Returns True if both the conditions are True (e.g. (2>1) AND (3>2) is True) |
| | OR | Returns True if any of the two conditions are True (e.g. (2>1) OR (1>2) is True) |
| | NOT | Returns the reverse of the condition (e.g. NOT True is False and NOT False is True) |
| Concatenation Operator | & | Concatenates two String operands (e.g. "hello" & "world" returns "helloworld") |

**Variable Scope**

- o The scope of a variable in Excel VBA determines where that variable may be used. You determine the scope of a variable when you declare it. There are three scoping levels:
  - o **Procedure level**: This means that a variable can only be used within a Sub-routine or a Function but not outside it. For example, see below two Sub-routines sub1 and sub2 both using a variable called 'text' but the 'text' variable is only declared inside sub1.



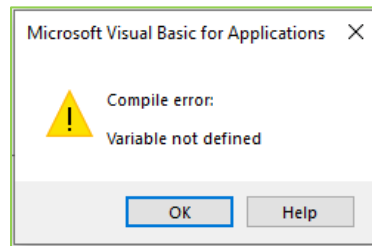  - o If you run this code, sub1 will run successfully as 'text' is declared inside it but sub2 will give the following error:

**Variable Scope**

o **Module level:** In the previous example code, if you want a variable to be available to all procedures in a module, this means that the variable needs to have module level scope. In order to do this, the variable in question must be declared at the beginning of the module outside all Sub-routines or Functions in the module per below.
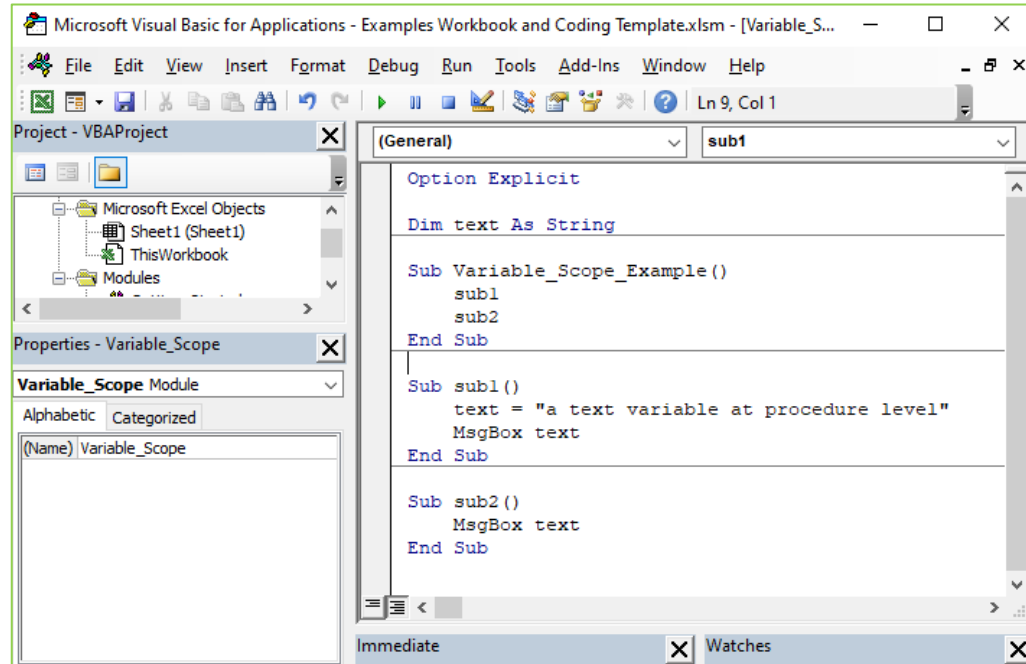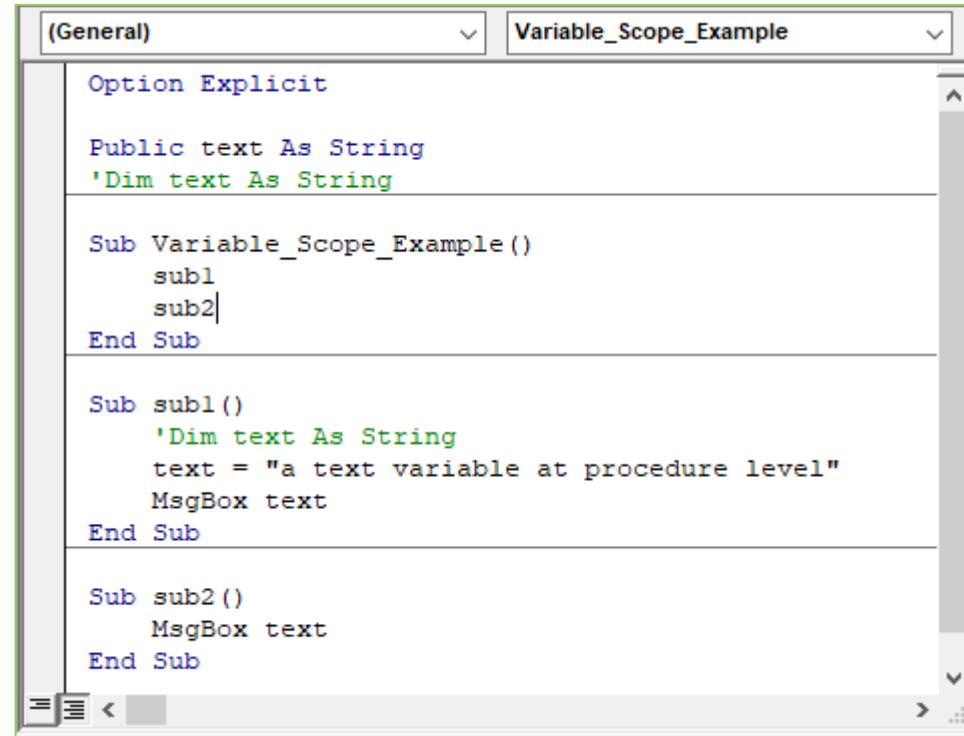


