Microsoft Excel VBA

TABLE OF CONTENTS

Chapter 6: *Modules*

- Why is module needed?
- Code Module
- User Form in brief
- Class module in brief
- Worksheet module in brief
- Workbook module in brief
- Procedures scoping
- Dealing with ambiguities

Chapter 7: *When you make mistake…*

- Type of errors
- Dealing with compilation errors
- Dealing with runtime errors
- Dealing with logical errors
- The debugger and debugging process

Chapter 8: *The Variables*

- Why are variables needed?
- Basic Data Types
- Variable declaration and shorthand
- Variable scoping and life cycle
- Variable initialization
- **Option Explicit** directive

Chapter 9: *Useful VBA Native Functions*

- MsgBox function
- InputBox function
- Number functions
- String functions
- Date/Time functions
- Format function
- Data Type Validation

Chapter 10: *Other Useful Basic VBA Entities*

- Comments
- Keywords
- Identifiers
- Constants
- Selection keyword
- Application object
- ActiveSheet object
- Sheets collection
- Workbooks collection

Chapter 11: *The Parameter*

- What is parameter?
- Optional parameters and techniques to handle default values
- Arbitrary argument support using ParamArray declaration
- Parameter passing mechanisms: **ByVal** vs. **ByRef**
- Named arguments

Chapter 12: *Operators*

- What is operator?
- Arithmetic operators
- Comparison Operators
- Logical Operators
- Special Operators

Chapter 13: *Branching Constructs*

- Unconditional Branching with **GoTo** statement
- Unconditional Branching with **GoSub** statement
- **If..Then..Else** Statement
- **Select Case** Statement

Chapter 14: *Iteration Constructs*

- Unconditional Loop with **GoTo** statement
- Using **For** Loop
- Using **For Each** statement
- Pre-test looping
- Post-Test looping
- Pre-mature termination using **Exit** keyword

Chapter 1*: Introduction*

What is VBA?

Microsoft® Visual Basic for Applications is a hosted language and part of the Visual Basic® family of development tools. Although VBA can be thought of as sitting below the retail version of VB and above VBScript in the VB hierarchy, VBA is an essential element of the retail version of VB, providing many language elements used in VB. When hosted in VB, VBA provides language support and an interface for forms, controls, objects, modules, and data-access technologies. When hosted in other applications such as Excel or Word, VBA, using a technology called **Automation**, provides the means of interacting with and accessing the host application's object model, as well as the object models of other applications and components.

Until the launch of VBA 5.0 in early 1997, the language had no development environment; very much like VBScript today, VBA was simply a language interpreter. VBA 5.0 marked the start of an exciting new chapter for VBA; it now has its own integrated development and debugging environment running within the process space of the host application.

VBA itself becomes more object-oriented with each release, but the latest release (Version 6.0) adds relatively few functions and keywords to the VBA language. Instead, extra functionality has been incorporated into VB6 using new object models, and again it's the VBA language that allows you to integrate these object models into your application.

Why do I need to do programming for MS-Excel?

To customize complex applications such as Excel from Microsoft, VBA allows the developer to provide solutions that take advantage of sophisticated components that have been tried and tested. VBA is a glue language: a language that interfaces with the various objects that make up an application via the host application's object model. VBA is how applications can become extensible, and it's ActiveX (or Object Linking and Embedding automation) that provides the interface between VBA and its host application. It's this support for OLE automation that makes VBA an outstanding tool for rapidly developing robust Windows applications.

Few interesting examples

Example 1:
Given a term in a cell, you are required to form an abbreviation that based on the uppercase letters of each word in the term. How can you do this with excel formula?

Abbreviation	Term
VB	Visual Basic
VBA	Visual Basic for Applications
OLE	Object Linking and Embedding
IDE	Integrated Development Environment
VBE	Visual Basic Editor

Example 2:
Prime number is an integer number that can be fully divided by 1 and itself. Given a set of positive integer numbers, can you create Excel formula to determine a given cell is a prime number?

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

Example 3:
How to automate the proper formatting to range of cells in MS Excel with the macro?

Proper Name
Name
yOng Tau FOO
LOW see Fun
TOng sAM paH
low mai kai

Proper Name
Name
Yong Tau Foo
Low See Fun
Tong Sam Pah
Low Mai Kai

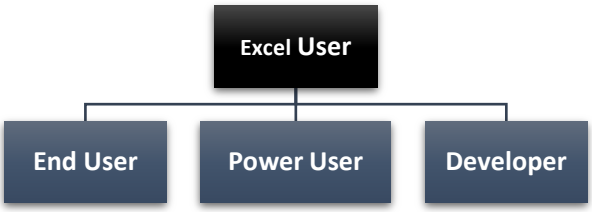
Example 4:
How to auto generate random numbers and highlight all the prime numbers under a selected range?

Randam Data			Highlight Primes		
58	10	10	80	28	5
30	38	30	95	98	40
28	16	16	65	41	41
71	33	63	21	19	58
8	46	91	26	79	38
29	92	63	63	43	10
56	69	91	83	2	54

Randam Data			Highlight Primes		
58	10	10	80	28	5
30	38	30	95	98	40
28	16	16	65	41	41
71	33	63	21	19	58
8	46	91	26	79	38
29	92	63	63	43	10
56	69	91	83	2	54

Example 5:
How to automatically convert text entered in a cell to uppercase upon leaving the cell?

Basically, there are 3 types of MS Excel users as listed below:



Based on the few examples above, you should realize that many of the problems (even simple ones) cannot be solved until you become a developer. This is the reason why you need to learn VBA.

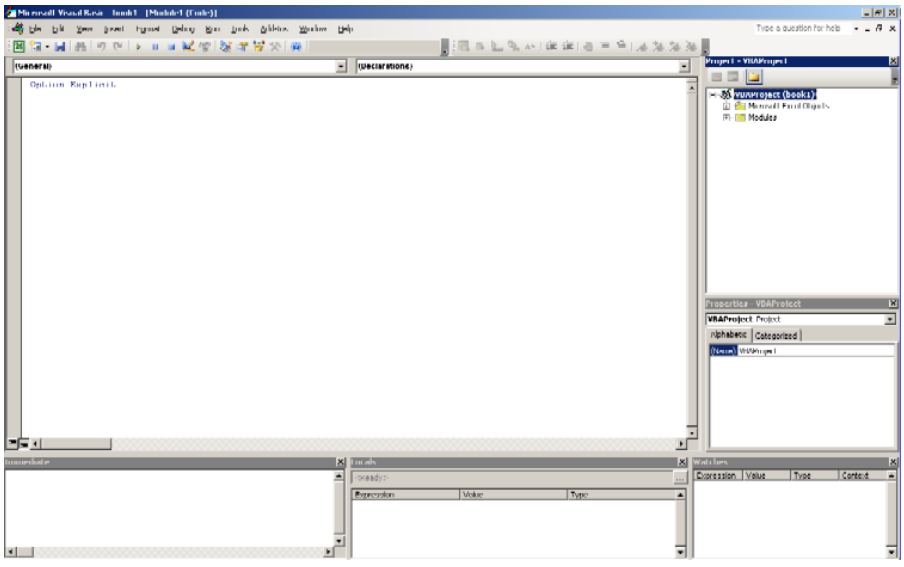
Computer programming has become much easier than before. Of course, to be a great programmer the journey is long, but at least with simple programming you might be able to solve some simple problems where even the power user not able to solve.

Programming means developing instructions that the computer automatically carries out.

Using the development IDE

Developers need to depend on tools to develop a piece of product. Software developers likewise use many software tools in the development process. Modern development depends on tools that normally integrated as one under a single environment. This is the concept of **I**ntegrated **D**evelopment **E**nvironment.

VBA is a powerful programming language that is embedded in every Microsoft Office product under a "hidden world". Whether you are using Word, Excel or PowerPoint, you are only ever a few mouse clicks (or keystrokes) away from starting to write your own programs under the IDE called **V**isual **B**asic **E**ditor.

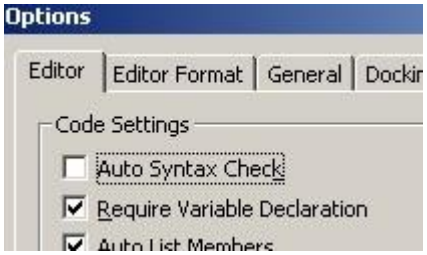


You can set up the IDE to support the way you work. Some people like to have all the main tool windows open all the time. Others like to just have one or two open. Note though once you have them docked like above you quickly learn not to change it as it is so hard to re-dock them the same way.

VBA Environment

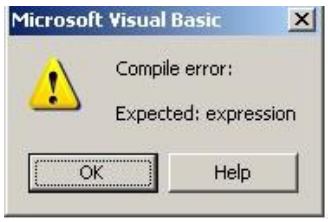
Unfortunately, the default environment settings are not geared towards high quality development, they are set to make it easy for beginners to get productive quickly.

These are our preferred settings for professional quality development (**Tools**->**Options**)



The default is to have those top 2 reversed.

You always get Auto Syntax Check (line goes red), removing the tick stops the disruptive modal error message boxes from constantly popping up every time you move off a line to copy something:

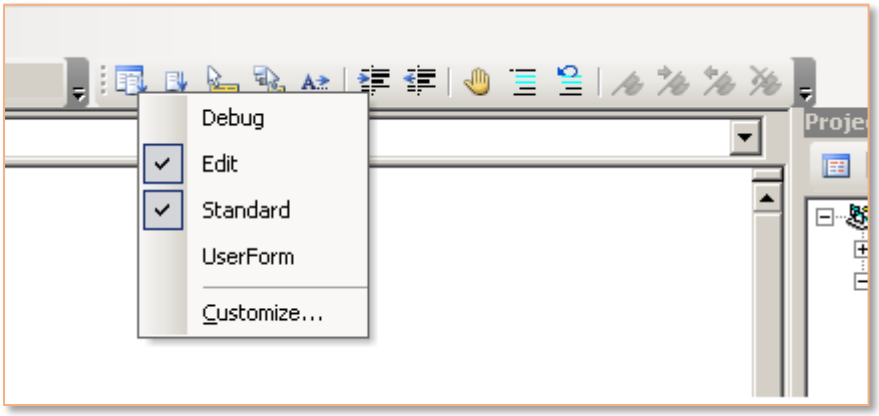


If the advice were a bit more helpful it might be worthwhile, but even beginners would struggle to get any value out of the above example. And it isn't going to get better as VBA is the end of line. Advice: Turn off Auto Syntax Check.

Require Variable Declaration puts an 'Option Explicit' at the top of each new code resource you open (module, class, form etc) (note it is not retrospective, hence the need to set it ASAP). Not using option explicit is just sloppy and is sure to lead too hard to spot errors in any significant coding.

if you want to use a variable called x (you may be able to think of a more meaningful name) Option Explicit forces you to declare it first. If without Option Explicit, VB will implicitly declare the variable as a variant the first time you use it, if you later mistype the variable name VB will create another new variable, rather than warn you 'Variable not defined'. This default feature is degerous. It will easily cause **Logical Error** in our program. This type of error is the most defficult to fix.

Another bizarre default setting is the way the IDE does not show one of the most important toolbars as standard. Luckily you can right click in the menu area and show the Edit toolbar.

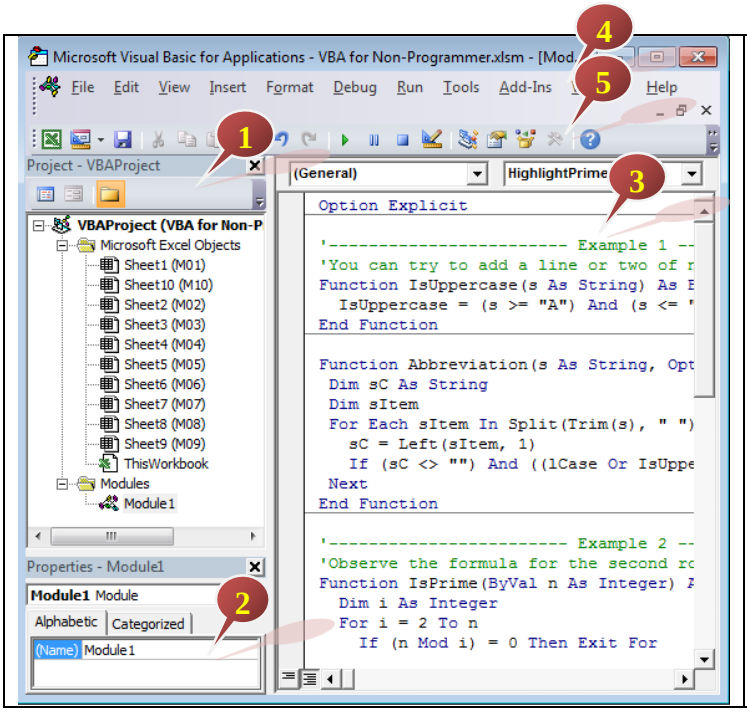


The 2 very useful commands here are the ones with blue lines; the first one comments out a line, the one with the blue arrow uncomments it.

***Note**: These settings will be reflected in other Office applications.*

Getting to the Visual Basic Editor

Whichever Microsoft Office application you happen to be using, you can be sure that embedded within it is another application component called theVBE. This is the special component that you use to write your VBA code. There are various menu or ribbon options that will take you to the VBE, depending on which application and which version of Office you are using, but you can always get to the VBE with a keyboard shortcut. To do this, hold down the **ALT** key on the keyboard, and then press **F11**.



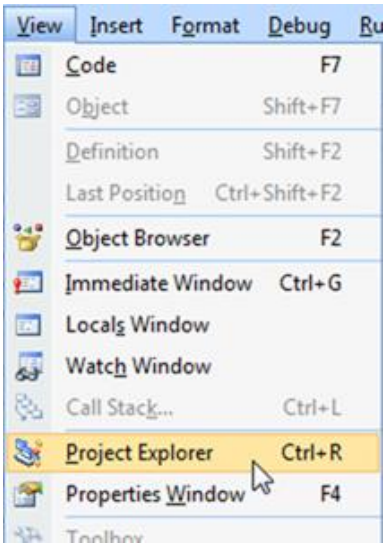
This is how the VBE should look the very first time you open it.

The two highlighted windows are:

1. *The Project Explorer*
2. *The Properties Window*
3. *Code Editor*
4. *Main Menu*
5. *Toolbar*

The VBE is similar for different Office applications you are in when you open it. You will see slightly different things in the **Project Explorer** and **Properties** window depending on which application you are using - the diagram above is using Excel 2007. Don't worry, the IDE for Excel 2003 to 2018 looks the similar.

If you can't see the two windows that we've highlighted in the image above, you can go to the **View** menu at the top of the VBE to display them.



You can click the options shown here to show the two relevant windows or use the keyboard shortcuts that are listed next to the options in the menu.

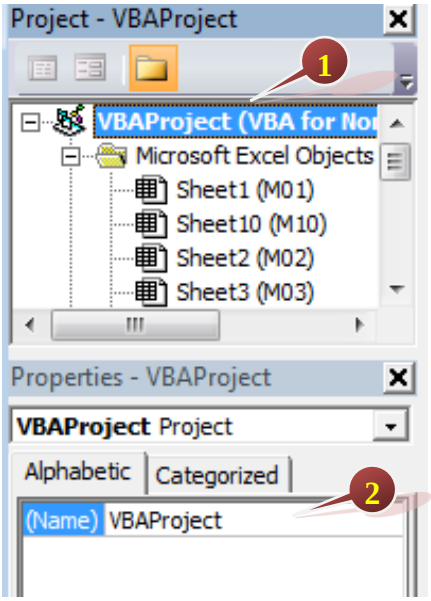
Alternatively, you can press Ctrl + R *to turn on the Project Explorer window, and* F4 *to turn on Properties Window*

The Project Explorer - *Introduction to the VB project concept and project components*

A ***project*** is the name for the collection of VBA objects that are part of the file you are working on. A project is created automatically when you open a new Office file (such as an Excel workbook, Word document, or PowerPoint presentation), and you can only have one project associated with one file.

Working with Projects

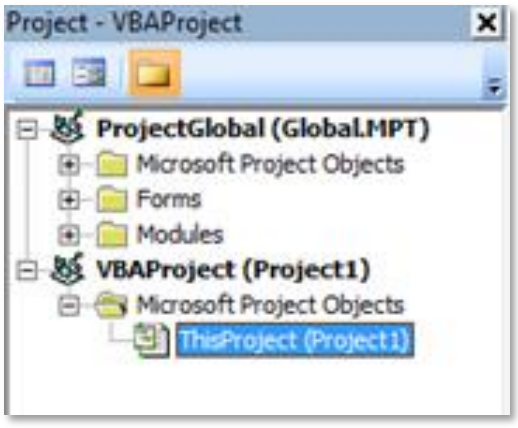
The only useful thing you can really do with a project at this point is to rename it. To do this:

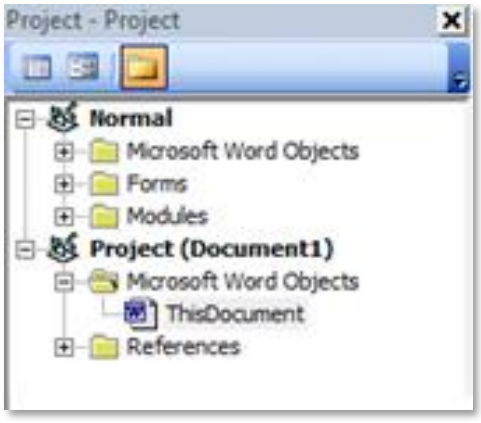


1. *Click on the project in Project **Explorer**.*
2. *Type in a new name for the project in the **Properties** window.*

A project name can't contain spaces and various other punctuation characters - it's best to stick to text and numbers.