o Running the above code, will successfully prompt two separate messageboxes from both Sub-routines, sub1 and sub2 as the text variable is declared at module level and can be used by both sub1 and sub2.
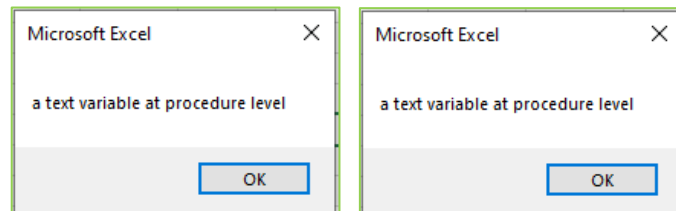
Variable Scope

o **Public module level (or Global level):** If you have multiple modules in a workbook and you'd like your variable to be available to all Sub-routines and Functions in all modules in the workbook, you can declare your variable using the 'Public' keyword per the below screenshot: **Public** <<*variable name goes here…>>* **As** <<*variable data type goes here…>>*

```
(General)                          Variable_Scope_Example

Option Explicit

Public text As String
'Dim text As String

Sub Variable_Scope_Example()
    sub1
    sub2
End Sub

Sub sub1()
    'Dim text As String
    text = "a text variable at procedure level"
    MsgBox text
End Sub

Sub sub2()
    MsgBox text
End Sub
```

# Learning by Doing: VBA Fundamentals

Some useful built-in functions for this course

Below is a list of some useful built-in Excel functions, which we can use in our VBA programs throughout this course. These functions are for manipulating String values (i.e. text).

- **LEFT(*text_string, number_of_characters*)**: returns the specified number of characters in a text string, starting from the first or left-most character. For example, in the 'Immediate Window', typing Left("abcde",4) will return the left 4 characters of "abcde" per below screenshot.

```
?Left("abcde",4)
abcd
```

- **RIGHT(*text_string, number_of_characters*):** returns the specified number of characters in a text string, starting from the last or right-most character. For example, in the 'Immediate Window', typing Right("abcde",4) will return the right 4 characters of "abcde" per below screenshot.

```
?Right("abcde",4)
bcde
```

- **MID(*text_string, start_character_number, number_of_characters*):** returns the specified number of characters in a text string, starting from a specified position (i.e. start_character_number). For example, Mid("abcde",2,3) returns 3 characters starting from a character position 2 within "abcde" per below screenshot.

```
?Mid("abcde",2,3)
bcd
```

- **LEN(*text_string*):** returns the number of characters in a given string (i.e. text_string). For example, Len("abcde") will return 5 per below screenshot.

```
?Len("abcde")
  5
```

- **InStr(*start_character_number, text_string1, text_string2*):** returns the character position of the first occurrence (from left to right) of text_string2 within text_string1. The 'start_character' parameter specifies the position (i.e. character number) within string from where you want to start your search for the substring. E.g. "bc" is found in character position 2 of the string "abcde"

```
?InStr(1,"abcde","bc")
  2
```

# Learning by Doing: VBA Fundamentals

**Excel 'Objects'**

o Excel 'Objects' refer to the 'Excel Object Model', which are essentially the building blocks that make up an Excel workbook, such as worksheets, rows, columns, cell ranges and of course the Excel workbook itself.

o Using VBA, we can access these objects and change 'properties' of these objects such as cell range's value, colour, worksheet's name and many others including charts, pivot tables, etc. This gives VBA programmers power to automate a lot of tasks in Excel.

o In the Example workbook's 'Excel Object' worksheet, we will go through examples and exercises for the following Excel Objects:
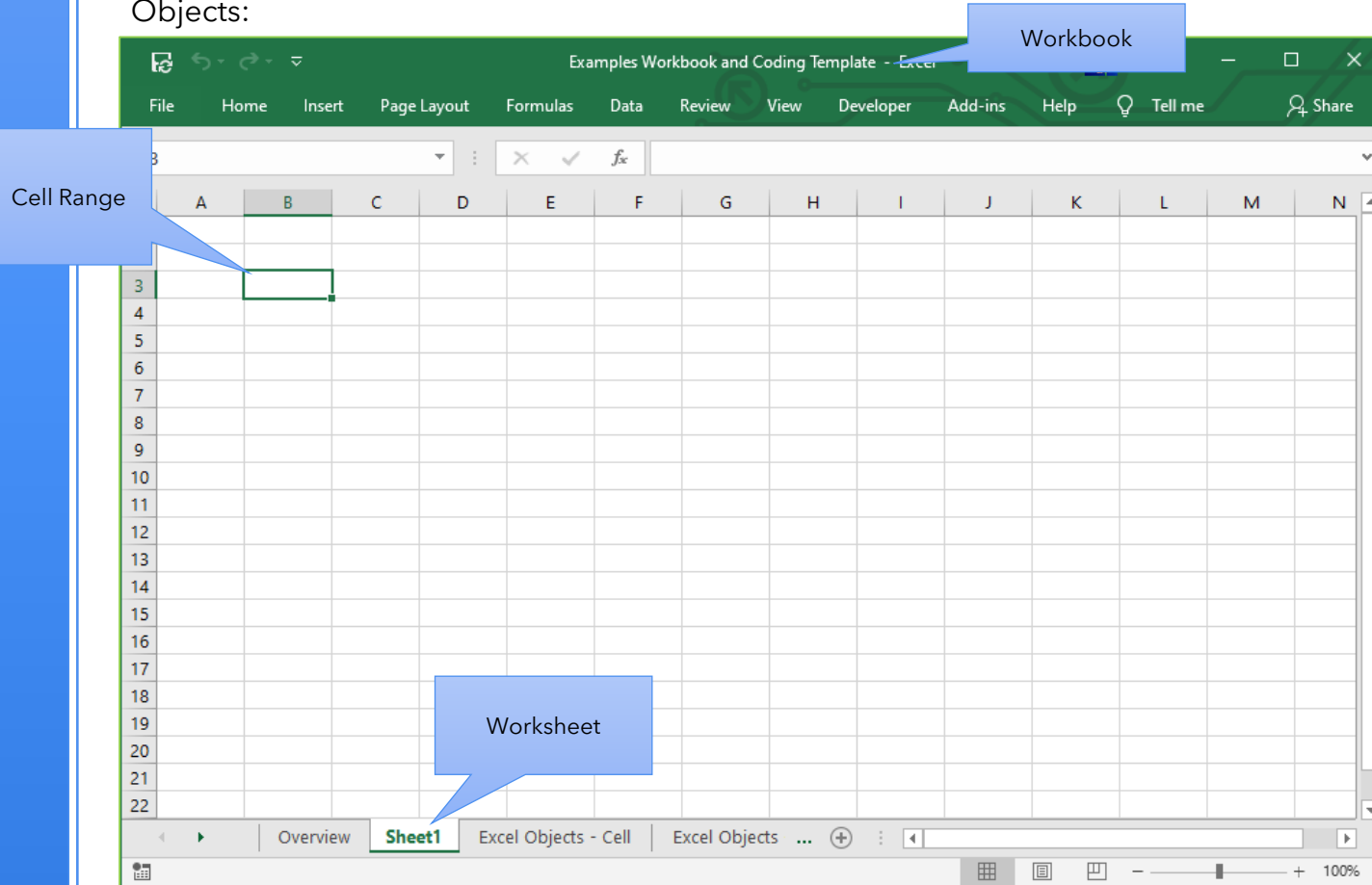


Variable assignments for **Objects** use **Set** keyword. For example:

**Dim** wb **As** Workbook
**Set** wb = ThisWorkbook

**Dim** ws **As** Worksheet
**Set** ws = ActiveWorksheet

**Dim** cell **As** Range
**Set** cell = Range("A1")

# Learning by Doing: VBA Fundamentals

**Recap: Sub-routines**

o A **Sub-routine** is a piece of code that performs a specific task and does NOT return a result. **Sub-routines** are used to break down large pieces code into small manageable parts, which can be called repeatedly.