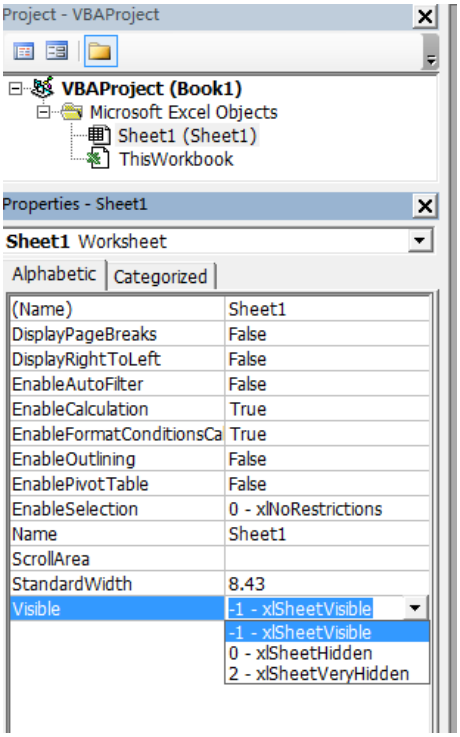
Although you can only have one VBA project for each Office file, you might also see other projects listed when you go into the VBE. These extra projects include things like the Excel **Personal Macro Workbook**, the Word **Normal template**, Excel **Add-Ins**, and the Project **Global template**.



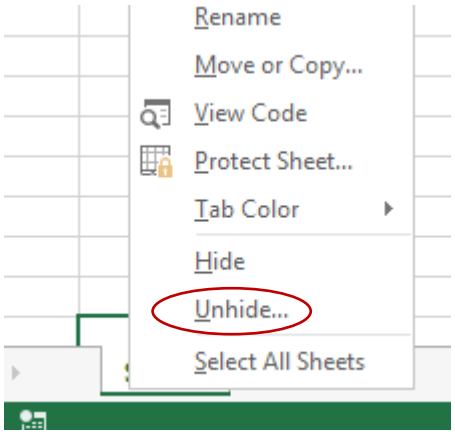


The property Window

We can view the Excel workbook consisting of a set of objects. These objects have a set of aspects called Property. Each of these properties holds a value that determines the object state. For example, the **Visible** property of each worksheet object has three possible values, namely *xlSheetVisible*, *xlSheetHidden*, and *xlSheetVeryHidden*.



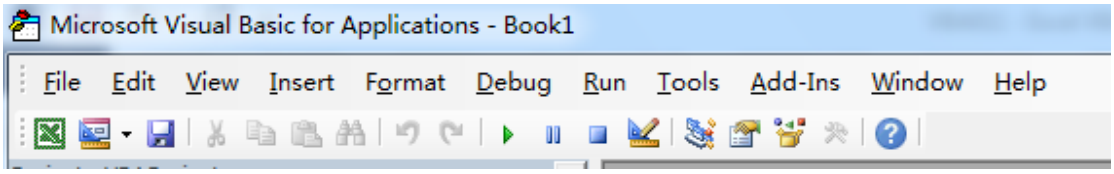
Both property values xlSheetHidden and xlSheetVeryHidden will cause the worksheet diappear. But you can use normal Excel interface to unhide the worksheet with property value xlSheetHidden,



But not for worksheet with Visible property value **xlSheetVeryHidden**.

The IDE main menu

Below is the IDE main menu:



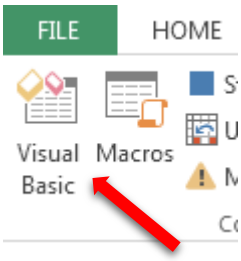
This main menu has many useful options to help us to deal with VBA development. To trigger these options, we can use mosue click or many of them can react to **Hot Key** or **Shortcut Key**.

We will explore most of these options soon.

Switching between Excel normal interface and IDE interface

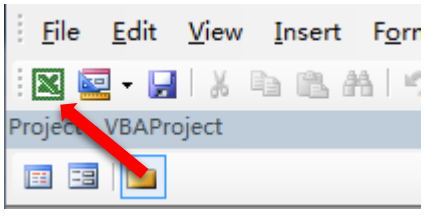
We can trigger VBA IDE interface from normal interface by using the following ways:

1. Use Edit from macro dialog box
2. Pressing **Alt** + **F11** key
3. Use **Developer** tab (enable it first), press the **Visual Basic** button.



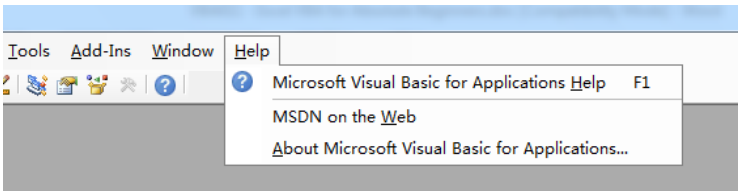
Both normal Excel interface and IDE interface are referring to same workbook(s). We can use dual screen mode to perform development more productively. If there is only one screen in used, we can use **Alt** + **Tab** window key to switch around different interface windows.

To move from VBA IDE back to normal Excel interface, we also can press this button:



Help System

Under the VBA IDE, we can press **F1** key or select from the main menu to lauch the help system.



Depending on the installation, the help might point the web by using browser, or we can choose to use offline version.

Help also available from the editor by highlight keyword then press **F1**.

Chapter 2: *Automation via Macros*

Why do we need automation in MS-Excel?

Many Excel users use Excel to perform their daily tasks. Some of these tasks are time consuming and if do it manually might cause mistakes. The solution applies macros. The user can record the task into macro and play back to perform the task.

Therefore, using macos we can gain two main advantages:

1. Save time
2. Eliminate manual mistakes

What is macro?

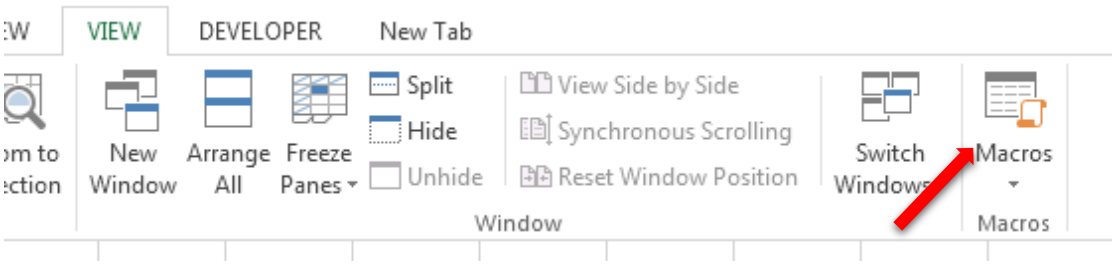
Basically, the macro is a VBA subroutine stored in the excel module. The macro can be created in two ways:

1. Recording
2. Hand coding

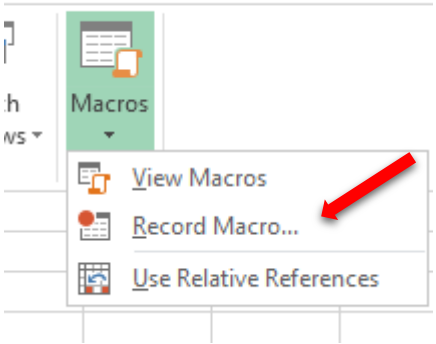
We can view the macro from VBA IDE.

Recording macros

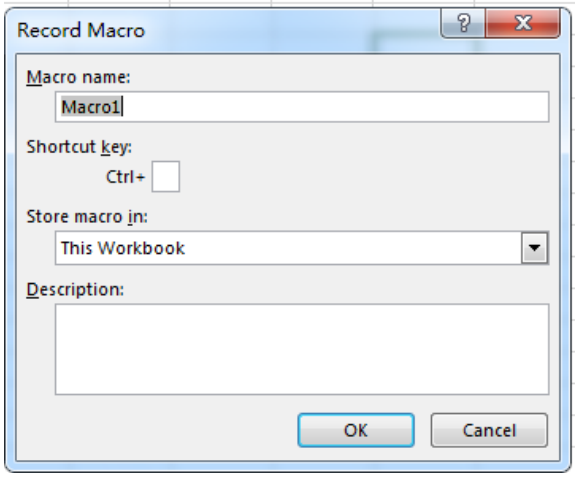
To record macro, switch to the **View** ribbon tab, the select the macro botton:



Under the drop down, select **Record Macro…**



The Record Macro dialog box appears:

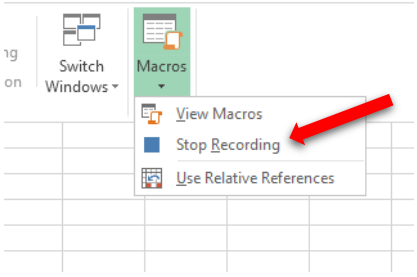


The next step is to key in the macro name. The default name is macro*, where * is a running number based on the times macro is recorded since the workbook is opened. We can change the name by using VBA naming convension (will discuss further in other section)

We can decide the shortcut key later. Macro also can be assigned to other open workbooks.

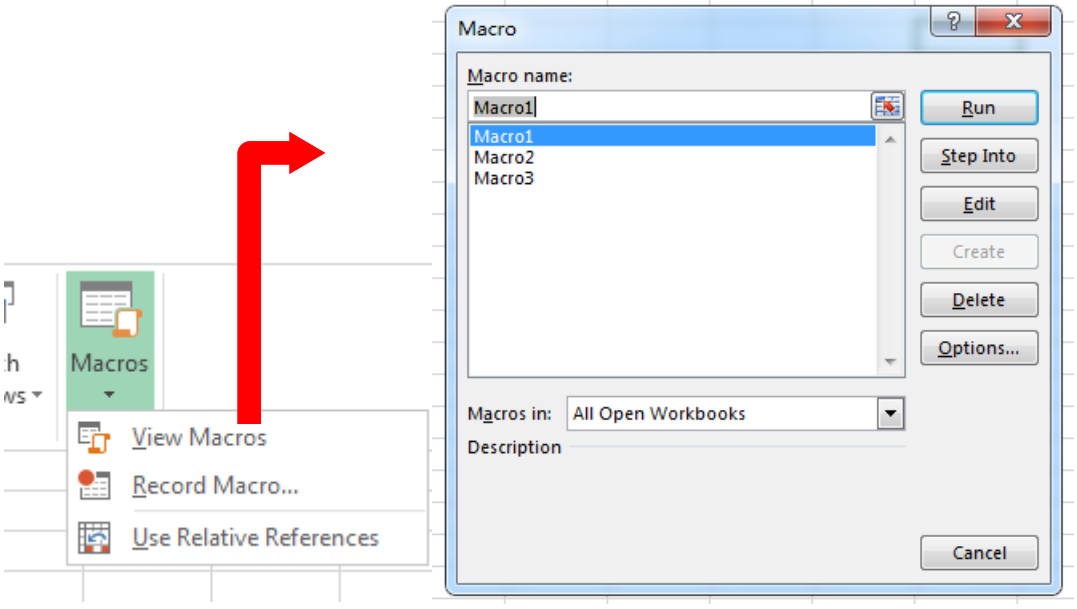
Press **OK** button to start. From now on, most of the action's user perform will be recorded.

After the task is completed, select the **Macros** button from **View** ribbon tab, click on the **Stop Recording** option.



To view the recorded macro:

Switch to the **View** ribbon tab, the select the macro botton again, then select **View Macros** option, Macro dialog box appears:

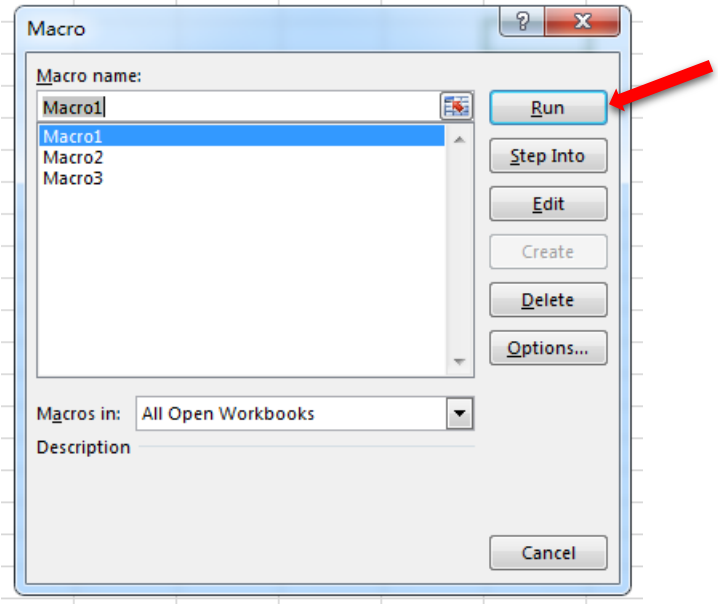


Select the macro to view from the listbox, then press **Edit** button. The macro will be shown at the IDE editor.

How to trigger macros from Excel normal interface?

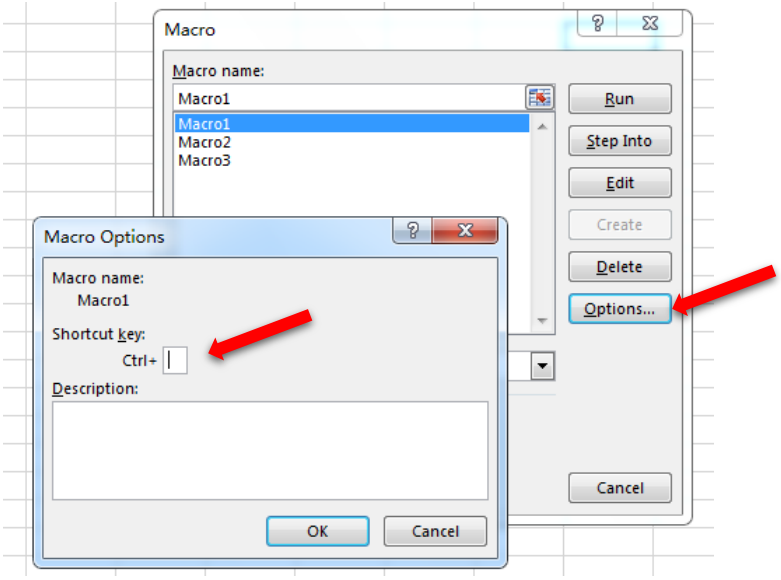
There are few ways we can trigger/run the macro from Excel normal interface:

1. From the Macro Dialog Box

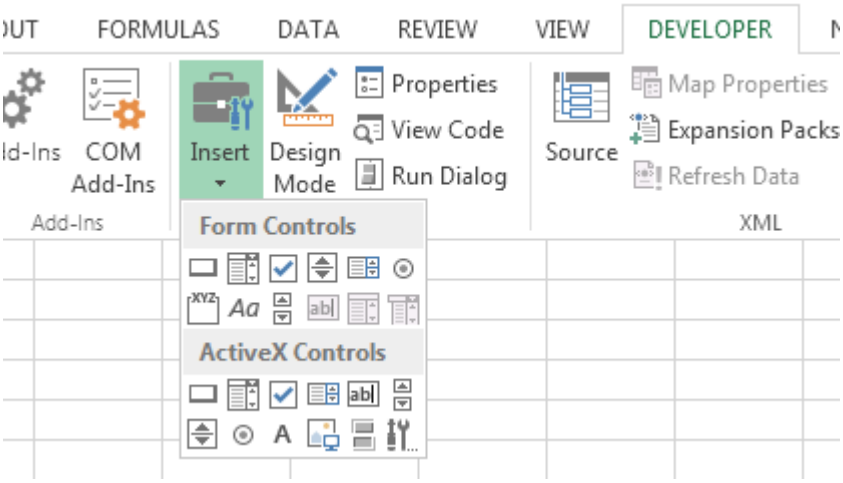


2. Use shortcut key. Firstly, we must assign shortcut key either during macro creation or assign later by using **Options...** button.

   Take note that normally Ctrl+<letter> reserved for Excel itself, therefore we can use Ctrl+Alt+<letter> instead.



3. Assign the macro to Form Controls, such as button. This required us to turn on the Developer tab from excel interface. For Excel 2003, we use Visual Basic Toolbar.

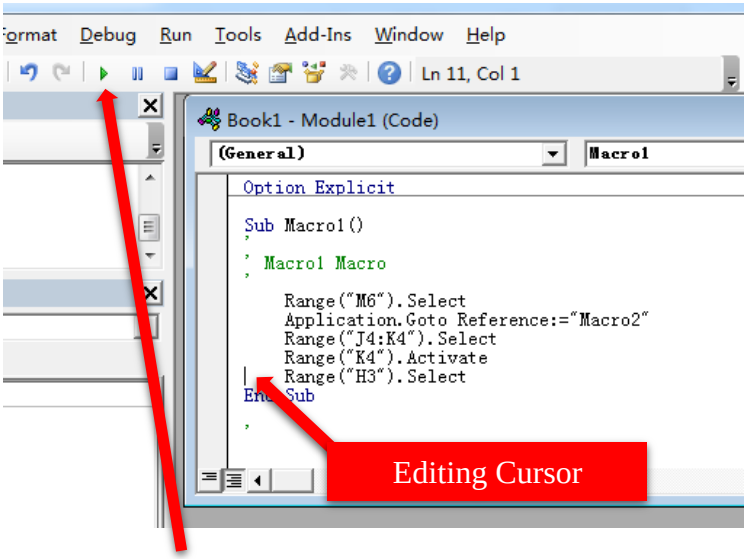


4. For Excel 2010 onward, we can easily attach macros to ribbon tab too. Sorry, Excel 2007 is hard to do this.

How to trigger macros from VBA IDE?

To activate the macros under VBA IDE, we can

1. Call the macro from Immediate window (Discuss later)
2. Move the editing cursor the subroutine declaration, then



   a. Press the **Run** button, or

   b. Press **F5**

3. Call the macro from other procedures

Chapter 3: *Using Instructions*

The computer program consists of a series of organized instructions. These instructions normally are coded by human programmers or generated by software tools.

Macro recorder is the Excel component that observes Excel user actions then translates these actions into sequence of computer instructors.

Besides being generated by macro recorder, macros also possibly coded by human developers. Techinically, macros are just a subset of VBA programming. In fact, there are many other aspects of VBA programming that are not about macros.

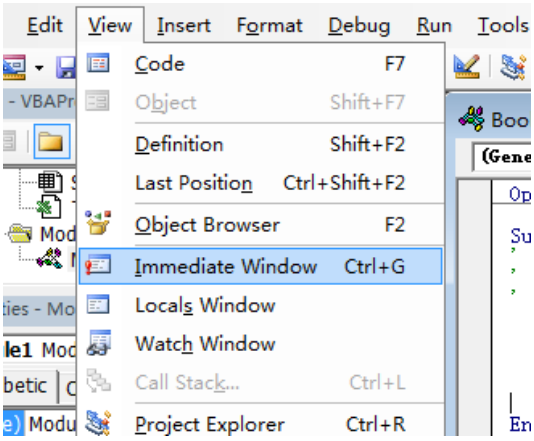
For beginners, to learn VBA programming one of the best approaches is to start with learning instructions.