[**Public** | **Private**] **Sub** *name* [ ( *arg_list* ) ]
[Code goes here…]
**End Sub**

```
Option Explicit

Sub First_Subroutine()
    'Your First VBA sub-routine
    'Variable Declarations
    Dim a As Integer
    Dim b As Integer
    Dim c As Integer
    Dim message As String

    'Variable Assignment
    a = 1
    b = 2

    'Operators for variables
    c = a + b
    message = "a+b=" & Str(c)

    MsgBox message
End Sub
```

Beginning of a Sub-routine

Code that gets run when the Sub-routine gets executed

End of a Sub-routine

# Learning by Doing: VBA Fundamentals

**Functions/Methods**

o A **Function or Method** is a piece of code that performs a specific task and in that sense is very similar to a **Sub-routine.** The key difference between a Function/Method and a Sub-routine is that, Function/Method returns an output (e.g. a value or a collection of values) whereas a Sub-routine does not return anything. A **Function/Method** is used to break down large pieces code into small manageable parts, which can be called repeatedly.

o A function/method follows the below syntax in VBA:

[**Public** | **Private**] **Function** *name* [ ( *arg_list* ) ] [ **As** *type* ]
[Code goes here…]
[name = expression]
**End Function**

o For example, please see below a function, which adds two input variables and returns the sum. Running 'Add_Nums(1,2)' in the Immediate Window will return 3.

```
Public Function Add_Nums(ByVal a As Integer, ByVal b As Integer) As Integer
    'Returns a + b

    Add_Nums = a + b
End Function
```

This function can be called "publicly" both inside and outside the model. If **Private** used instead, the function can only be called inside the module

The output is an integer

The output is calculated by a + b

The input for the function are a and b which are both Integers

**ByVal** means input parameters are passed in "by value"
**ByRef** means input parameters are passed in by "reference"

**Functions/Methods – ByVal and ByRef Input Parameters**

- ○ **ByVal** specifies that an input argument of a Function is <u>passed by value</u>, so that the called Function **cannot** change the value of a variable underlying the argument. If no modifier is specified, ByVal is the default.
- ○ **ByRef** specifies that an input argument of a Function is <u>passed by reference</u>, so that the called Function **can** change the value of a variable underlying the argument.
- ○ The concept of **ByVal (**pass input parameter by value) and **ByRef (**pass input parameter by reference) can seem complicated and it may not be obvious where to use this (for now) but the below example should hopefully illustrate the point. The same keyword can be used for input parameters to a **Sub-routine** as well as **Functions/Methods.**

```vba
Sub ByVal_Example(ByVal a As Integer)
    a = 1
End Sub

Sub ByRef_Example(ByRef a As Integer)
    a = 1
End Sub

Sub ByVal_ByRef()
    Dim a As Integer

    a = 10

    ByVal_Example a

    Debug.Print ("ByVal: " & Str(a))

    ByRef_Example a

    Debug.Print ("ByRef: " & Str(a))

End Sub
```

- When the ByVal_ByRef() Sub-routine is executed, it will print the following:

```
ByVal:  10
ByRef:  1
```

- When the variable **a** (initial value is 10) is passed in by value (ByVal) to 'ByVal_Example' Sub-routine, the Sub-routine <u>cannot</u> change the value of the variable **a** to 1 even if it tries to do so.
- However, When the variable **a** (initial value is 10) is passed in by reference (ByRef) to 'ByRef_Example' Sub-routine, the Sub-routine <u>can</u> change the value of the variable **a** to 1 inside the code.

**IF statement**

o **If…Then…Else** statement conditionally executes a group of statements, depending on the value of a conditional expression. For a single line statements, a single-line syntax can be used per below but multi-line syntax are most commonly used.

```
' Multiline syntax:
If condition_expression hen
    [ statements ]
ElseIf elseifcondition _expression Then
    [statements ]
Else
    [ statements ]
End If


' Single-line syntax:
If condition Then [ statements ]  Else [ statements ]
```

o For example, please see below a Sub-routine which prints different messages depending on the value of the input parameter.

```
Option Explicit

Sub Multiline_If_Example(ByVal n As Integer)
    Dim strMessage As String

    If n = 1 Then
        strMessage = "There is 1 item"
    ElseIf n = 2 Then
        strMessage = "There are 2 items"
    Else
        strMessage = "there are" & Str(n) & " items"
    End If


    Debug.Print (strMessage)
End Sub
```

# Learning by Doing: VBA Fundamentals

**For Loop**

o Repeats a group of statements a specified number of times. There are two ways of writing a For loop syntax per below:
  o For…To…Next – Syntax:

  **For** counter = start **To** end [ **Step** step ]
  [ statements ]
  …
  [ statements ]
  **Next** [ counter ]

  > Step is an optional parameter and if omitted default step = 1

  o Example code and output:

```vba
Sub For_To_Loop()
    Dim i As Integer

    For i = 1 To 10
        Debug.Print (i)
    Next
End Sub
```

  Immediate
  ```
  1
  2
  3
  4
  5
  6
  7
  8
  9
  10
  ```

  o For…Each…Next – Syntax:

  **For Each** element **In** group
  [ statements ]
  …
  [ statements ]
  **Next** [ element ]

  o Example code and output:

```vba
Sub For_Each_Loop()
    Dim cell As Range
    Dim i As Integer

    i = 1

    For Each cell In Range("A1:A10")
        cell.Value = i
        i = i + 1
    Next
End Sub
```

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

# Learning by Doing: VBA Fundamentals

**Do...Loop**

o Repeats a group of statements while a condition is met. There are two ways of writing a Do loop syntax per below:

o Do...While...Loop – Syntax:

**Do While** [condition expression]
  [statement]

  ...
  [statement]
**Loop**

o Example code and output:

```
Sub Do_Loop()
    Dim i As Integer

    i = 1

    Do
        Debug.Print (i)
        i = i + 1
    Loop While (i <= 10)
End Sub
```

Immediate
```
1
2
3
4
5
6
7
8
9
10
```

o Do...Loop...While – Syntax:

**Do**
  [statement]

  ...
  [statement]
**Loop While** [condition expression]

o Example code and output:

```
Sub Do_While_Loop()
    Dim i As Integer

    i = 1

    Do While (i <= 10)
        Debug.Print (i)
        i = i + 1
    Loop
End Sub
```

Immediate
```
1
2
3
4
5
6
7
8
9
10
```

# Learning by Doing: VBA Fundamentals

**Arrays**

o Usually, a variable holds one value or has one "compartment" to store a value. For example an integer variable stores one integer value. An **array** however is a variable that can hold multiple values of the **same data type** (i.e. has many compartments to store values of the same data type). Arrays could be 1-dimensional (a "list"), 2-dimentional (multiple rows and columns or multi-dimensional. For this course, we will master the 1-dimensional array only and we will cover 2+ dimensional arrays in an advanced course which will follow soon.

o Why would we need such a variable? Let's say that we want to store the name of the days in a week (e.g. Monday, Tuesday, etc). We could create 7 separate variables of type String to store these names but declaring a single array allows us to store all 7 days, where each **element** in the array will store a day of **String** type. Pictorially, this can be represented per below and example code and output are shown below.

**Array**

| "Monday" | "Tuesday" | "Wednesday" | ... | "Sunday" |

```
(General)

Option Explicit

Sub Array_Week_Example()
    Dim days(7) As String
    Dim day As Variant

    days(1) = "Monday"
    days(2) = "Tuesday"
    days(3) = "Wendesday"
    days(4) = "Thursday"
    days(5) = "Friday"
    days(6) = "Saturday"
    days(7) = "Sunday"

    For Each day In days
        Debug.Print (day)
    Next
End Sub
```

Array Declaration Syntax:
**Dim** name(num_elements) **As** data_type

Storing values within each element of an Array

Using **For Each** loop to iterate through all elements in an array and performing an action, in this case, printing to the 'Immediate Window'.

Immediate
```
Monday
Tuesday
Wendesday
Thursday
Friday
Saturday
Sunday
```

**Debugging a VBA program**

- **Debugging** is a process of identifying and removing errors or unintended behaviour from your program.
- As you start building more complex program in VBA, you will encounter errors/unintended behaviour from your program. The more lines of code you have in your program, the harder it will become to identify errors by just reading through your code only.
- Fortunately, the VBA editor comes with a powerful debugging tool, which will help you to step through your code and see outcomes and state of each variables for each line of code. In the VBA editor, the 'Debug' menu option holds a series of useful tools for debugging, such as '**Step Into (F8)'** functionality, which lets us step through each line of code from top to bottom and '**Watches'** window that can be used to monitor the values of variables throughout the execution of the program.
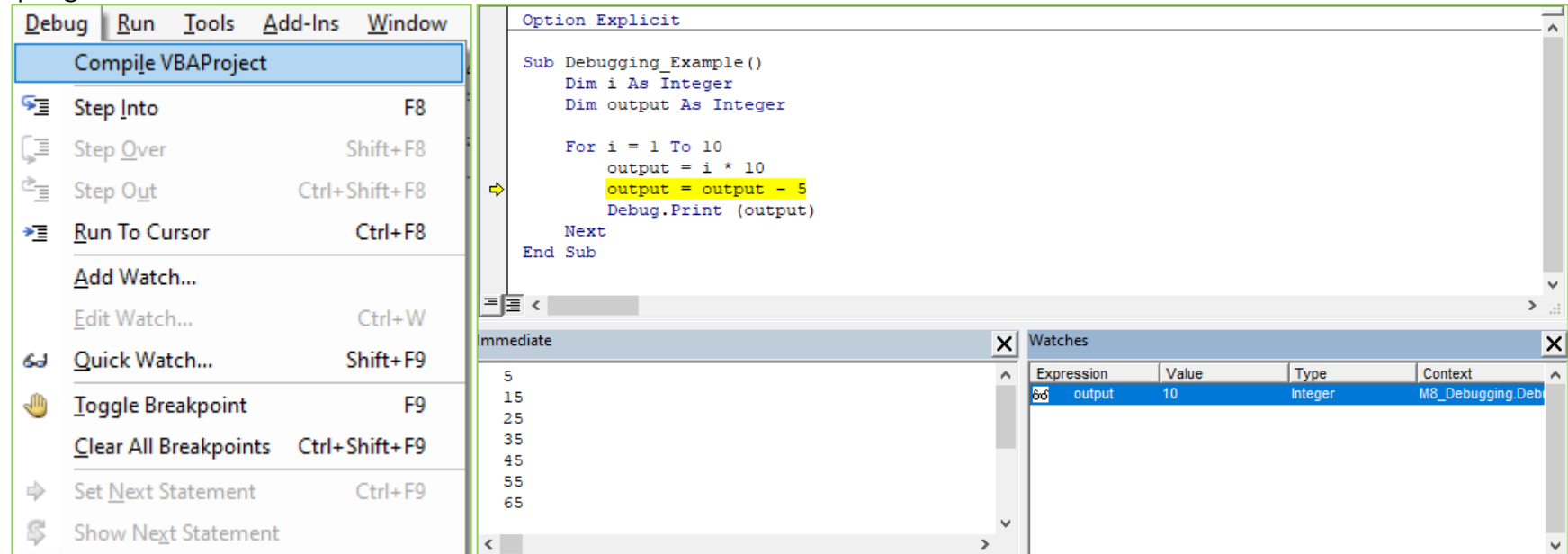