VBA IDE provides a special window called **Immediate Window**. This is the ideal component to test and learn instruction by instruction.

The Immediate Window

To start the immediate Window, under IDE we can either

1. Press **Ctrl** + **G**
2. Select main menu option **View**, then select **Immediate Window**



A blank window will appear

What is instruction?

Computers are designed to accept instructions. Based on these instructions, the computer will perform tasks as instructed. By organizing a series of these instructions, will form so called **Computer Program**. These instructions must follow special "Grammer" so that computer can understand what the instructions about. Technically this grammer is called **Computer Language Syntax**.

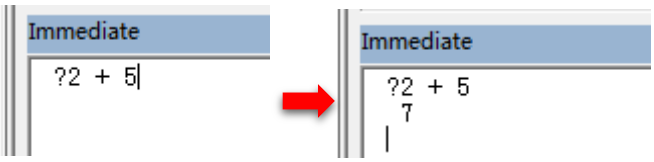
Today we have many types of programming language, VBA is a special programming language where we can instruct Excel to perform actions.

Immediate window is designed to take instruction line by line. To start learning VBA language with immediate window, we need to understand there are two type of instructions that we can use under immediate window:

1. Evaluation Instruction
2. Command

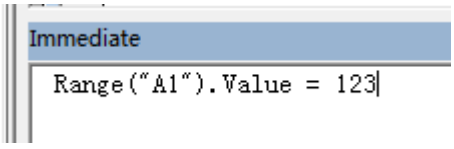
Evaluation instructions

This type of instruction will cause Excel to evaluate the instructions and return value at the line after each instruction. This type of instruction must start with "?" mark. For example,



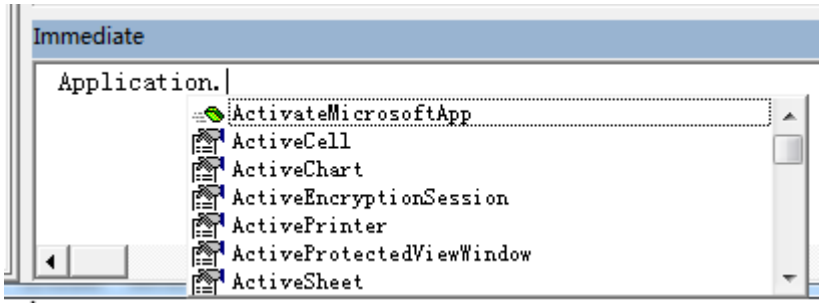
Command Instructions

Unlike evaluation instructions, command instructions do not need to start with "?" mark. It is for instructing Excel to perform some tasks. For example, the following instruction is to set the cell A1 of current active worksheet to value 123:

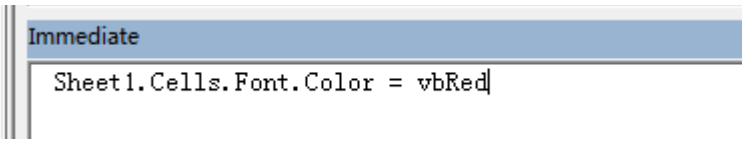


Dealing with Excel VBA objects and their properties

Excel VBA provides abstractions of programming objects. Such as Application, Workbooks, Worksheets (or Sheets), and many others. All these objects have many properties and operations. For example, we can press a "." after the Application name, the **Intellisense** will list all the operations and properties that belong to this object.



Some of these object properties refer to another object. For example,



Lab Exercise:

In this lab exercise, you will learn how to use Immediate Window to:

1) Find out current date and time
2) Change Worksheet name
3) Check the number of worksheets in the workbook
4) Use help system

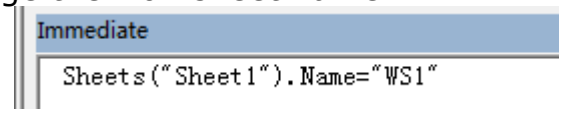
Duration: *15 Minutes*

Instructions:

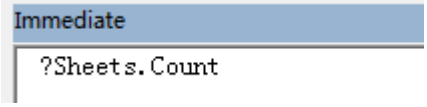
In the Immediate Window, run the following instructions

1) Find out current date and time

2) Change the worksheet name



3) Check the number of worksheets in the workbook



4) ? UCase("Hello")
   Highlight UCase and press **F1**

Chapter 4: *Linking VBA with Excel*

VBA is a programming language for MS Office applications. But when programming Excel with VBA, it is important to know various specific methods to refer entities of Excel from VBA instructions.

This Chapter will cover the following:

- Single cell reference methods
- Range reference methods
- Inter-worksheets reference
- Inter-workbook reference

Single cell reference methods

Below are the methods referring to cell B3 of current active worksheet,

1. [B3] – Short form of Evaluate("B3")
2. Range("B3")
3. Cells(3,2) or Cells(2,"B")

Method-1 is simple, but the object type returned by Evaluate function only can be determined during program runtime, therefore is not adequate for new learner.

Method-2 uses Range object to refer the cell. Intellisence will help programmers during coding.

Method-3 suitable for accessing cell in 2-Dimensional grid.

Range reference methods

Range is a more complex entity. It can used to refer to single cell and multi cells. For example:

- **[B3:E7]** or **Range("B3:E7")** **[B:E]** or **Range("B:E")**
- **[B:B]** or **Range("B:B")**
- **[3:7]** or **Range("3:7")**
- **[3:3]** or **Range("3:3")**
- **[A2:C3,5:6,F:G,J10]** or **Range("A2:C3,5:6,F:G,J10")**
- **Range(Cells(2,3),Cells(6,5))**
- **Range(B3:E7).Cells(3,2)**
- **Range(B3:E7).Cells(3,10)**

Inter-worksheets reference

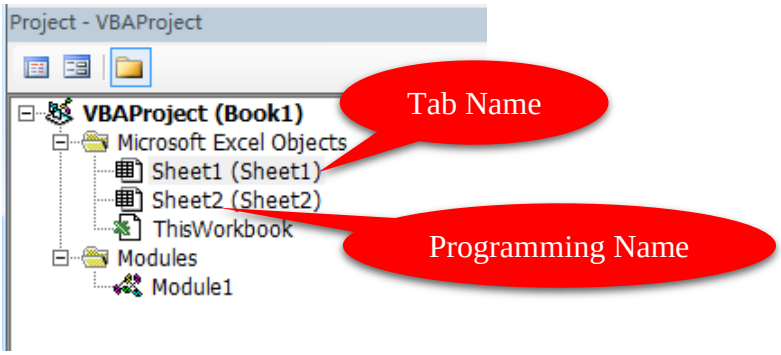
To refer to single cell or multi cells range from non-active worksheets, we must prefix the reference with the either

1. Worksheets (or Sheets) object
    - Sheets(2).Range("A1")
      Cell A1 of second worksheet in the current workbook
    - Sheets("Sheet1").Range("A1")
      Cell A1 of worksheet with tab name "Sheet1"

2. Worksheet Programming name

    - Sheet1.Range("A1")
      Cell A1 of worksheet with programming name "Sheet1"

Every worksheet has two names, one is tab name, and the other is programming name.



To change the tab name, we can use normal interface or IDE, whereas programming name only can be changed from IDE using property window.

Tab names can have space, but programming names must follow standards VBA naming convernsion.

It is advisable to use programming in VBA programming due to the following reasons:

1. Shorter in coding
2. Intellisence helps
3. End users changing tab name will not break the references.

Because worksheet programming names can not be changed when program runtime, in this situation we can use tab name.

Inter-workbook reference

To refer to entity of other open workbook, we can prefix the reference with **Workbooks(*filename or number*)** object followed by ".". Example:

```
Workbooks("Test.xlsm").Sheets(1).Cell(3,2) = 123
```

Lab Exercise:

In this lab exercise, you will learn how to use Immediate Window to refer entities from Excel workbook.

Duration: *15 Minutes*

Instructions
Perform the following tasks:

1) Create a new worksheet with tab name "Module 4"
2) Change its programming name to Mod4
3) Set this new worksheet as current active worksheet
4) In the Immediate Window,
   a. Use evaluates format assign value 100 to cell A1
   b. Use range format assign value 200 to cell A2
   c. Set cell A3 "Formula" property with "=Sum(A1:A2)"
   d. Set range D4 to G10 "Formula" property to "=RandBetween(0,100)"
5) Switch to normal interface, check the result
6) Set another worksheet active and switch back to IDE
7) In the Immediate Window, find out the cell A3 value of previously created worksheet by using
   a. Worksheets or Sheets object
   b. Worksheet programming name

Chapter 5: *The instructions building block*

Computer programs consist of a series of instructions. These instructions can be up to thousands or even millions. If the complexity of the program increases, it is hard to solve the problem at one go in a single group of instructions. Just like in daily life problem solving, we can break down complex problems to smaller pieces, and further break them down to even smaller ones until they are solvable or manageable. This technique is called **Problem Decompisition Technique**. To support this technique, programming languages normally provide functional buiding blocks.

Example 1:

Write a sample code that can ask for the user's name, and then show the greeting message?

Translate this in to steps:

1) Ask for the user's name
2) Capture the user's name into somewhere
3) Show the user's name in the greeting message

The sample code that implements the steps:

No	Code
1	Option Explicit
2	
3	Sub **Greeting1**()
4	Dim name As String
5	name = **InputBox**("What is your name?")
6	MsgBox "Hi,"& name
7	End Sub

Example 2:

Follow up from example 1 above. What should we do if the user does not provide his/her name?

Basically, there are few things we need to clearify here:

1) How do we know that user does not provide the name?
2) What should we do if the name is not provided?

The sample code that implements the steps:

No	Code
1	Option Explicit
2	
3	Sub **Greeting2**()
4	Dim name As String
5	AskForName:
6	name = **InputBox**("What is your name?")
7	If name = "" Then
8	MsgBox "Sorry, you did not tell me who you are?", vbExclamation
9	GoTo AskForName
10	Else
11	MsgBox "Hi," & name
12	End If
13	End Sub

Example 3:

There are also other possible alternatives in the program logic. For instance, the user can just provide the name with spaces.

How should we consider this situation?

The sample code that implements the steps:

No	Code
1	Option Explicit
2	
3	Sub **Greeting3**()
4	Dim name As String
5	AskForName:
6	name = **Trim**(**InputBox**("What is your name?"))
7	If name = "" Then
8	MsgBox "Sorry, you did not tell me who you are?", vbExclamation
9	GoTo AskForName
10	Else
11	MsgBox "Hi," & name
12	End If
13	End Sub

Well train programmer, you need to take care of all the possible situitions in the program logic.

Program logic is formal (unambiguous) translation of programmer's logical thinking into instructions that can be interpreted by computers. As illustrations, let consider the following examples:

The program flow is determined by the statements/instructions in used. These statements represent the logical thinking of the programmer who creates the program.

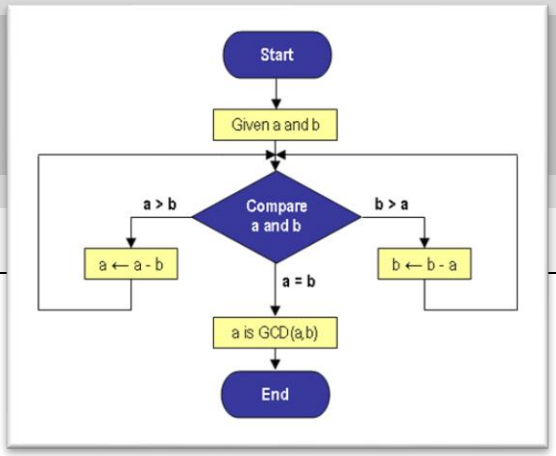
Normally before the programmer starts writing program statements, some form of tools will be used to represent the design of the solution. One of the common tools in use is called **Flowchart**.

The following scenario illustrates the use of Flowchart, and translation into the program code.

Euclidean Algorithm

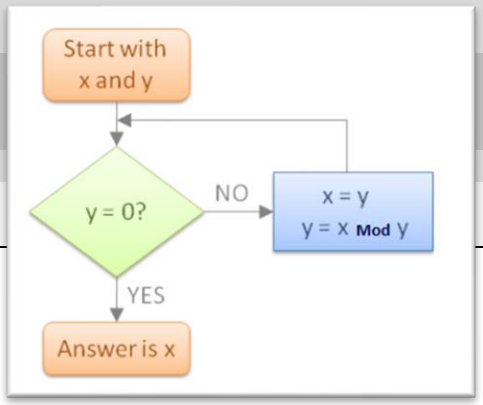
"*The Euclidean algorithm (also called Euclid's algorithm) is an algorithm to determine the **G**reatest **C**ommon **D**ivisor of two integers. Given two natural numbers a and b, check if b is zero. If yes, then a is the GCD. If not, repeat the process using b and the remainder after integer division of a and b.*"

No	Code
1	Option Explicit
2	
3	Function **MyGCD** (ByVal a As Long, ByVal b As Long) As Long
4	While (a <> b)
5	If a > b Then
6	a = a - b
7	Else
8	b = b - a
9	End If
10	Wend
11	MyGCD = a
12	End Funcion



Anaother Alternative

No	Code
1	Option Explicit
2	Function My**GCD** (ByVal x As Long, ByVal y As Long) As Long
3	Dim oldX As Long
4	While (y<>0)
5	oldX = x
6	x = y
7	y = oldX Mod y
8	Wend
9	MyGCD = x
10	End Funcion



When the number of steps in solving problem increases, we can break the program statements to multiple building blocks.

For example, in arithmetic and number theory, the **L**east **C**ommon **M**ultiple (also called the lowest common multiple or smallest common multiple) of two integers x and y, usually denoted by LCM(x, y), is the smallest positive integer that is a multiple of both x and y.

You are required to write a program to input 2 positive integer values. The program should be able to find out the LCM of the values.

Consider the 2 versions of code below

Alternative-1

```
1  Option Explicit
2  Private Sub BadSub()
3    Dim x As Long
4    Dim y As Long
5    Dim oldX As Long
6    Dim x2 As Long
7    Dim y2 As Long
8    Dim gcd As Long
9    Dim lcm As Long
10
11   x =
12 CLng(InputBox("x="))
13   y =
14 CLng(InputBox("y="))
15   x2 = x
16   y2 = y
17   While y2<>0
18     oldX = x2
19     x2 = y2
20     y2 = oldX Mod y2
21   Wend
22   gcd = x2
23   lcm = (x * y) / gcd
     MsgBox "LCM=" & lcm
   End Sub
```

Alternative-2

```
1  Option Explicit
2  Public Function MyLCM(ByVal x As_
3          Long, ByVal y As Long) As Long
4    MyLCM = (x * y) / MyGCD(x, y)
5  End Function
6
7  Private Sub GoodSub()
8    Dim x As Long
9    Dim y As Long
10
11   x = CLng(InputBox("x="))
12   y = CLng(InputBox("y="))
13   MsgBox "LCM=" & MyLCM(x, y)
14 End Sub
```

There are few observations:

1) Both alternatives of code can solve the same problem
2) Alternative-1 is shorter and easier to understand
3) Alternative-1 makes use of existing function MyGCD (assume created some way in the project).
4) Alternative-1 introduces new MyLCM() function that canbe used by others

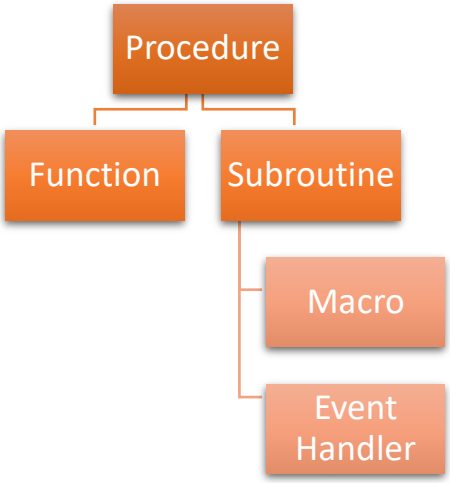
Alterntive-1 places all the statements in a single building block. This version of code although can solve the problem, but it is not a good way of writing program. It violated a very important software development principle called **High Cohesion**. Which means the program entity is doing too many unrelated tasks.

Alternative-2 is higer cohesive, therefore it is more Reusable.

The Procedure Concept

Procedures are the code building blocks in the all the modules (Module, User Form, Class Module, and Object model).

There are two types of procedures:



Procedures (Subroutines)

Procedure that has no return value. It consists of a group of instructions to accomplish some tasks.

Procedure declaration must be in the following form:

```
[Public | Private] Sub <Subroutine Name>(<parameter list>)
        :
    <Instructions>
        :
End Sub
```

Macro in fact is a special form of subroutine (No parameter and with **Public** accessibility).

Procedures (Functions)

It is a procedure that returns value. It can be used in expressions and formulas.

```
[Public | Private] Function <Function Name>(<parameter list>)
        :
    <Instructions>
        :
End Function
```

Procedures (Event Handlers)

Event Handler is a special form of subroutine that is used extensively in User Form and Document Object Model to support the event driven model.

Pre-mature terminations with Exit keyword

When the End statement of procedure (End Sub or End Function) reached, procedure considered end. But sometime, we want to terminate the procedure before reached the End statement, we can use **Exit keyword**. For subroutine use **Exit Sub** and for function use **Exit Function**.

```
[Public | Private] Sub <Subroutine Name>(<parameter list>)
        :
    … Exit Sub
        :
End Sub
```

```
[Public | Private] Function <Subroutine Name>(<parameter list>)
        :
    … Exit Function
        :
End Function
```

Grouping instructions using with statement

Using **With** statement can reduce repetation in coding and maintenance easier.

Before	After

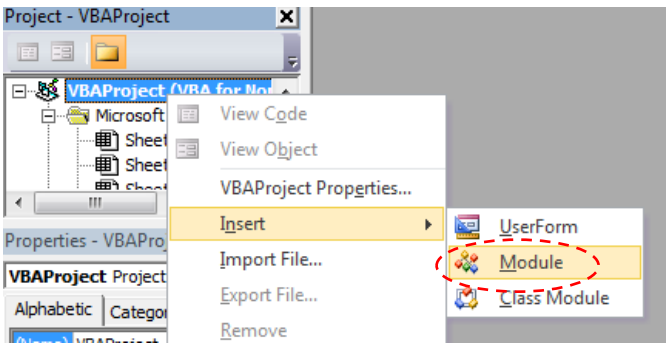
Sub **Greeting3**()
  [A1].Value = 123
  [A1].Font.Color = vbRed
  [A1].Font.Bold = True
  [A1].Interior.Color = vbYellow
End Sub
	Sub **Greeting3**()
	  With [A1]
	  .Value = 123
	  With .Font
	  .Color = vbRed
	  .Bold = True
	  End With
	  .Interior.Color = vbYellow
	  End With
	End Sub

Lab Exercise:

In this lab exercise, you will learn how to create procedures.

Duration: *60 Minutes*
Instructions:

1) Switch to VBE (If you are not yet).
2) Create new code module



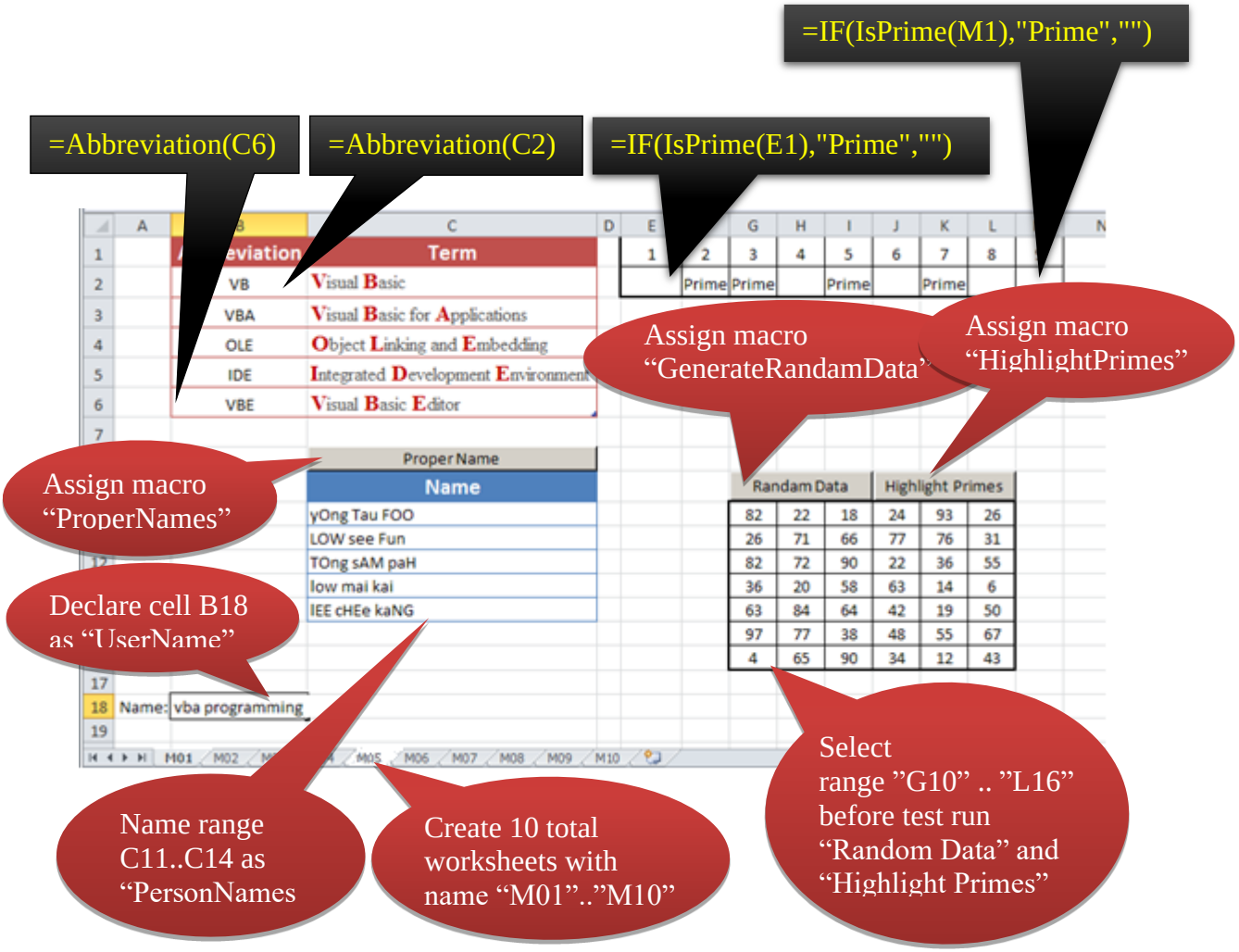
3) Rename it to "M05".
4) Key in the following code in the module M05's code editor

No	Code
1	Option Explicit
2	
3	'---------- Example 1 ----------
4	Function **Abbreviation**(s As String, Optional ByVal lower As Boolean=False) As String
5	Dim sC As String
6	Dim sItem
7	For Each sItem In **Split**(**Trim**(s)," ")
8	sC = **Left**(sItem,1)
9	If (sC<>"") And (lower Or ((sC >= "A") And (sC <= "Z"))) Then
10	Abbreviation = Abbreviation & sC
11	End If

```
12    Next
13  End Function
14
15  '---------------------- Example 2 ----------------------------------------------------
16  Function IsPrime(ByVal n As Integer) As Boolean
17    Dim x As Integer
18    For x=2 To n
19      If(n Mod x)=0 Then Exit For
20    Next x
21    IsPrime = (x=n)
22  End Function
23
24  '---------------------- Example 3 ----------------------------------------------------
25  Sub ProperNames()
26    Dim r As Range
27    Dim c
28    Set r = Range("PersonNames")
29    For Each c In r.Cells
30      c.Value=Excel.WorksheetFunction.Proper(c.Value)
31    Next c
32  End Sub
33
34  '---------------------- Example 4 ----------------------------------------------------
35  Sub GenerateRandamData()
36    Dim c
37    Selection.Interior.Pattern = xlNone
38    For Each c In Selection.Cells
39      c.Value = Round(Rnd() * 100, 0)
40    Next c
41  End Sub
42
43  Sub HighlightPrimes()
44    Dim r As Range
45    Dim c
46    Set r = Selection
47    For Each c In r.Cells
48      If IsNumeric(c) Then
49        If IsPrime(c) Then c.Interior.Color = vbRed
50      End If
51    Next c
52  End Sub
```

Line#	Notes
3	Single quote mark (‘) is for comment. Don’t miss it
	Number of hyphen (-) is not crucial; Just serve as seperators
7	There is a space character in between 2 double quote marks (“ ”)
8	The second arguments of Left(sItem,1) is 1 (One) not ‘L’ or ‘I’
9	sC<>””. There isn’t anything in between s double quate marks

5) Prepare your worksheets for testing the code. Based on the hints, fill in formula for the rest of the cells as needed.



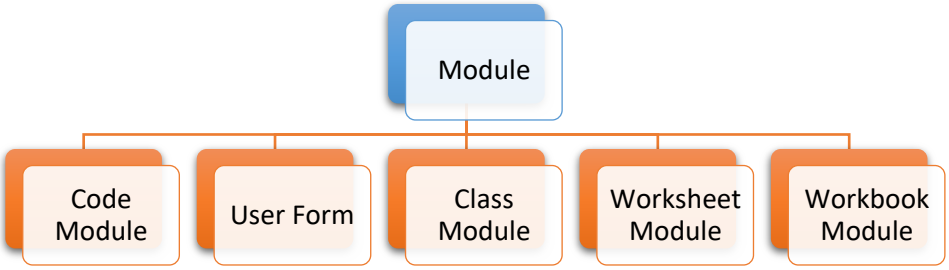
6) From the project explorer, double click on the worksheet M05.
7) In the code editor, prepare the following code:

No	Code
1	Option Explicit
2	
3	Private Sub **Worksheet_Change**(ByVal Target As Range)
4	If Not Application.Intersect(Target,[UserName]) Is Nothing Then
5	Application.EnableEvents = False
6	[UserName] = **UCase**([UserName])
7	Application.EnableEvents = True
8	End If
9	End Sub

--- End of Lab ---

Chapter 6: *Modules*

There are 5 types of modules in Excel VBA.

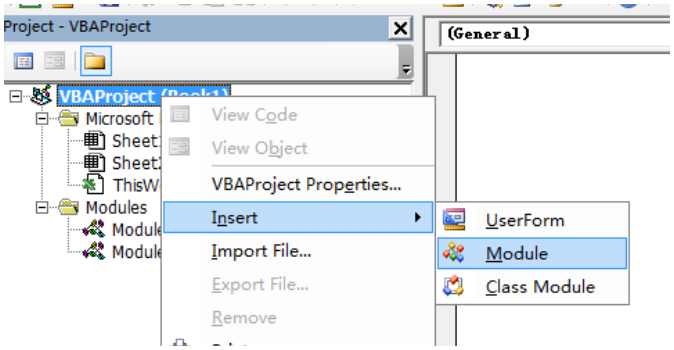


Why is module needed?

Different modules in Excel VBA programming serve different purposes. It provides basic unit of reuse, hiding complexity, and allows object-based coding.

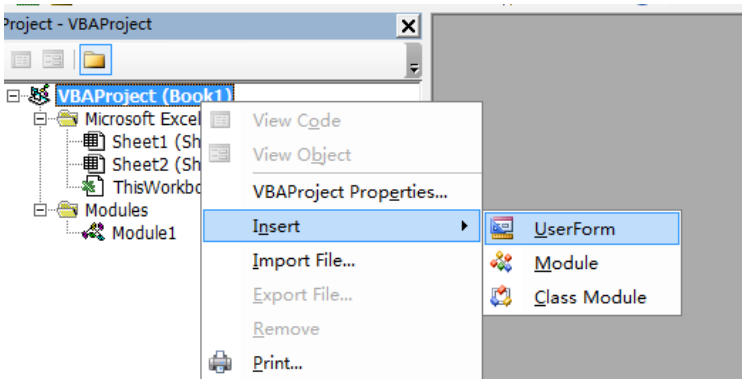
Code Module

These modules are normally used to declare reusable subroutines (including macros), functions, variables, user types and constants for the entire project. To add these modules, we can use project explorer:



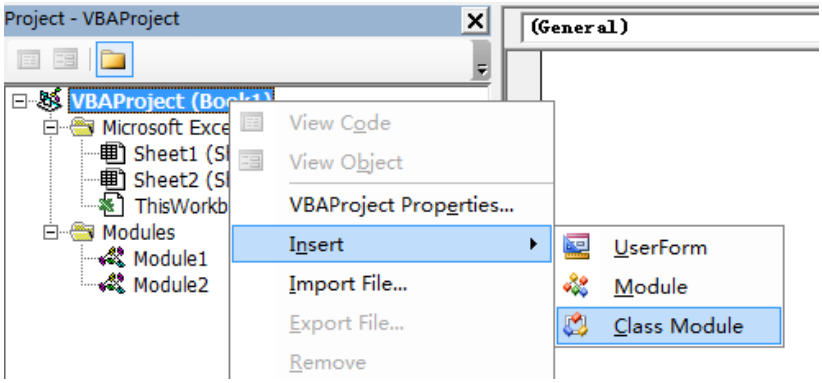
User Form in brief

User for provides alternative interface to Excel. It is commonly used for creating custom dialog boxes. In user form we will use event-based programming. To insert new User Form:



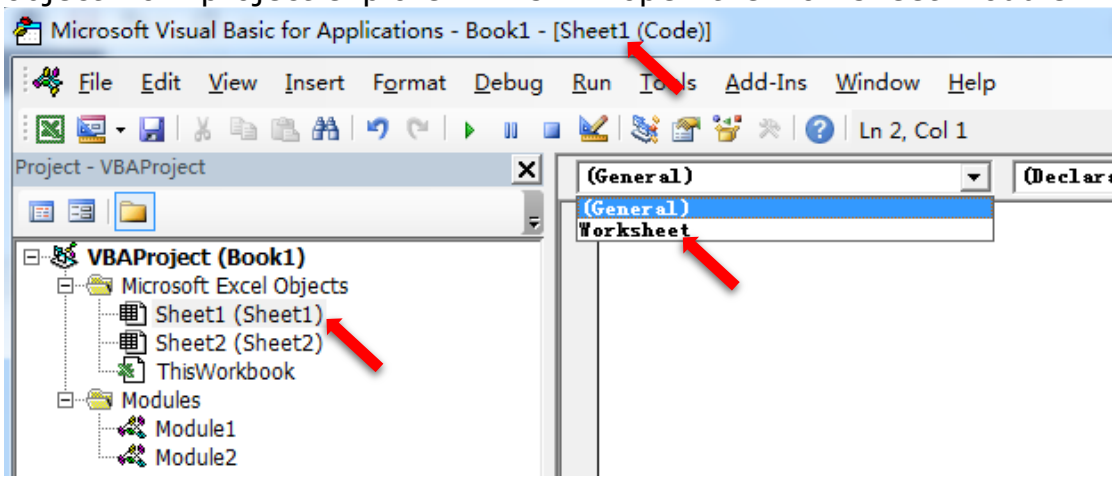
Class module in brief

The class module is a bit more technical for beginers. It is for creating Abstract Data Types. To Insert class Class Module:



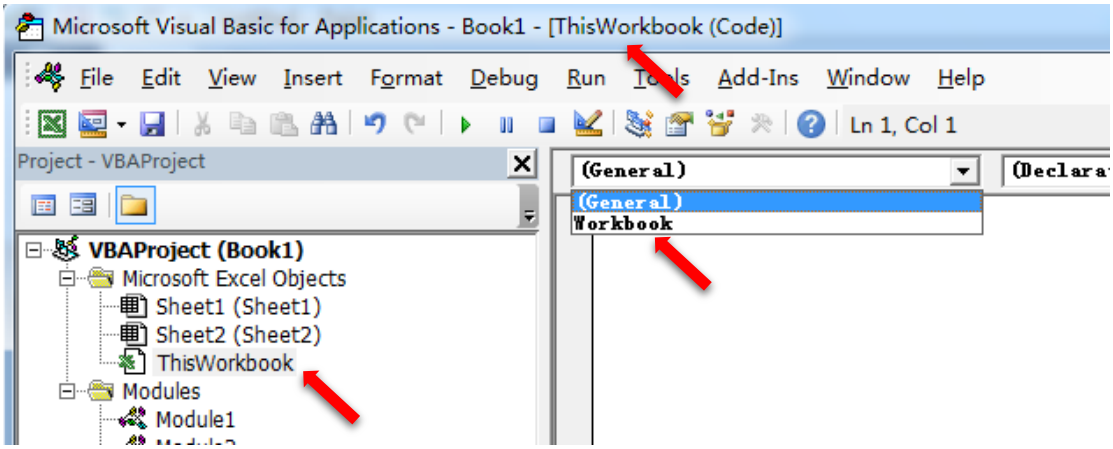
Worksheet module in brief

VBA allows individual worksheets to reply to different events. To start the worksheet module, simply double click on the worksheet object from project explorer. This will open the worksheet module.



Workbook module in brief

Unlike the worksheet level programming, for the Excel workbook, there is only one workbook module. Simply double click on the workbook object from project explorer will open the module editor.



Besides handling workbook level events, in this module we can declare worksheet level event handlers shared by all worksheets.

Procedures scoping

Entity declared in the module with **Private** keywords only can be referred by procedures in the same module only. **Public** keywords are to allow entities declared referred all in the project. All the procedure is public by default. Meaning

```
Sub MyTestSub1()
   MsgBox "MySub1"
End Sub
```

Same as

```
Public Sub MyTestSub1()
   MsgBox "MySub1"
End Sub
```

Macros must be public.

Dealing with ambiguities

Code module name normally is not required when referring to its public entities. It is needed only when there exists ambiguity in naming.

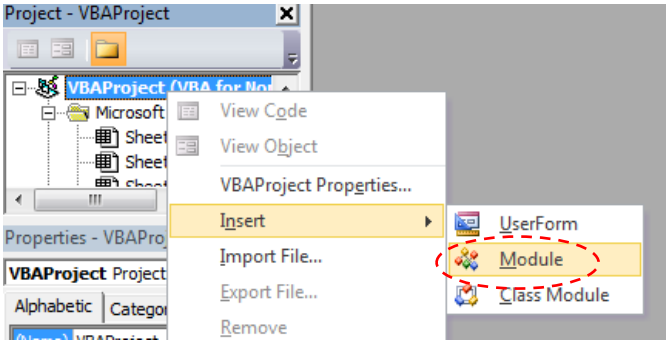
Lab Exercise

In this lab exercise, you will learn how to hide procedures by using special keyworks in VBA.

Duration: *20 Minutes*

Instructions:

1) Switch to VBE (If you are not yet).
2) Create code module6



3) Key in the following code in the module6's code editor

No	Code
1	Option Explicit
2	
3	Dim myVar1
4	Private myVar2
5	Public myVar3
6	
7	Private Sub **MyTestSub1**()
8	MsgBox "MySub1"
9	End Sub
10	
11	Public Sub **MyTestSub2**()
12	MsgBox "MySub2"
13	End Sub
14	
15	Sub **MyTestSub3**()
16	MsgBox "MySub3"
17	End Sub
18	
19	Public Sub **ShowVariable**()
20	MsgBox "myVar1=" & myVar1 & vbCrLf & _
21	"myVar2=" & myVar2 & vbCrLf & _
22	"myVar3=" & myVar3
23	End Sub

4) Turn on the immediate window. Follow the demo steps of instructor.

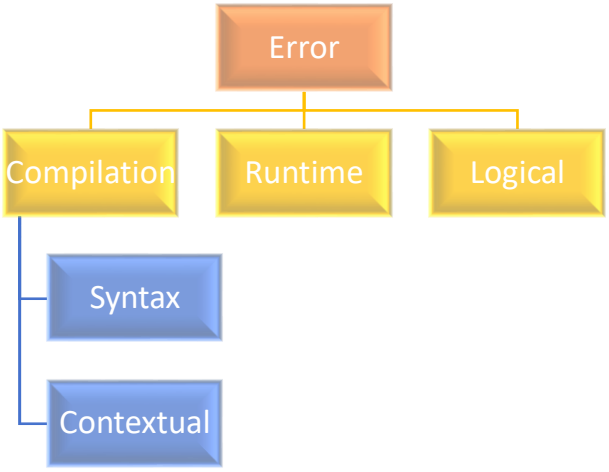
--- End of Lab ---

Chapter 7: *When you make mistake…*

Humans tend to make mistakes. No exception in programming. It is common for programmers to make mistakes during the coding process. We call these mistakes made **Faults** or **Errors**.