- In the Example workbook's 'Debugging' module, we will go through an example program and use these debugging tools.

**Error Handling: On Error GoTo statement**

o Error handling means a planned response to error scenarios. This involves a programmer to put in a specific error-handling routines, which will perform a specific action if errors occur during a program execution.
o In VBA, there are a number of ways of enabling error handling routines but in this course we will go through an example of one of the commonly "**On Error GoTo...**" statement.
o In the Example workbook's 'Error_Handling' module, we will go through 2 example programs, both with a deliberate error but one Sub-routine **without** error handling and another Sub-routine **with** error handling routine.

```
Sub Error_Example_With_Error_Handling()
    Dim i As Integer
    Dim s As String
On Error GoTo error_handler

    i = 10
    s = "hello"

    Debug.Print (s + i)

    Exit Sub

error_handler:
    If Err Then
        Debug.Print (Err.Description)
    End If
End Sub
```
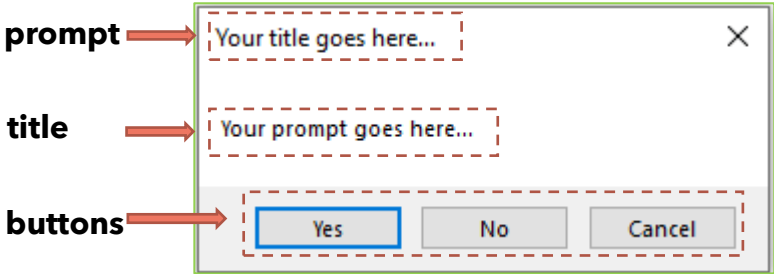
**On Error GoTo** error_handler_name
• Having this statement at the beginning of the program ensures the program code to jump to the point of code with the 'error_handler_name' label, where an error handling routine should be implemented

When an error occurs during program execution, the **On Error GoTo** statement instructs the VBA interpreter to jump straight to the specified label and any code below the label gets executed. The VBA's **Err** object stores details about the error raised during program execution, such as the description of the error, which we are printing out in this example using **Err.Description** statement.

# User Interaction

## Message Box

o **MsgBox** function can be used to display messages as well as get end user feedback via the buttons. Basic syntax is per below:

> **MsgBox** (*prompt*, [ *buttons*, ] [ *title*, ] [ *helpfile*, *context* ])

o Example output and what input arguments change:



o Example argument values for the 'buttons' input parameter:

| Buttons argument values (examples) | Return Value | Description |
| --- | --- | --- |
| vbOKOnly | 0 | Display OK button only. |
| vbOKCancel | 1 | Display OK and Cancel buttons. |
| vbAbortRetryIgnore | 2 | Display Abort, Retry, and Ignore buttons. |
| vbYesNoCancel | 3 | Display Yes, No, and Cancel buttons. |
| vbYesNo | 4 | Display Yes and No buttons. |

o Let's walk through the examples and exercises in the Examples workbook to see how to use 'MsgBox' function.

# User Interaction

## Input Box

o **InputBox** function displays a prompt in a dialog box for end user's input, which will then be returned as a String output when user clicks the OK button. Basic syntax is per below:

**InputBox**(*prompt*, [ *title* ], [ *default* ])
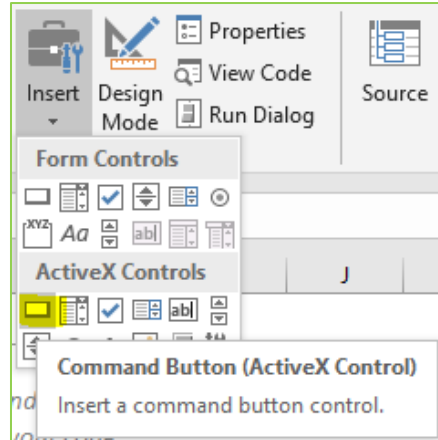
o Example output and what input arguments change:
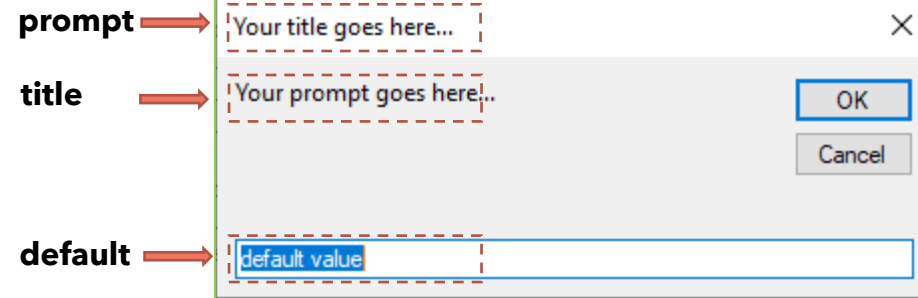


o Let's walk through the examples and exercises in the Examples workbook to see how to use 'InputBox' function.
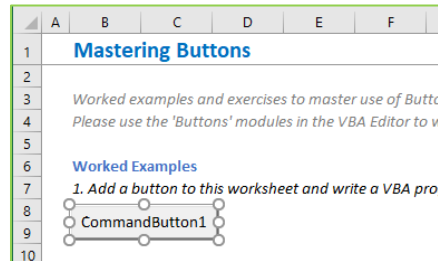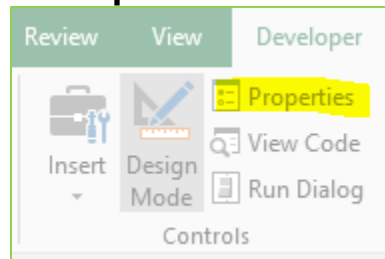
# User Interaction

## Programming a Button

o How to create a button in a spreadsheet?
  o **Developer** tab > **Insert** > **Active X Controls' Command Button**



o Drag and drop a button in a suitable position in a "target" worksheet. Resize as necessary by dragging.



o Keep the Button object highlighted per above (i.e. clicked) and then click on the '**Properties'** button on the **Developer tab**
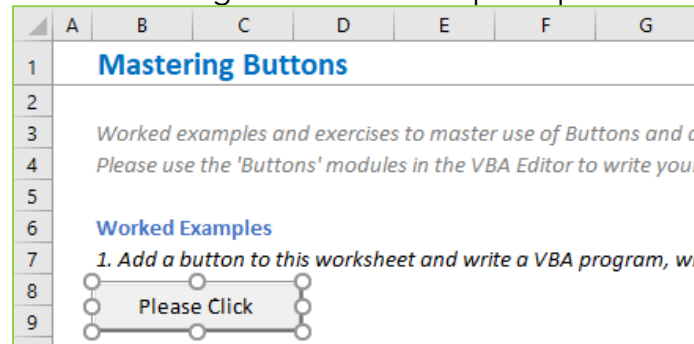
**Programming a Button - Continued**

o The 'Properties' window shows all properties of the selected Button, which we can change. For example, let's change the Button object's name to 'ExampleButton1' and also change the caption (i.e. what gets displayed on the face of the Button) to 'Please Click' per below screenshot. When done, close the Properties window.
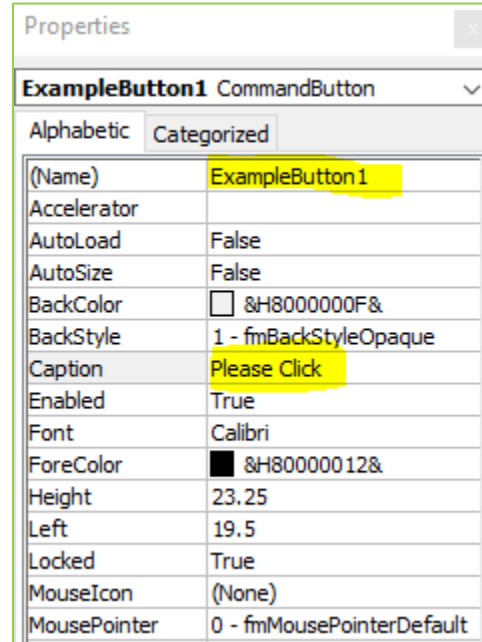


o This will change the button's caption per below screenshot.
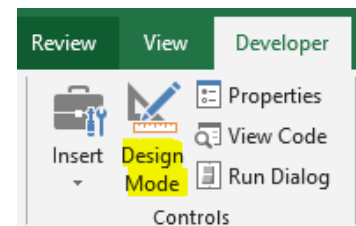
# User Interaction