Type of errors

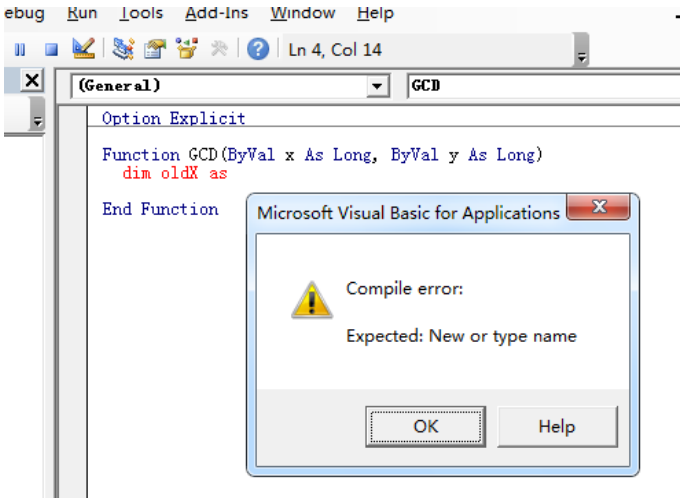
The types of errors in Excel VBA programming.



Dealing with compilation errors

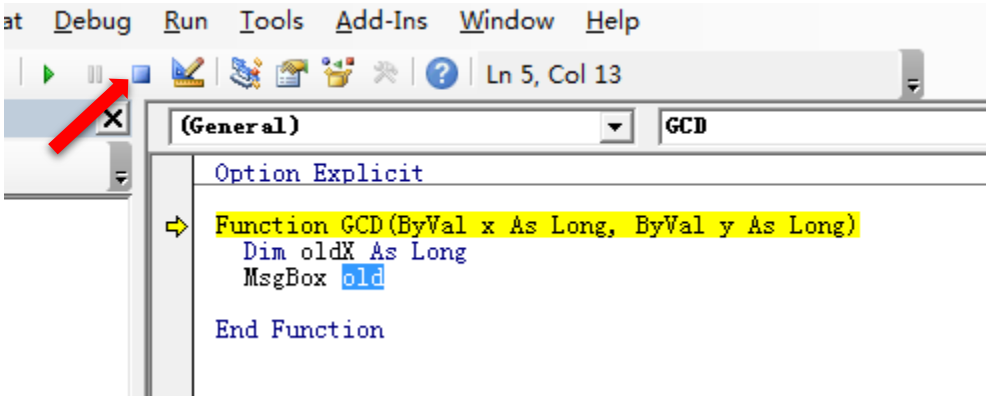
Programming language is a communication tool for programmers to give instructions to computers. The computer then will follow these instructions to complete the tasks instructed.

To avoid any communication problem between humans and computers, programming must be formal. This is to eliminate the ambiguity that might cause by human in giving instructions. Programmers must give instructions by obeying the language "grammar" imposed. This grammar is called Syntax in the context of programming. Failure in obeying the syntax where cause **Syntax Error**.This error will be detected during compile time by the compiler. Therefore, sometimes it is also called **Compilation Error**.



The statement causing error will be highlighted with red. Just answer the dialog box, then fix the problem and continue coding.

Some of these errors will cause yellow color indicator. Press **Reset** button and fix the program.



Referring to entity out of scope also can be detected during compile time. But this is considered as **Contextual Error**.

When compilation occurs, normally the editor will stop us with dialog box:

The compilation errors are the simplest among the rest. In fact, the VBA editor can detect such errors while programmer writing their code.

Dealing with runtime errors

After the program passed the compilation, the program was still not error free. Some of the mistakes only can be encountered during program running. When the program crashes due to a runtime situation, it is called **Runtime Errors**. This type of errors can be handled by using special error handling techniques (Only cover in more advanced VBA course)

Dealing with logical errors

Even if the program passed the runtime test, it still could cause error. Typically, the program produces wrong results. This type of error is called **Logical Errors**.

This is the most difficult and normally most hampful error. Only can be discovered with proper software testing process.

The debugger and debugging process

When we encounter runtime or logical errors, we can make use of the IDE Debugging feature to find the root of problem.

The IDE includes a software component called **Debugger** to assists the programmer to detect the runtime and logical errors. The programmer can mark the line of code where the debugger will pause and wait for the decision of the programmer during debugging mode. This is called **Breakpoints**. When the debugger pauses the program, programmer normally can start a **Watch Window**. In the watch window, programmers can add expressions, such as the variable names to observe the state values of the variables. Then programmers can **Step-Into** or **Step-Over** the code to study the code execution line by line until the problem is found.

With this debugging process, programmer can normally find out the following:

1) Invalid assignment of the value to variables
2) Unexpected execution flow of the program

Lab Exercise:

In this lab exercise, you will learn how to design program logic in problem solving.

Duration: *45 Minutes*

The Challenge:

"*Given any positive integer number, find all the prime factors*"

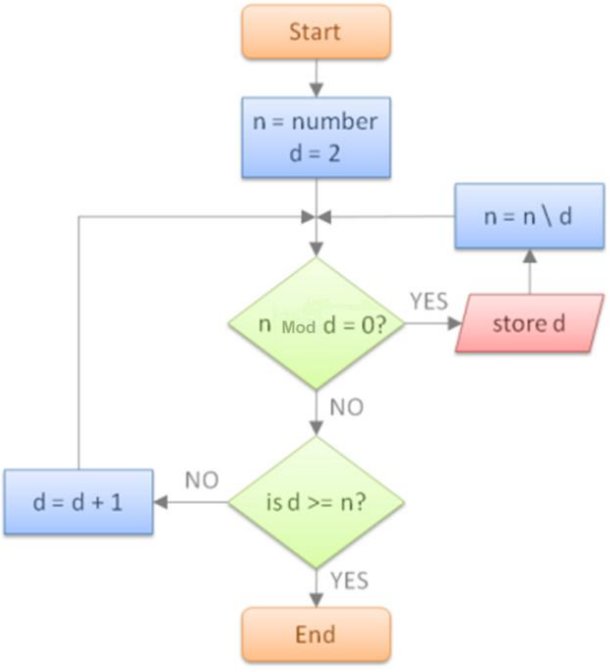
For example, the prime factors of 13195 are 5, 7, 13 and 29. Meaning

13195 = 5 x 7 x 13 x 29.

Consider how we might use this process to find the prime factors of 140.

140	Is 140 evenly divisible by 2? Yes! Remember 2 and divide 140 by 2.
2 x 70	Is 70 evenly divisible by 2? Yes! Remember 2 and divide 70 by 2.
2 x 2 x 35	Is 35 evenly divisible by 2? No, how about 3? No. 4? Nope. 5? Yes! Remember 5 and divide 35 by 5.
2 x 2 x 5 x 7	And we're done!

The Algorithm (Flowchart):

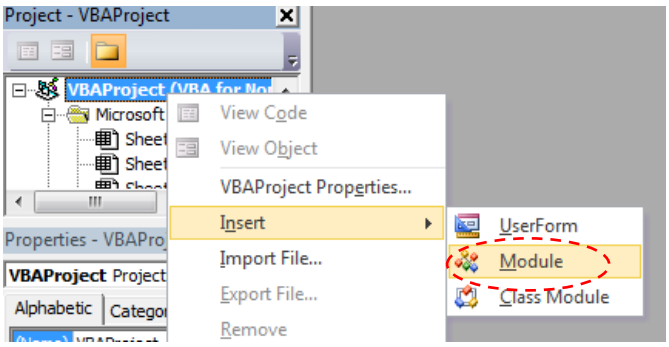


Notes:

1) n Mod d means the remainder after integer division of n by d.
2) n = n \ d means the n will take the result of integer devision of n by b

Instructions:

1) Switch to VBE (If you are not yet).
2) Create code module7



3) Key in the following code in the module7's code editor

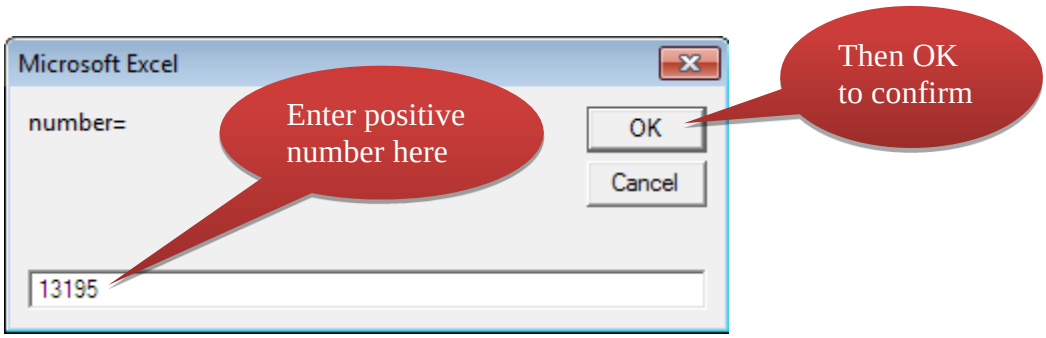
No	Code
1	Option Explicit
2	
3	Sub **PrimeFactors**()
4	Dim n As Long
5	Dim d As Long
6	Dim s As String
7	n = **CLng**(**InputBox**("number="))
8	d = 2
9	Start:
10	If (n Mod d)=0 Then
11	s = s & vbCrLf & d
12	n = n \ d
13	GoTo Start
14	Else
15	If d >= n Then
16	GoTo Finish
17	Else
18	d = d + 1
19	GoTo Start
20	End If
21	End If
22	Finish:
23	MsgBox s
24	End Sub

Notes:

- Line 12: Use operator "\" for integer devision
- The label at line 9 and 22 must end with ":" symbol (colon). It is for the purpose of Goto statements.
- The use of Goto statement is discouraged (More discussion later). In this lab we use it for the purpose of matching the flowchart logical flow.

4) Run the subroutine.

5) Key in the positive integer number when the dialog box appears.



6) The answer is: 5,7,13,29
7) Intentionally change the code at line 15 to `If d > n Then`
8) Try to run the debugging process to find the problem.

--- End of Lab ---

Chapter 8: *The Variables*

The state of the program is determined by the data value during runtime.

Data typical stored under different segments of the runtime memory:

Segment Type	Example	Characteristic
Data Segment	1) Public/Private variable declared outside the procedures 2) Static Variable	Longer life
Stack	1) Variable declare with Dim in the procedure 2) Parameters	Shared. Will be destroyed at the end of procedure call.
Heap	Dynamic Objects	Shared. Managed.

Why are variables needed?

Variables are the name storage that holds runtime values of the program. It has the following facets:

1) Name
2) Data Type
3) Visibility and Scope
4) Life Cycle

Variables can be declared inside and outside the procedures.

Basic Data Types

By providing type to data in the program means giving the following information to the system:

1) The storage size needed by the data
2) Valid type of data
3) The value ranges the data can hold
4) Valid operations on data

All this information is important for the system to produce a more effective system and detect any inconsistency caused by the programmer.

Sample Declaration:

```
Dim iVar As Integer
```

Data Types			
Type	**Size (Bytes)**	**Range**	**Default Value**
Boolean	2	True or False	False
Byte	1	0 to 255	0
Currency	8	–922,337,203,685,477.5808 to 922,337,203,685,477.5807	0
Date	8	1 January 100 to 31 December 9999	30 Dec 1899 00:00:00
Decimal	12	No decimal point: +/– 79,228,162,514,264,337,593,543,950,335 With up to 28 decimal points: +/– 7.9228162514264337593543950335	0
Double	8	**-ve:**-1.79769313486231570E+308 -4.94065645841246544E-324 **+ve:** 4.94065645841246544E-324 to 1.79769313486231570E+308	0
Integer	2	–32,768 to 32,767	0
Long	4	–2,147,483,648 to 2,147,486,647	0
Object	4	Any object reference	Nothing
Single	4	**-ve:** –3.402823E38 to –1.401298E-45 **+ve:** 1.401298E-45 to 3.402823E38	0
String (fixed length)	Length of string	1 to 65,400 characters	Spaces
String (variable length)	10 + length of the string	0 to 2 billion characters	""
User-defined	Sum of storage size of the individual elements	Same range as data type of individual elements	The default value of the individual elements
Variant (character)	22 bytes + length of string	Same as variable length string	Empty
Variant (numeric)	16	Same as Double	Empty

Variable declaration and shorthand

When declaration without mentioned type, by default means variant:

```
Dim V          Same as    Dim V As Variant
```

But here are few special shorthands

```
Dim V%         Same as    Dim V As Integer
Dim V$         Same as    Dim V As String
Dim V#         Same as    Dim V As Double
```

Variable scoping and life cycle

Listed below are the keywords for declaration:

Declaration Statements	
Keyword	**For**
Const	Declares a constant
Dim	Declares a local variable at procedure level
Private	Declares the procedure or variable to have scope only in the module in which it is defined
Public	Declares a procedure or variable to have scope in the module and project in which it is defined and, if declared in an object module, to have scope outside the current project
ReDim	Declares a dynamic array variable
Type...End Type	Declares a user-defined type

Variable initialization

All variables are declared with its type of default value. But sometimes we want the variables with other initial values.

The following is not allowed in VBA:

```
Dim V As Integer=10
```

However, we can use multiple statement format:

```
Dim V As Integer : V=10
```

Option Explicit directive

VBA still inherit some legacy BASIC language characterics. It allows auto variable declaration. This means the programmer has no need to declare the variables to use them. When firsttime the variable is first use by program, the variable will be declared automatically as type variant.

This feature seems handy, but it come with some serious cost in modern programming:

1) If the programmer typed wrongly the name, it is easily causing logical error.

2) The use of variant type is costly

To avoid auto variable declaration, we can place statement **Option Explicit** as the first line of any module. With this line of statement, compiler with not perform the auto variable declaration for that module.

It is always a good practice to start any module with this **Option Explicit** statement.

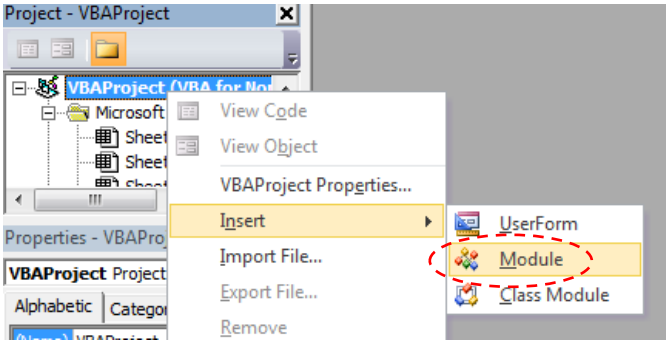
Lab Exercise:

In this lab exercise, you will learn how to deal with various storage classes and variable scoping via accessibility keywords in VBA.

Duration: *30 Minutes*

Instructions:

1) Switch to VBE (If you are not yet).
2) Create code module8



3) Key in the following code in the module8's code editor

No	Code
1	`Option Explicit`
2	
3	`Public Sub TestOutOfRange()`
4	`  Dim iVar As Integer`
5	`  Dim lVar As Long`
6	`  iVar = 100000  'This will cause overflow!`
7	`  lVar = 100000  'This is fine`
8	`End Sub`
9	
10	`Public Sub TestStatic()`
11	`  NormalSub`
12	`  NormalSub`
13	`  NormalSub`
14	`  SubWithStaticVar`
15	`  SubWithStaticVar`
16	`  SubWithStaticVar`
17	`  StaticSub`
18	`  StaticSub`
19	`  StaticSub`
20	`End Sub`
21	
22	`Private Sub NormalSub()`
23	`  Dim n%`
24	`  n = n + 1`
25	`  Debug.Print "NormalSub" & vbTab & n`
26	`End Sub`
27	
28	`Private Sub SubWithStaticVar()`
29	`  Static n%`
30	`  n = n + 1`
	`  Debug.Print "SubWithStaticVar" & vbTab & n`

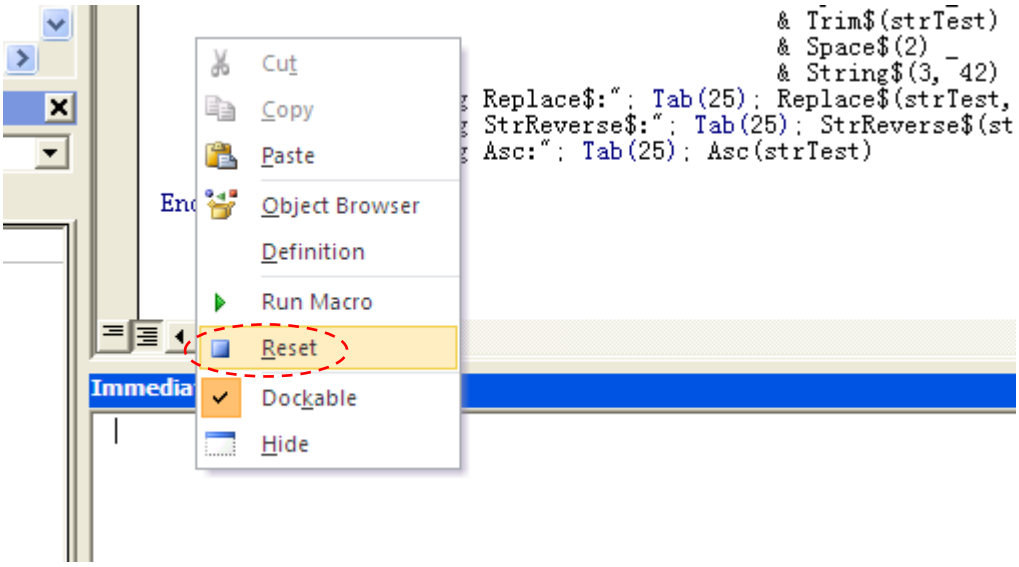
```
31  End Sub
32
33  Static Sub StaticSub()
34    Dim n%
35    n = n + 1
36    Debug.Print "StaticSub" & vbTab & n
37  End Sub
38
```

4) Turn on the immediate window.

5) Execute Subroutine **TestOutOfRange()**. Discuss with instructor your observation

6) Execute Subroutine **TestStatic()**. Discuss with instructor your observation

7) Before you want to try again, you need to reset the immediate window by right-click Reset.



--- End of Lab ---

Chapter 9: *Useful VBA Native Functions*

Besides can borrow functions from Excel, VBA has various useful native functions. Learning how to use these functions can make our coding much simpler.

MsgBox function

It is one of the commonly used dialog boxes, waits for the user to click a button, and returns an Integer indicating which button the user clicked.

Syntax:

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

Parameter	Description
prompt	Required. String expression displayed as the message in the dialog box. The maximum length of prompt is approximately 1024 characters, depending on the width of the characters used. If prompt consists of more than one line, you can separate the lines using a vbNewLine constant between each line.
buttons	Optional. For type of buttons, icons, and default button
title	Optional. If omitted, will shows application name
helpfile	Optional. String expression that identifies the Help file to use to provide context-sensitive Help for the dialog box. If helpfile is provided, context must also be provided.
context	Optional. Numeric expression that is the Help context number assigned to the appropriate Help topic by the Help author. If context is provided, helpfile must also be provided.

If we use MsgBox for acknowledgement, we normally ignore its return value.

InputBox function

Displays a prompt in a dialog box, waits for the user to input text or click a button, and returns a String containing the contents of the text box. Syntax:

InputBox(*prompt*[, *title*] [, *default*] [, *xpos*] [, *ypos*] [, *helpfile*, *context*])

Parameter	Description
prompt	Required. String expression displayed as the message in the dialog box. The maximum length of prompt is approximately 1024 characters, depending on the width of the characters used. If prompt consists of more than one line, you can separate the lines using a vbNewLine constant between each line.
title	Optional. If omitted, will shows application name
default	Optional. String expression displayed in the text box as the default response if no other input is provided. If you omit default, the text box is displayed empty
xpos	Optional. Numeric expression that specifies, in twips, the horizontal distance of the left edge of the dialog box from the left edge of the screen. If xpos is omitted, the dialog box is horizontally centered.
ypos	Optional. Numeric expression that specifies, in twips, the vertical distance of the upper edge of the dialog box from the top of the screen. If ypos is omitted, the dialog box is vertically positioned approximately one-third of the way down the screen.
helpfile	Optional. String expression that identifies the Help file to use to provide context-sensitive Help for the dialog box. If helpfile is provided, context must also be provided.
context	Optional. Numeric expression that is the Help context number assigned to the appropriate Help topic by the Help author. If context is provided, helpfile must also be provided.

Number functions

Number functions deal with numbers in VBA programming.

Math Functions	
Function	**Returns:**
Abs(x)	the absolute value of x
Atn(x)	the trigonometric arctangent of x (in radians)
Cos(x)	the trigonometric cosine of x (in radians)
Exp(x)	exponential function e^x
Fix(x)	the integer portion of x
Int(x)	The integer portion of x, except that if x is negative, it will return the next smallest number. For example, Int(-4.3) would return –5, not –4 as you might expect. To get the integer portion of a number, the Fix function will always produce the expected result.
Log(x)	natural logarithm of x (base e)
Round(x, y)	x rounded to y decimal places
Rnd	a random number less than 1 but greater than or equal to zero
Sgn(x)	-1 if x is negative, 0 if x is 0, 1 if x is positive
Sin(x)	the trigonometric sine of x (in radians)
Sqr(x)	the square root of x
Tan(x)	the trigonometric tangent of x (in radians)

Financial Functions	
To do this:	**Use one of these functions:**
Calculate depreciation.	DDB, SLN, SYD
Calculate future value.	FV
Calculate interest rate.	Rate
Calculate internal rate of return.	IRR, MIRR
Calculate number of periods.	NPer
Calculate payments.	IPmt, Pmt, PPmt
Calculate present value.	NPV, PV

String functions

Below are some common string functions:

Function	Purpose
Len(string)	Returns a Long containing the length of the specified string
Mid(string, start[, length])	Returns a substring containing a specified number of characters from a string.
Left(string, length)	Returns a substring containing a specified number of characters from the beginning (left side) of a string.
Right(string, length)	Returns a substring containing a specified number of characters from the end (right side) of a string.
UCase(string)	Converts all lowercase letters in a string to uppercase. Any existing uppercase letters and non-alpha characters remain unchanged.
LCase(string)	Converts all uppercase letters in a string to lowercase. Any existing lowercase letters and non-alpha characters remain unchanged.
InStr([start,] string1, string2 [, compare])	Returns a Long specifying the position of one string within another. The search starts either at the first character position or at the position specified by the start argument and proceeds forward toward the end of the string (stopping when either string2 is found or when the end of the string1 is reached).
InStrRev(string1, string2[, start, [, compare]])	Returns a Long specifying the position of one string within another. The search starts either at the last character position or at the position specified by the start argument and proceeds backward toward the beginning of the string (stopping when either string2 is found or when the beginning of the string1 is reached).
String(number, character)	Returns a string containing a repeating character string of the length specified.
Space(number)	Returns a string containing the specified number of blank spaces.
Replace(expression, find, replacewith[, start[, count[, compare]]])	Returns a string in which a specified substring has been replaced with another substring a specified number of times.
StrReverse(string)	Returns a string in which the character order of a specified string is reversed
LTrim(string)	Removes leading blank spaces from a string.
RTrim(string)	Removes trailing blank spaces from a string.
Trim(string)	Removes both leading and trailing blank spaces from a string.
Asc(string)	Returns an Integer representing the ASCII character code corresponding to the first letter in a string.
Chr(charcode)	Returns a string containing the character associated with the specified character code.

Date/Time functions

Below are some common date/time functions:

Function	Purpose
Date()	Return current date
DateAdd(interval, number, date)	Returns a Variant (Date) containing a date to which a specified time interval has been added
DateDiff(interval, date1, date2[, firstdayofweek[, firstweekofyear]])	Returns a Variant (Long) specifying the number of time intervals between two specified dates
DatePart(interval, date [,firstdayofweek[, firstweekofyear]])	Returns a Variant (Integer) containing the specified part of a given date.
DateSerial(year, month, day)	Returns a Variant (Date) for a specified year, month, and day.
DateValue(date)	Returns a Variant (Date).
Day(date)	Returns a Variant (Integer) specifying a whole number between 1 and 31, inclusive, representing the day of the month.
Hour(time)	Returns a Variant (Integer) specifying a whole number between 0 and 23, inclusive, representing the hour of the day.
Minute(time)	Returns a Variant (Integer) specifying a whole number between 0 and 59, inclusive, representing the minute of the hour.
Now()	Return current date and time
Second(time)	Returns a Variant (Integer) specifying a whole number between 0 and 59, inclusive, representing the second of the minute.
Time()	Returns a Variant (Date) indicating the current system time.
Year(date)	Returns a Variant (Integer) containing a whole number representing the year.

Format function

Returns a Variant (String) containing an expression formatted according to instructions contained in a format expression. Syntax:

Format(*expression*[**,***format* [**,***firstdayofweek* [**,***firstweekofyear*]]]**)**

Parameter	Description
expression	Required. Any valid expression
format	Optional. A valid named or user-defined format expression
firstdayofweek	Optional. A constant that specifies the first day of the week
Firstweekofyear	Optional. A constant that specifies the first week of the year
ypos	Optional. Numeric expression that specifies, in twips, the vertical distance of the upper edge of the dialog box from the top of the screen. If ypos is omitted, the dialog box is vertically positioned approximately one-third of

	the way down the screen.

Data Type Validation

Sometimes we need to check the actual type of data before further action is taken on to the data. This is especially important if we are using Variant type to hold the data. VBA provides few intrinsic functions for data type validation:

Function	Description
IsDate	Returns a Boolean value indicating whether an expression can be converted to a date. Useful for validating input.
IsNumeric	Returns a Boolean value indicating whether an expression can be evaluated as a number. Can be useful for validating input; but use with caution: if the expression contains the letter "E" or "D", the input argument could be interpreted as a number in scientific notation (thus returning True when you would expect the function to return False).
IsNull	Returns a Boolean value that indicates whether an expression contains no valid data (Null). Usually used with database fields, although a Variant variable can also contain Null (no other intrinsic VB datatype can store Null values).
IsMissing	Returns a Boolean value indicating whether an optional Variant argument has been passed to a procedure.
IsEmpty	Returns a Boolean value indicating whether a Variant variable has been initialized (i.e., if it has ever been assigned a value).
IsObject	Returns a Boolean value indicating whether a Variant variable represents an object.
IsError	Returns a Boolean value indicating whether a Variant variable contains the special value's Error.
IsArray	Returns a Boolean value indicating whether a variable is an array.

You can also use **If TypeOf *<ref. var>* Is *<Class>* Then** to check for the object type.

Lab Exercise:

In this lab exercise, you will learn how to use some of the string manipulation functions.

Duration: *30 Minutes*

Instructions:

1) Switch to VBE (If you are not yet).
2) Create code module9
3) Key in the following code in the module9's code editor

No	Code
1	Option Explicit
2	
3	Sub **TestString**()
4	Dim s As String
5	
6	s = "VBA Programming is Fun"
7	
8	Debug.Print "Using Len:"; **Tab**(25); **Len**(s)
9	Debug.Print "Using Mid$:"; **Tab**(25); **Mid$**(s,3, 4)
10	Debug.Print "Using Left$:"; **Tab**(25); **Left$**(s,3)
11	Debug.Print "Using Right$:"; **Tab**(25); **Right$**(s, 2)
12	Debug.Print "Using UCase$:"; **Tab**(25); **UCase$**(s)
13	Debug.Print "Using LCase$:"; **Tab**(25); **lCase$**(s)
14	Debug.Print "Using Instr:"; **Tab**(25); **InStr**(s,"a")
15	Debug.Print "Using InstrRev:"; **Tab**(25); **InStrRev**(s,"a ")
16	Debug.Print "Using LTrim$:"; **Tab**(25); **LTrim$**(s)
17	Debug.Print "Using RTrim$:"; **Tab**(25); **RTrim$**(s)
18	Debug.Print "Using Trim$:"; **Tab**(25); **Trim$**(s)
19	Debug.Print "Using String$ & Space$:"; **Tab**(25); **String$**(3,"*") _
20	& **Space$**(2) & **Trim$**(s) _
21	& **Space$**(2) & **String$**(3, 42)
22	Debug.Print "Using Replace$:"; **Tab**(25); **Replace$**(s,"a", "*")
23	Debug.Print "Using StrReverse$:"; **Tab**(25); **StrReverse$**(s)
24	Debug.Print "Using Asc:"; **Tab**(25); **Asc**(s)
25	End Sub

--- End of Lab ---

Chapter 10: *Other Useful Basic VBA Entities*

In this Chapter we will learn some of the essential elements of the VBA programming language.

Comments

Comments can provide better readability of your code. But comments will be ignored by the system.

VBA only supports single line comments. You can use "SingleQuote" or Rem keyword for the comment. Under code editor, comments normally appear in green color (Unless you changed the setting).

No	Code
1	Option Explicit
2	
3	'This is comment
4	Rem This is also comment
5	

Today, **Rem** rarely in used due to two reasons:

1) More typing needed
2) Cannot use after statements

Keywords

Keywords, also called **Reserved Words**, are a set of terms used by the language. You cannot use them as names for your programming elements such as variables or procedures. The table below shows some of the common keywords

Common Keywords							
And	As	Boolean	ByRef	Byte	ByVal	Case	Const
Currency	Date	Declare	Dim	Do	Double	Each	Else
End	Enum	Eqv	Error	Exit	For	Function	GoSub
GoTo	If	Imp	In	Integer	Is	Long	Loop
Mod	New	Next	Not	Nothing	Null	Object	Option
On	Optional	Or	ParamArray	Preserve	Private	Public	ReDim
Resume	Return	Select	Single	Static	Step	String	Sub
Then	To	Type	Variant	Wend	While	With	Xor

Notes: *There are more keywords in VBA but omitted here. You will know them when you reach to more advance levels of VBA programming.*

Identifiers

The concept of identifier is for reference to the programming ellements, such as variables, subroutine, function, constant, label, etc.

Declared Element Names

Every declared element has a name, also called an identifier, which is what the code uses to refer to it.

Rules

An element name in VBA must observe the following rules:

- It must begin with an alphabetic character or an underscore (_).
- It must only contain alphabetic characters, decimal digits, and underscores.
- It must contain at least one alphabetic character or decimal digit if it begins with an underscore.
- It must not be more than 1023 characters long.

The length limit of 1023 characters also applies to the entire string of a fully qualified name.

Name Length Guidelines

As a practical matter, your name should be as short as possible while still clearly identifying the nature of the element. This improves the readability of your code and reduces line length and source-file size.

On the other hand, your name should not be so short that it does not adequately describe what the element represents and how your code uses it. This is important for the readability of your code. If somebody else is trying to understand it, or if you yourself are looking at it a long time after you wrote it, suitable element names can save a considerable amount of time.

Constants

Systems declared constants (vb*), such as:

Language Constants	
Constant	**Value**
vbBack	Chr$(8)
vbCr	Chr$(13)
vbCrLf	Chr$(10) & Chr$(13)
vbFormFeed	Chr$(12)
vbLf	Chr$(10)
vbNewLine	Platform-specific
vbNullChar	Chr$(0)
vbNullString	Zero-length string
vbObjectError	–2147221504
vbTab	Chr$(9)
vbVerticalTab	Chr$(11)

The VBA constants can be used for all MS Office applications.

Excel Constants

In Excel VBA we need to know another set of constants only applicable under Excel VBA. These constants start with xl*. For example:

```
Range("A1").Interior.Pattern = xlNone
```

Where the xlNone here is to set the cell A1 background to "No Fill"

Consider the sample code below:

User defined constant

Sample system constants

```
1  Option Explicit
2
3  Private Const PI As Double = 3.14159
4  Function Area(ByVal radius As Double) As Double
5    Area = PI * radius * radius
6  End Function
7
8  Function Circumference(ByVal radius As Double) As Double
9    Circumference = 2 * PI * radius
10 End Function
11
12 Private Sub GetRadius()
13   Dim r As Double
14   r = CDbl(InputBox("The Circle Radius"))
15   MsgBox "Details:" & vbCrLf & _
16             vbTab & "Radius=" & r & vbCrLf & _
17             vbTab & "Area=" & Area(r) & vbCrLf & _
18             vbTab & "Circumference=" & Circumference(r), _
19             vbOKOnly + vbInformation, _
20             "Circle Information"
21 End Sub
```

The elements in Red color and Blue Color in the about sample code are keywords.

Basically, there are 2 main advantages of using constants.

1) The program statements are more **Readable**.
2) The constants can be used in many places in the program. In case we need to change the value represented by the constant, just change at the place we declare it. This is better than **hardcode** the value at many places that will cause maintenance more difficult.

Defining constants

Constants are value declared but cannot be modified. Constants can be declared by programmer by using keyword **Const**. For example:

```
Const PI As Double = 3.14159
```

Selection keyword

Selection is the range object representing current range selection in a worksheet. Since it is a range, we can apply any range operations and refer to the range attributes.

Application object

In the context of Excel VBA, Application object is the abstraction of the entire Microsoft Excel application.

The Application object contains:

- Application-wide settings and options.
- Methods that return top-level objects, such as ActiveCell, ActiveSheet, and so on.

For example, the following statement creates a Microsoft Excel workbook object in another application and then opens a workbook in Microsoft Excel.

```
Set xl = CreateObject("Excel.Sheet")

xl.Application.Workbooks.Open "newbook.xls"
```

ActiveSheet object

Returns an object that represents the active sheet (the sheet on top) in the active workbook or in the specified window or workbook. Returns **Nothing** if no sheet is active.

The following example displays the name of the active sheet.

```
MsgBox "The name of the active sheet is " & ActiveSheet.Name
```

Sheets collection

A collection of all the sheets in the specified or active workbook.

The Sheets collection can contain Chart or Worksheet objects.

The Sheets collection is useful when you want to return sheets of any type. If you need to work with sheets of only one type, see the object topic for that sheet type.

Following example use the Add method to create a new sheet and add it to the collection. The following example adds two chart sheets to the active workbook, placing them after sheet two in the workbook.

```
Sheets.Add type:=xlChart, count:=2, after:=Sheets(2)
```

Workbooks collection

A collection of all the Workbook objects that are currently open in the Microsoft Excel application.

Following example use the Open method to open a file. This creates a new workbook for the opened file. The following example opens the file Array.xls as a read-only workbook.

```
Workbooks.Open FileName:= "Array.xls", ReadOnly:=True
```

Lab Exercise:

In this lab exercise, you will learn how to use and declare constants. Besides, you will learn how to use immediate window and use it to call sub routines and functions.

Duration: *20 Minutes*

Instructions:

4) Switch to VBE (If you are not yet).
5) Create code module10
6) Key in the following code in the module10's code editor

No	Code
1	Option Explicit
2	
3	Private Const PI As Double = 3.14159
4	
5	Private Function **Area**(ByVal radius As Double) As Double
6	Area = PI * radius * radius
7	End Function
8	
9	Private Function **Circumference**(ByVal radius As Double) As Double
10	Circumference = 2 * PI * radius
11	End Function
12	
13	Public Sub **GetRadius**()
14	Dim r As Double
15	r = **CDbl**(**InputBox**("The Circle Radius"))
16	MsgBox "Details:" & vbCrLf & _
17	vbTab & "Radius=" & r & vbCrLf & _
18	vbTab & "Area=" & **Area**(r) & vbCrLf & _
19	vbTab & "Circumferene=" & **Circumference**(r), _
20	vbOKOnly + vbInformation, _
21	"Circle Information"
22	End Sub
23	
24	'This is comment
25	Rem This is also comment

7) Start the immediate window by pressing Ctrl + G
8) In the immediate window try to key in each of the following and then press enter key to execute:

```
? Area(10)
GetRadius
```

--- End of Lab ---

Chapter 11: *The Parameter*

What is parameter?

Instead of creating many similar versions of procedures, we can pass arguments to procedure to handle different states that require similar processing logic. Therefore, many VBA procedures have **Parameters** to accept these argument values.

Optional parameters and techniques to handle default values

In some situations, we can omit arguments when calling the procedures. Therefore, the parameter supposes to accept the argument value must be declared as **Optional**. If no argument value is provided during the procedure call, these parameters must be initialized with **Default** Value.

There are three techniques to handle the default value for optional parameters:

1) Data Type default
2) Absolute default
3) Conditional default

If we declared any parameter as optional, all others parameter to the right must be optional as well. This will explain why all the optional parameters appear at the end of the procedures.