## Programming a Button - Continued

○ **Double-click** on the button and this will take you to a Sub-routine which gets automatically generated in the VBA editor in the worksheet where you have just created the button (i.e. Sheet12 (Buttons)). This is the Sub-routine, which gets executed when the button gets clicked.
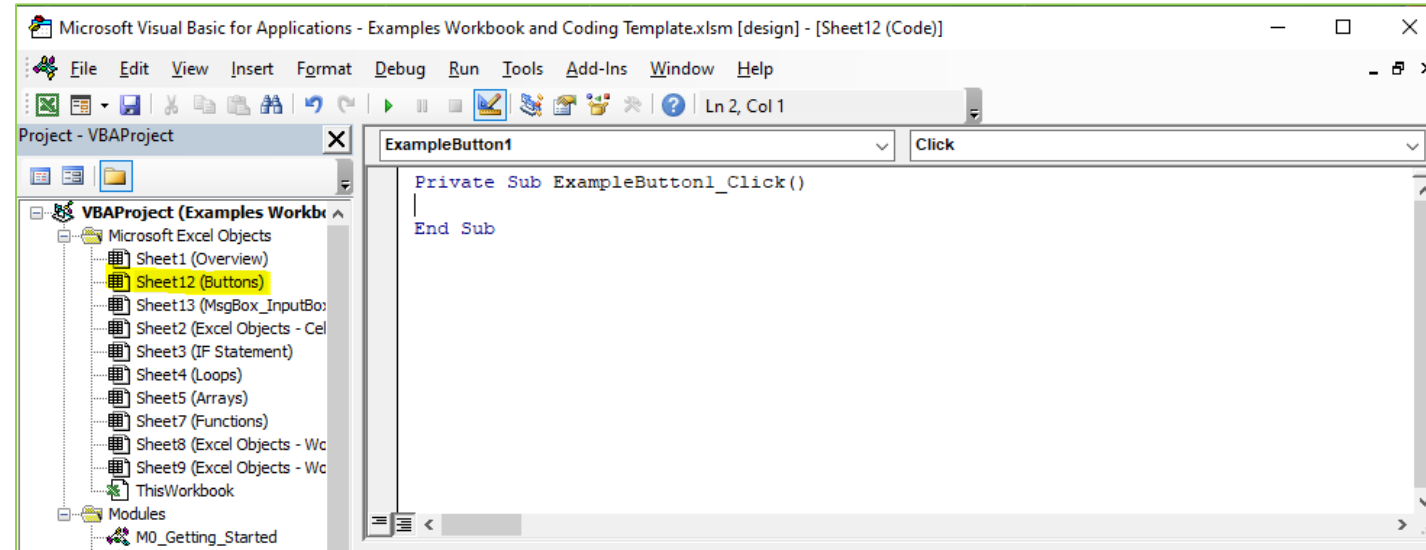


○ Let's write a simple code which prompts a MsgBox, when the button is clicked.

```
Private Sub ExampleButton1_Click()
    MsgBox "Hello world"
End Sub
```
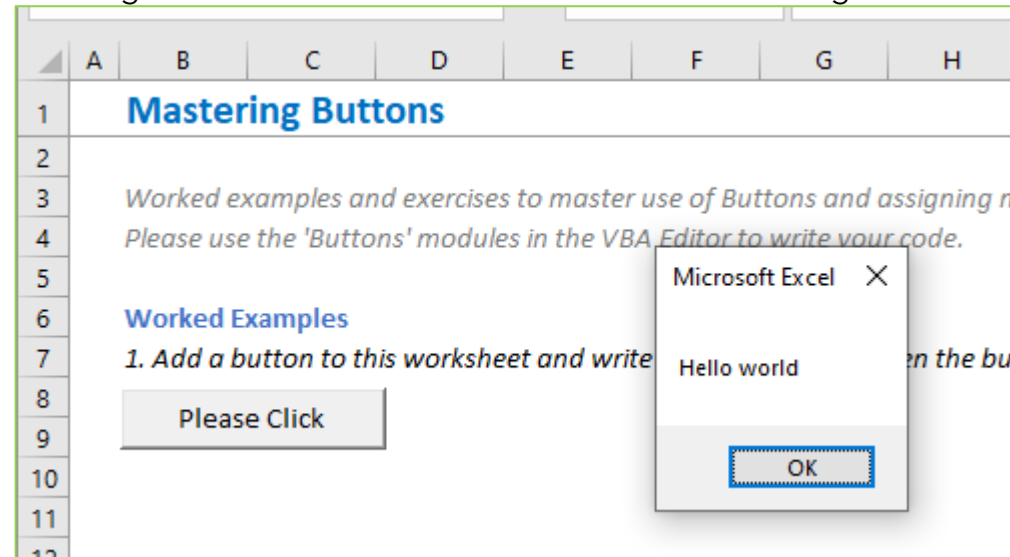
○ To test out the button, go back to the **Developer** bar and make sure the **Design Mode** is unclicked per below. When the **Design Mode** is clicked, Active X controls will not run its assigned VBA code even if clicked.

**Programming a Button - Continued**

o Clicking on the Button on the worksheet shows a message box with "Hello world" message as programmed.



o In the Example workbook's 'Buttons' module, we will go through examples and exercises to learn more about programming buttons.