Arbitrary argument support using ParamArray declaration

Perhaps you are aware that in Excel there are functions that can take variable numbers or arguments. Typical examples are Sum, Average, etc.

In VBA we can declare procedures that can accept variable number of arguments by using keyword **ParamArray**.

Parameter passing mechanisms: ByVal Vs. ByRef

When you define a procedure, you have two choices regarding how arguments are passed to it: by reference or by value. When a variable is passed to a procedure by reference, VBA passes the variable's address in memory to the procedure, which can modify it directly. When execution returns to the calling procedure, the variable contains the modified value.

When an argument is passed by value, VBA passes a copy of the variable to the procedure. Then, the procedure modifies the copy, and the original value of the variable remains intact; when execution returns to the calling procedure, the variable contains the same value that it had before being passed.

By default, VBA passes arguments by reference. To pass an argument by value, precede the argument with the **ByVal** keyword in the procedure definition, as shown here:

```
Function SomeProc(strText As String, ByVal lngX As Long) As Boolean
```

If you want to denote explicitly that an argument is passed by reference, you can preface the argument with the **ByRef** keyword in the argument list.

Passing by reference can be useful if you understand how it works. For example, you must pass arrays by reference; you will get a syntax error if you try to pass an array by value. Because arrays are passed by reference, you can pass an array to another procedure to be modified, and then you can continue working with the modified array in the calling procedure.

Named arguments

Consider the following procedure call:

```
fv = MyFV(1000, 0.05F, 10)
```

We can rewrite it as:

```
fv = MyFV(pv:=1000, rate:=0.05F, terms:=10)
```

Where pv, rate, and terms are the parameters for function MyFV.

This format is called **Named Arguments**.

With this Named Arguments, we can rewrite the above statement as:

```
fv = MyFV(rate:=0.05F, terms:=10, pv:=1000)
```

Normally named arguments are commonly used for procedures with many parameters.

There two main advantages by using Named Arguments:

1) Better code self descriptive. Less explicit documentation needed to explain the code
2) Arguments sequence independence. This is very useful when we use named arguments together with optional parameters.

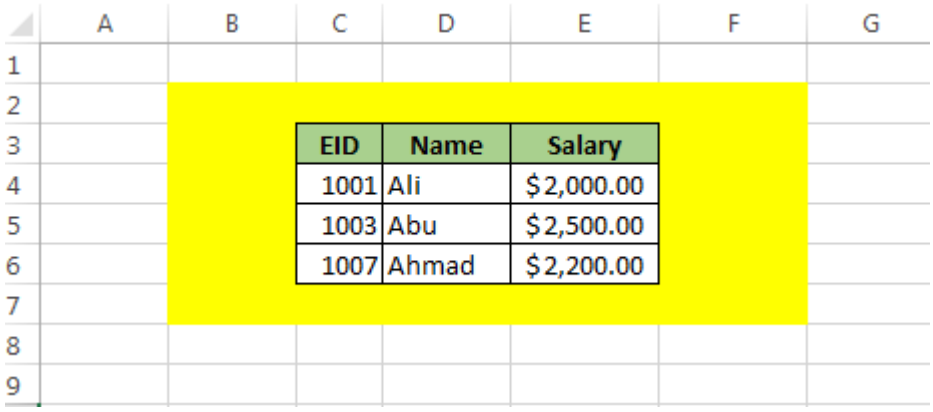
Lab Exercise:

In this lab exercise, you will learn how to write a macro to find the total of employee salary without knowing fixed data address, column sequence, and record numbers. You will learn the concept of **Region** in Excel.

Duration: *45 Minutes*

Instructions:

1) Create a new worksheet and change its tab name to "M11".
2) Prepare the contents as below:



EID	Name	Salary
1001	Ali	$2,000.00
1003	Abu	$2,500.00
1007	Ahmad	$2,200.00

3) Switch to VBE (If you are not yet).
4) Create code module11
5) Key in the following code in the module11's code editor

No	Code
1	`Option Explicit`
2	
3	`Sub FindTotalSalary()`
4	`  Dim rng As Range`
5	`  Dim r%, c%, sum As Currency`
6	
7	`  Set rng = Sheets("M11").Cells.Find(What:="EID", LookAt:=xlWhole, MatchCase:=True)`
8	`  If rng Is Nothing Then`
9	`    MsgBox "Data not found!"`
10	`  Else`
11	`    Set rng = rng.CurrentRegion`
12	`    For c = 1 To rng.Columns.Count`
13	`      If rng.Cells(1, c) = "Salary" Then`
14	`        For r = 2 To rng.Rows.Count`
15	`          sum = sum + CCur(rng.Cells(r, c))`
16	`        Next r`
17	`        MsgBox "The total salary is " & Format(sum, "$#,###.00")`
18	`        Exit Sub`
19	`      End If`
20	`    Next c`
21	`    MsgBox "Salary column not found!"`
22	`  End If`
23	`End Sub`

6) Run the macro FindTotalSalary. Study how it works
7) Try to run the macro again after each of these attemps:

a. Try to move the data around in the worksheet.
b. Insert a new column before the **Salary** column.
c. Insert new column after the **Salary** column.
d. Add new record.

8) Explain why the macro fail if
   a. Change the column title **EID** to **EIDs**.
   b. Change the column title **Salary** to **Salaries**.
   c. Add some data to the cell just on top of the cell with column title **Salary**.

--- End of Lab ---

Chapter 12: *Operators*

What is operator?

Operators is for related operands in the expressions. There are five groups of operators in VBA: Arithmetic, String, Comparison, Logical, and Special. We will look at each group of operators in turn.

Arithmetic operators

Arithmetic operators are generally for numerical related operations.

Arithmetic Operators			
Binary		Unary	
+	Addition	-	Negation
-	Subtraction		
*	Multiplication		
/	Division (Floating point)		
\	Division (Integer)		
^	Exponentiation		
Mod	Modulo		

Operator Precedence

Precedence refers to what operations will be done first when a computation involves more than one operation. The rules in VB are the same as those in algebra. You may find it helpful to think of the phrase "Please Excuse My Dear Aunt Sally" for **P**arentheses / **E**xponentiation / **M**ultiplication and **D**ivision (same precedence left-to-right) / **A**ddition and **S**ubtraction (same precedence left-to-right). Bear in mind that in VB, the precedence of "integer division" and "modulo" are below multiplication and "real" division, but above addition and subtraction. Try to use parentheses to make code more readbale instead of depending on operator precedence, in case whoever read your code might not familiar with the precedence.

Comparison Operators

To compare two values, these operators are in used.

Comparison Operators	
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
=	Equal to
<>	Not equal to
Is	Object type equality

Logical Operators

The purpose of these operators is to deal with complex/composite conidition. These operators are also used for bitwise operations.

Logical and Bitwise Operators			
Binary		Unary	
And	Logical conjunction	Not	Logical negation
Or	Logical disjunction		
Xor	Logical exclusion		
Imp	Logical implication		
Eqv	Logical equivalence		

String Operator

String concatenation is the only supporting operator in VBA. With this operator, we can join string and other data types to for another string.

String Operator	
&	Concatenation

Special Operators

There also special operator that deal with Objects.

Special Operator	
TypeOf	Get object type

Lab Exercise:

In this lab, you are required to solve a given problem in VBA. The solution of this problem involves numbers of operators.

Duration: *45 Minutes*

The Challenge:

Write a VBA program that allows a user to input the loan amount, interest rate, and number of years to pay off the loan. The program should display the monthly payment, total amount that will be paid out over the life of the loan, and the cost of credit.

The monthly payment is computed as follows:

$$Payment = \frac{Rate(1 + Rate)^N}{(1 + Rate)^N - 1} \times Loan$$

where Payment = monthly payment
Rate = monthly interest rate
N = number of payments in months
Loan = amount of loan

You will need to convert the annual interest rate to the monthly rate by dividing the annual rate by 12 and then by 100 (or by 1200). Convert the time in years to the number of monthly payments by multiplying by 12.

The total amount paid is the monthly payment times the number of years times 12. The cost of credit is the total amount paid minus the loan amount.

For example, if the user finances $14,000 to be paid out over four years at an interest rate of 7%, I would enter the following values:

Loan Amount: 14000
Interest Rate: 7
Number of Years: 4

The program should then come up with the following results:

Monthly Payment: $335.25
Total to be paid: $16,091.88
Cost of Credit: $2,091.88

Instructions:

1) Switch to VBE (If you are not yet).
2) Create code module12
3) Write the code in module12's code editor and test it using immediate window

--- End of Lab ---

Chapter 13: *Branching Constructs*

The flow of execution of the program can be determined by the state of execution of the program. In this Chapter we will look at the special decision-making constructs provided by VBA.

Unconditional Branching with GoTo statement

The GoTo statement can be used to branch to statements within a Sub or Function procedure. We have all heard lectures on the "evils" of using GoTo statements. In general, use of modern programming virtually eliminates the need to use GoTos. You'll find that the only time that the use of GoTo is required is when setting up error-handling code (error-handling is covered in more advanced course later).

The rules for using GoTo are as follows:

- The destination of a GoTo statement must be a line label or line number that resides within the same Sub or Function procedure as the statement that issues the GoTo (i.e., you cannot "go to" a line in a Sub or Function other than the one you are currently in).
- The name of the line label follows the same rules as that for naming a variable. The line label must begin in the first column of the line and must be followed by a colon (:).
- If a line number is used, it must begin in the first column of the line.

Unconditional Branching with GoSub statement

In earlier versions of BASIC, the only way you could make your programs modular was to break your program up into "subroutines" and use GoSub to execute that subroutine and return to the calling statement. GoSub was included in VB to maintain backward compatibility; it is not a particularly "bad" construct to use, but its use is generally discouraged – it is recommended that Sub and Function procedures be used instead.

The rules for using GoSub are as follows:

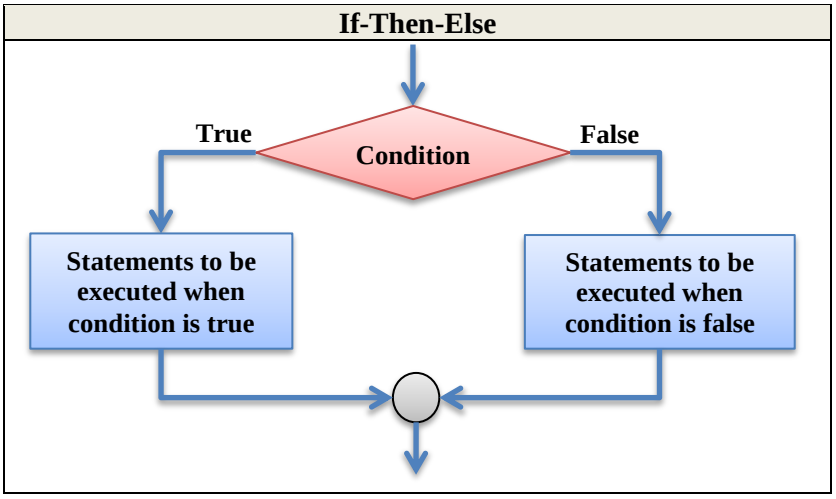
- The destination of a GoSub statement must be a line label or line number that resides within the same Sub or Function procedure as the statement that issues the GoSub (i.e., you cannot "GoSub" to a line in a Sub or Function other than the one you are currently in). In effect, using GoSub lets you have "subs within a sub" (with the exception that you cannot pass parameters to a "GoSubbed" routine).
- If a line label is used, the name of that line label must follow the same rules as that for naming a variable. The line label

must begin in the first column of the line and must be followed by a colon (:).
- If a line number is used, it must begin in the first column of the line.

Once the destination of the GoSub is reached, a Return statement will return control to the statement after the one that issued the GoSub.

If..Then..Else Statement

The selection control structure allows one set of statements to be executed if a condition is true and another set of actions to be executed if a condition is false. A selection structure, also called an "If-Then-Else" structure, is flowcharted as follows:



After either the true set of actions or the false set of actions is taken, program control resumes with the next statement (the statement that would be placed below the connector in the flowchart above).

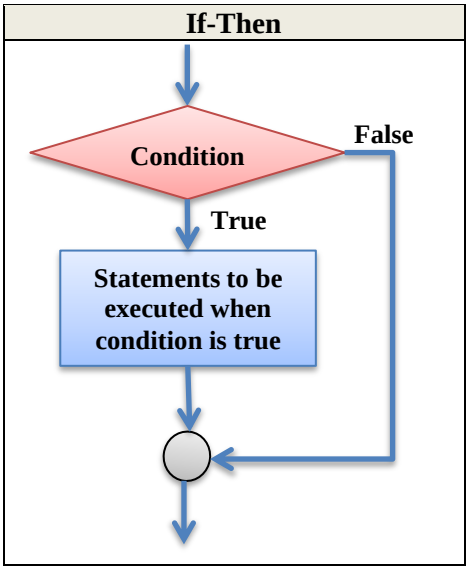
In VBA, the following form is preferred for implementing the If-Then-Else structure (this is the "block", or "multi-line" form of the If statement):

```
If <conditional expression> Then
    <one or more statements to be executed if condition is true>
Else
    <one or more statements to be executed if condition is false>
End If
```

If the conditional expression is true, the statements between the keywords Then and Else will be executed (and the statements between the keywords Else and End If will be bypassed). If the

conditional expression is false, the statements between the keywords Else and End If will be executed (and the statements between the keywords Then and Else will be bypassed). In any case, program control will resume with the statement following End If.

The If-Then-Else statement is a "two-alternative" decision - actions are taken on both the "If" side and the "Else" side. Sometimes, however, you may only want to perform an action or set of actions if a condition is true but do nothing special if the condition is false. This could be flowcharted as follows:



Extended Block If Statement (If/Then/ElseIf)

Format:

```
If <conditional expression 1> Then
      <one or more statements to be executed if condition 1 is true>
ElseIf <conditional expression 2> Then
      <one or more statements to be executed if condition 2 is true>
  . . .
ElseIf <conditional expression n> Then
      <one or more statements to be executed if condition n is true>
Else
      <one or more statements to be executed if none of the above conditions is true>
End If
```

Note that one or more ElseIf clauses are "sandwiched" between the first "If" clause and the last "Else" clause. Note also the keyword ElseIf is one word. Using the format above, this extended If structure is to be understood as follows: if "conditional expression 1" is true, perform the statements associated with that condition, then exit to the statement following the End If; if "conditional expression 1" is false, then check "conditional expression 2" - if "conditional expression 2" is true, perform the statements

associated with that condition, then exit to the statement following the End If, and so on. VB will execute the statements associated with the first true conditional expression it finds and then exit to the statement following the End If. The final Else statement is often useful to trap errors that may occur when unexpected conditions arise, none of which matches the conditions in the previous If or ElseIf clauses.

Note that the use of "ElseIf" saves the coding of multiple "End If" statements in a nested If structure.

The IIf (Immediate If) Function

For cases where you want to assign a particular variable one value if a condition true and another value if a condition is false, you can use the IIf function. The syntax is:

```
IIf(<conditional expression>, <true part>, <false part>)
```

The statement

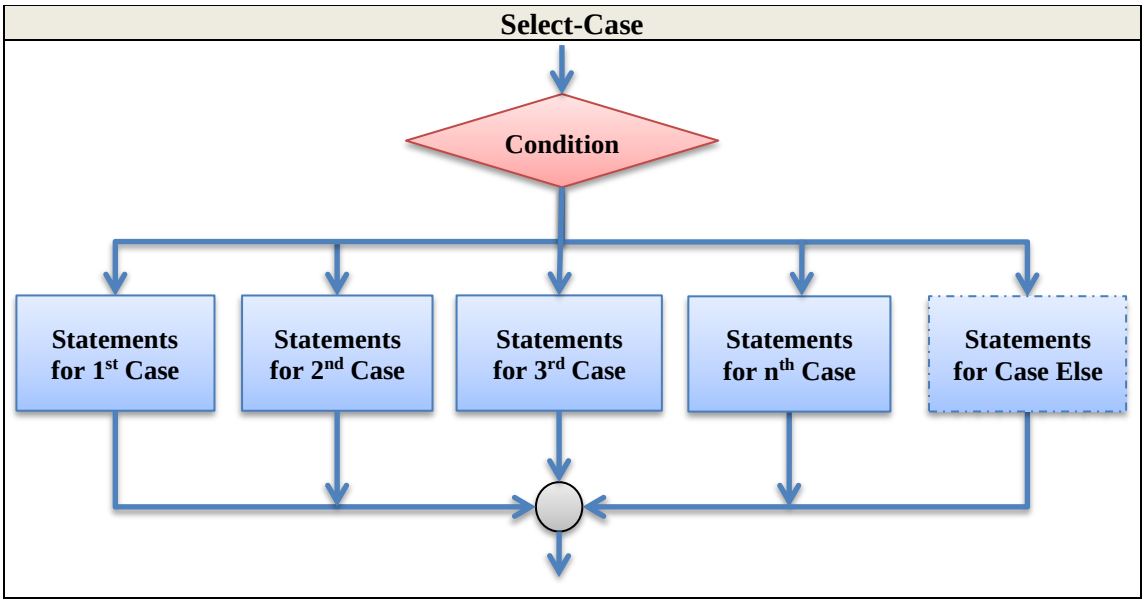
```
strMessage = IIf(sngAvgGrade >= 60, "You passed!", "You failed!")
```

is equivalent to

```
If sngAvgGrade >= 60 Then
   strMessage = "You passed!"
Else
   strMessage = "You failed!"
End If
```

Select Case Statement

When the situation arises where you need to choose between more than two alternatives, an extended form of the selection structure, called the case structure, must be used. A flowcharted example follows:



VBA's Select Case is a powerful statement with several options. The format is:

```
    Select Case <test expression>
Case <expression list 1>
        <statement list 1>
Case <expression list 2>
        <statement list 2>
...
[Case Else <statement list n>]
    End Select
```

The format above is to be understood as follows: The Select Case statement specifies an expression to be tested. Each subsequent Case clause specifies an expression(s) that the test expression will be compared to. The first Case clause that contains an expression that matches the test expression will have its associated actions executed, then program control will branch to the statement following End Select. The final Case Else clause is often useful to trap errors that may occur when an unexpected value of the test expression is present, none of which matches the expression list specified in any of the above Case clauses.

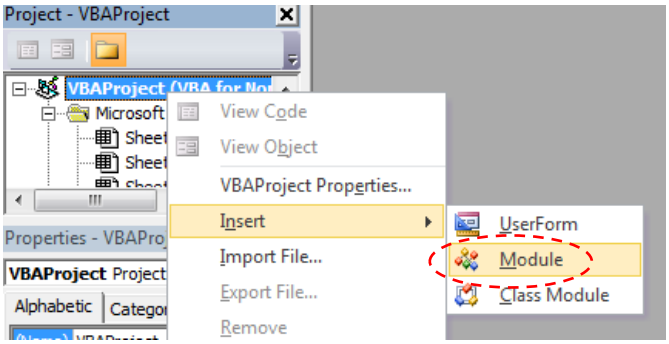
Lab Exercise:

In this lab exercise, you will apply conditional and conditional branching in VBA.

Duration: *60 Minutes*

Instructions:

1) Switch to VBE (If you are not yet).
2) Create code module13



3) Key in the following code in the module13's code editor

No	Code
1	Option Explicit
2	
3	Sub **testGoTo**()
4	Dim lngFactorial%, intInputNbr%, intLoopCtr%
5	
6	intInputNbr = **Val**(**InputBox**("Enter a number:", "GoTo Demo"))
7	lngFactorial = 1
8	intLoopCtr = 1
9	Loop_Start:
10	If intLoopCtr>intInputNbr Then GoTo 10
11	
12	lngFactorial = lngFactorial * intLoopCtr
13	intLoopCtr = intLoopCtr + 1
14	GoTo Loop_Start
15	
16	10' End of loop
17	Debug.Print **CStr**(intInputNbr); "! = "; lngFactorial
18	End Sub
19	
20	Sub **testGoSub**()
21	GoSub SubroutineA
22	GoSub SubroutineB
23	GoSub 1000
24	Exit Sub
25	SubroutineA:
26	Debug.Print "Hey kids, I'm in Subroutine A"
27	Return
28	SubroutineB:
29	Debug.Print "Hey kids, I'm in Subroutine B"
30	Return
31	1000
32	Debug.Print "Hey kids, I'm in Subroutine 1000"
33	Return

```
34  End Sub
35
36  Function MyABS%(ByVal v%)
37    'This function demonstrates the use of IF-THEN statement and negation operator
38    If v <0 Then v = -v
39    MyABS = v
40  End Function
41
42  Function MyMin#(ByVal x#, ByVal y#)
43    'Function to demonstrate subsitution of IIf function for IF-THEN-ELSE statement
44    If (x > y) Then
45      MyMin = y
46    Else
47      MyMin = x
48    End If
49    'MyMin = IIf(x > y, y, x)
50  End Function
51
52  Sub TestOddEven()
53    Dim sInput$
54    Dim iInput%
55    sInput = InputBox("The Integer Value", "Test Odd/Even")
56    iInput = CInt(sInput)
57    If IsEven(iInput) Then
58      MsgBox "The number is Even"
59    Else
60      MsgBox "The number is Odd"
61    End If
62  End Sub
63
64  Function CompareTo%(ByVal v1%, ByVal v2%)
65    If (v1 > v2) Then
66      CompareTo = 1
67    Else
68      If (v1 < v2) Then
69        CompareTo = -1
70      Else
71        CompareTo = 0
72      End If
73    End If
74  End Function
75
76  Sub WhatToEat()
77    'Subroutine shows nested IF statements
78    Dim sDay$
79    Dim sFood$
80    sDay = uCase(Trim(InputBox("Day=", "Which day?")))
81    If sDay = "MON" Then
82      sFood = "Burger"
83    Else
84      If sDay = "WED" Then
85        sFood = "Chicken"
86      Else
87        If sDay = "FRI" Then
88          sFood = "Satay"
89        Else
90          sFood = "Nasi"
91        End If
92      End If
```

```
93    End If
94    MsgBox "You should eat " & sFood
95  End Sub
96
97  Sub WhatToEat2()
98    'Subroutine shows using ElseIf to reduce code nesting
99    Dim sDay$
100   Dim sFood$
101   sDay = uCase(Trim(InputBox("Day=", "Which day?")))
102   If sDay = "MON" Then
103     sFood = " Burger"
104   ElseIf sDay = "WED" Then
105     sFood = "Chicken"
106   ElseIf sDay = "FRI" Then
107     sFood = "Satay"
108   Else
109     sFood = "Nasi"
110   End If
111   MsgBox "You should eat " & sFood
112 End Sub
113
114 Sub WhatToEat3()
115   'Subroutine shows a better alternative of WhatToEat() by using Select-Case
116   Dim sDay$
117   Dim sFood$
118   sDay = uCase(Trim(InputBox("Day=", "Which day?")))
119   Select Case sDay
120     Case"MON":      sFood = "Burger"
121     Case"WED":      sFood = "Chicken"
122     Case"FRI":      sFood = "Satay"
123     Case"TUE", "THU", "SAT", "SUN": sFood = "Nasi"
124     Case Else
125           MsgBox "Invalid entry"
126           Exit Sub
127   End Select
128   MsgBox "You should eat " & sFood
129 End Sub
130
131 Function TicketPrice(ByVal age%) As Currency
132   'Function shows various form cases
133   Select Case age
134     Case Is <1, Is >128
135           MsgBox "Age " & age & vbNewLine & "Is this human?", vbCritical, "Age error"
136     Case 1, 2: TicketPrice = 0          'Infant
137     Case 3 To 11: TicketPrice = 5       'Kid
138     Case Is <= 18:  TicketPrice = 6     'Teenager
139     Case Is >= 80:  TicketPrice = 2     'Eldery
140     Case Is >55: TicketPrice = 4        'Senior citizen
141     Case Else: TicketPrice = 10         'Normal Adult
142
143   End Select
144 End Function
```

4) Test all the procedures and understand how each of them works.

--- End of Lab ---

Chapter 14: *Iteration Constructs*

The repetition control structure is also known as the looping or iteration control structure. Looping is the process of repeatedly executing one or more steps of an algorithm or program; it is essential in programming, as most programs perform repetitious tasks.

Every loop consists of the following three parts:

The loop termination decision - determines when (under what condition) the loop will be terminated (stopped). It is essential that some provision in the program be made for the loop to stop; otherwise, the computer would continue to execute the loop indefinitely - a loop that doesn't stop is called an endless loop or infinite loop; when a program gets stuck in an endless loop, programmers say that the program is "looping".

The body of the loop - the step or steps of the algorithm that are repeated within the loop. This may be only one action, or it may include almost all the program statements. Important note: some action must occur within the body of the loop that will affect the loop termination decision at some point.

Transfer back to the beginning of the loop - returns control to the top of the loop so that a new repetition (iteration) can begin.

Unconditional Loop with GoTo statement

As mentioned in the branching Chapter, the GoTo statement can be used to branch to statements within a Sub or Function procedure. But, if the label or number are before the GoTo statement, this will cause the code execution branch backward. This will form an unconditional loop.

Using For Loop

The For/Next loop uses a built-in counting procedure to repeat a series of instructions a specific number of times.

The general format of the For/Next loop is as follows:

```
For <loop control variable> = <initial value> To <stop value> [Step increment value]
        <list of statements>
Next [loop control variable]
```

When the loop begins, the loop control variable (LCV) will be initialized with the value of the initial value. Subsequently, a test will be made to see if the LCV exceeds the stop value. If the LCV

exceeds the stop value, the loop will end, and control will pass to whatever statement follows the Next statement.

If the LCV is less than or equal to the stop value, the body of the loop will be executed, and the Next statement automatically increments the LCV by the value specified after the keyword Step (if the Step clause is omitted, the increment value is assumed to be 1). The Next statement also causes a transfer of control back to the **For** statement to begin another repetition of the loop. Note from the general format that the LCV may optionally be specified after the keyword Next, however, programming literature states that it is more efficient to leave it off.

Other notes regarding the For/Next loop:

Scenarios	Output

The Step value need not be 1. The statements
 	
`For X = 1 To 10 Step 2`
`  Print X`
`Next`
	1
	3
	5
	7
	9

The Step value can be negative, in which case the loop executes until the LCV is less than the stop value - therefore, the initial value should be greater than the stop value. The statements
 	
`For X = 3 To 1 Step -1`
`  Print X`
`Next`
	3
	2
	1

Assuming that the Step value is positive (or the default of 1 is used), if the stop value is less than the initial value when the loop begins, the loop will never execute. For example, the statements in the body of this loop will never execute:
 	
`For X = 1 To 5 Step -1`
`  <whatever>`
`Next`	

Using For Each statement

Another very common situation is the need for a loop which enumerates every element of a list.

The general format of the For Each loop is as follows:

```
For Each <iterating variable>In<list>
        <use the variant variable as the list item>
Next [<iterating variable>]
```

The list is commonly a Collection or Array but can be any other object that implements an enumerator. Note that the iterating variable has to be either a Variant, Object or Class that matches the type of elements in the list.

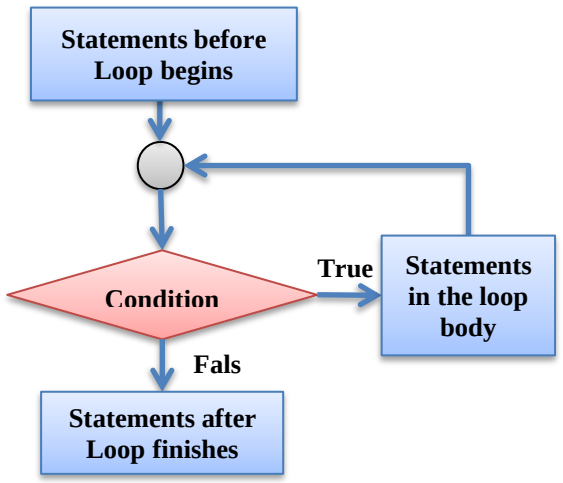
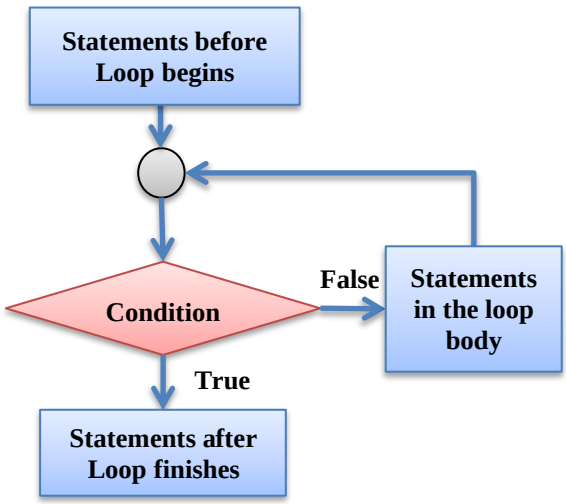
Pre-Test Looping

The general format for a pre-test loop in VB/VBA is:

```
Do {While | Until} <condition>
    <list of statements>
Loop
```

The pre-test loop is implemented in VBA with a "Do" statement followed by either the keyword "While" or "Until". "Do While" means execute the statements in the body of the loop While a certain condition is true. "Do Until" means execute the statements in the body of the loop Until a certain condition is true. In other words, doing something Until a condition is TRUE is the same as doing something While a condition is FALSE. For example, `Do While X <= 10` is the same as `Do Until X > 10`.

With a pre-test loop the "loop termination decision" is tested at the top of the loop, it is possible that the statements in the body of the loop will never be executed.

Do While pre-test loop	**Do Until** pre-test loop
	

The While/Wend Loop

The While/Wend loop is an older statement pair from previous versions of BASIC and was included in VB for compatibility. The format is:

```
While <condition>
        <list of statements>
Wend
```

There is no "Until" equivalent of "While/Wend".

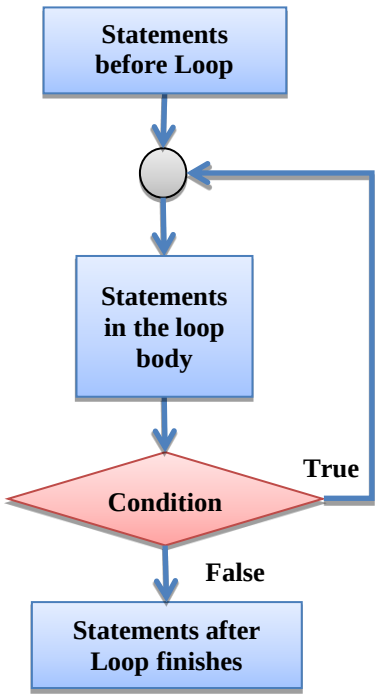
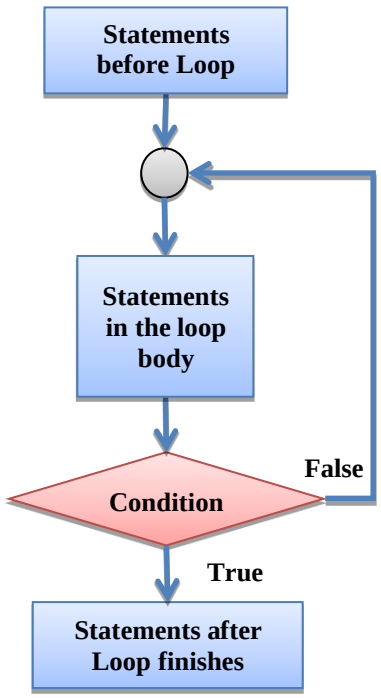
Post-Test Looping

The general format for a post-test loop in VB is:

```
Do
<list of statements>
Loop {While | Until} <condition>
```

With a post-test loop, the "loop termination decision" is tested at the bottom of the loop, therefore the statements in the body of the loop will always be executed **at least once**.

In VB, post-test loops are implemented with the Do/Loop statements, however, the "While" and the "Until" conditions appear after the keyword Loop, not Do. The "While" and the "Until" versions of the post-test loop are flowcharted as below. The only difference between the two is the placement of the "True" and "False" paths extending from the decision diamond.

Loop While post-test loop	**Loop Until** post-test loop
	

Pre-mature termination using Exit keyword

Occasionally, it may be necessary or more efficient to exit a loop early (i.e., before the loop termination decision is reached) due to some other condition that becomes true as the loop is processing. For this situation the **Exit Do** or **Exit For** statement can be used.

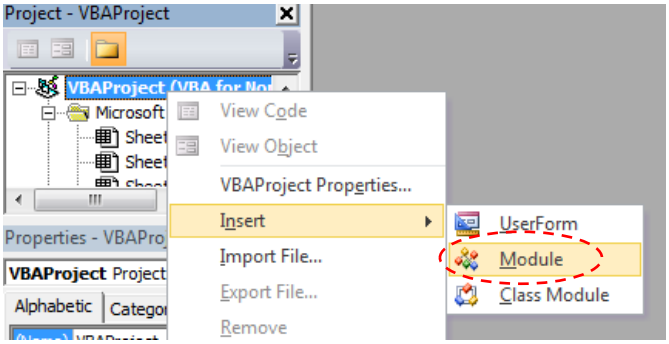
Lab Exercise:

In this lab, you will learn how to perform various types of iteration in VBA coding

Duration: *120 Minutes*

Instructions:

1) Switch to VBE (If you are not yet).
2) Create code module14



3) Copy the following code from instructor into the module14's code editor

No	Code
1	Option Explicit
2	
3	Sub **UnconditionalLoop**()
4	Dim n As Byte
5	
6	Start:
7	n = n + 1
8	Debug.Print n
9	GoTo Start
10	End Sub
11	
12	Sub **TestFor1**()
13	Dim i%
14	
15	For i = 1 To 10
16	Debug.Print i
17	Next i
18	Debug.Print "After loop, i=" & i
19	End Sub
20	
21	Sub **TestFor2**()
22	Dim i%
23	For i = 10 To 1 Step -1
24	Debug.Print i
25	Next i
26	Debug.Print "After loop, i=" & i
27	End Sub
28	
29	Sub **TestFor3**()

```
30    Dim i%
31
32    For i = 1 To 10
33      Debug.Print i
34      If (i = 5) Then Exit For
35    Next i
36    Debug.Print "After loop, i=" & i
37  End Sub
38
39  Function FactFor(ByVal x As Byte) As Long
40    Dim i%
41    FactFor = 1
42    For i = 2 To x
43      FactFor = FactFor * i
44    Next i
45  End Function
46
47  Function FactPre1(ByVal x As Byte) As Long
48    FactPre1 = 1
49    Do While x > 1
50      FactPre1 = FactPre1 * x
51      x = x - 1
52    Loop
53  End Function
54
55  Function FactPre2(ByVal x As Byte) As Long
56    FactPre2 = 1
57    While x > 1
58      FactPre2 = FactPre2 * x
59      x = x - 1
60    Wend
61  End Function
62
63  Function FactPre3(ByVal x As Byte) As Long
64    FactPre3 = 1
65    Do Until x <= 1
66      FactPre3 = FactPre3 * x
67      x = x - 1
68    Loop
69  End Function
70
71  Function FactPost1(ByVal x As Byte) As Long
72    Dim n%
73    FactPost1 = 1
74    Do
75      n = n + 1
76      FactPost1 = FactPost1 * n
77    Loop While (n < x)
78  End Function
79
80  Function FactPost2(ByVal x As Byte) As Long
81    Dim n%
82    FactPost2 = 1
83    Do
84      n = n + 1
85      FactPost2 = FactPost2 * n
86    Loop Until (n >= x)
87  End Function
```

4) Test all the subroutines (Not the functions) from immediate window, and try to understand how they work
5) Prepare a new worksheet. Change the tab name to "M14" and use it to all other factorial functions.

		Excel	For	Pre-Test			Post-Test	
	n	Fact(n)	FactFor(n)	FactPre1(n)	FactPre2(n)	FactPre3(n)	FactPost1(n)	FactPost2(n)
	0	1	1	1	1	1	1	1
	1	1	1	1	1	1	1	1
	2	2	2	2	2	2	2	2
	3	6	6	6	6	6	6	6
	4	24	24	24	24	24	24	24
	5	120	120	120	120	120	120	120
	6	720	720	720	720	720	720	720
	7	5040	5040	5040	5040	5040	5040	5040
	8	40320	40320	40320	40320	40320	40320	40320
	9	362880	362880	362880	362880	362880	362880	362880
	10	3628800	3628800	3628800	3628800	3628800	3628800	3628800
	11	39916800	39916800	39916800	39916800	39916800	39916800	39916800
	12	479001600	479001600	479001600	479001600	479001600	479001600	479001600

--- End of Lab ---

Happy Coding