User defined functions in VBA

Objectives: At the end of this lesson you shall be able to
- **create user defined functions**
- **describe passing values to functions byval and byref**
- **describe using arrays with functions**
- **describe the scope of variables**
- **describe the access specifiers public and private.**

Introduction

In Excel Visual Basic too, like in most programming languages, a set of commands to perform a specific task is placed into a procedure, which can be a function or a subroutine. The main difference between a VBA function and a VBA subroutine is that a function (generally) returns a result, whereas a subroutine does not.

Therefore, if you wish to perform a task that returns a result (ex. summing of a group of numbers), you will generally use a function, but if you just need a set of actions to be carried out (ex. formatting a set of cells), you might choose to use a subroutine.

User Defined Functions

One of the most power features of Excel VBA is that you can create your own functions or UDFs. A UDF (User Defined Function) is simply a function that you create yourself with VBA for your own defined tasks. UDFs are often called "Custom Functions". A UDF can remain in a code module attached to a workbook, in which case it will always be available when that workbook is open. Alternatively you can create your own add-in containing one or more functions that you can install into Excel. Here the user-defined functions can be entered into any cell or on the formula bar of the spreadsheet just like entering the built-in formulas of the MS Excel spreadsheet.

Custom functions, like macros, use the Visual Basic for Applications (VBA) programming language. They differ from macros in two significant ways. First, they use function procedures instead of sub procedures. They start with a Function statement instead of a Sub statement and end with End Function instead of End Sub. Second, they perform calculations instead of taking actions. Certain kinds of statements (such as statements that select and format ranges) are generally excluded from custom functions.

A simple function may look like this:

```
Function area()

Dim l, b

l = 10

b = 20

Debug.Print "area Is " & l * b

End Function
```

When executed from the immediate window this function displays the area.

Alternately this function can be called by another subroutine, for ex.

```
Sub test_fn()

Call area

End Sub
```

Returning a value from the procedures

In the example given below, the area() function calculates l*b.

The subroutine that calls this function is returned this value.

```
Sub test_fn()

Debug.Print "The function has returned the value " & area

End Sub

Function area()

Dim l, b, A

l = 10

b = 20

area = l * b

End Function
```

The result will be:The function has returned the value 200

Passing Arguments to functions

We can pass the arguments in two different ways:

1 By Value (ByVal): We pass the copy of the actual value to the arguments

2 By Reference (ByRef): We pass the reference to the arguments

By Ref is the default method of passing argument type in VBA. This means, if you are not specifying any type of the argument it will consider it as ByRef type. However, it is always a good practice to specify the ByRef even if it is not mandatory.

The following example shows the method of passing variables to a function byVal.

```
Sub test_fn()

Dim a, b As Integer

a = 4

b = multiply(a)

Debug.Print "a is " & a

Debug.Print "The function has returned the value " & b

End Sub

Function multiply(ByVal a As Integer)

a = a * 10

multiply = a

End Function
```

The result of this program will be :

```
a is 4

The function has returned the value 40

a is 4
```

This means that the value of the variable that was passed is not disturbed by the function.

The following example shows the method of passing variables to a function byRef.

```
Sub Test()

Dim A As Integer

A = 10

Debug.Print "The function has returned the value " & Modify(A)

Debug.Print "A is now  " & A

End Sub

Function Modify(ByRef A As Integer)

A = A * 2

  Modify = A

End Function
```

The result will be:

```
The function has returned the value 20

A is now 20
```

Calling a User Defined Function from Worksheet:

You call the user defined functions as similar to the built-in excel function. To do this type the arguments in the cells and type the name of the function as is done with normal functions in Excel.

Passing Arrays to User Defined functions

A Function can accept an array as an input parameter. Arrays are always passed by reference (ByRef). You will receive a compiler error if you attempt to pass an array ByVal. This means that any modification that the called procedure does to the array parameter is done on the actual array declared in the calling procedure.

(If you need to pass an array ByVal then you would need to use the Variant data type.)

An example of passing an array to a function is as follows:

```
Sub test()

Dim arr(1 To 10) As Integer

Dim i As Integer

'populates the array with the values 1 to 10

For i = 1 To 10

arr(i) = i

Next i

'call the function example 1 with arrIntegers as an input parameter

Call fn1(arr)

For i = 1 To 10

Debug.Print arr(i); Spc(2);

Next i

End Sub
```

```
'prints the values in arrIntegers to column A
Sub fn1(ByRef arr() As Integer)
Dim i As Integer
For i = LBound(arr) To UBound(arr)
  arr (i) = arr (i) * 2
  Cells (i,1) = arr (i)
Next i
End Sub
```

Scope of variables

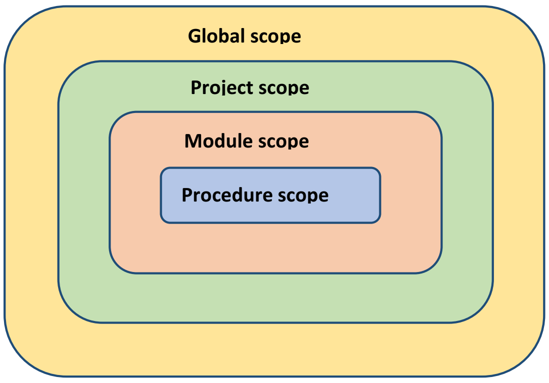
The term Scope is used to describe how a variable may be accessed. Depending on where and how a variable is declared, it may be accessible only to a single procedure, to all procedures within a module, and so on up the hierarchy of a project or group of related projects. The term visibilty is also is sometimes used to describe scope.

There are four levels of Scope:

- Procedure-Level Scope
- Module-Level Scope
- Project-Level Scope
- Global-Level Scope

Fig 1 shows the various scopes and their levels.

Fig 1



Procedure (local) scope

A local variable with procedure scope is recognized only within the procedure in which it is declared. A local variable can be declared with a Dim or Static statement.

When a local variable is declared with the Dim statement, the variable remains in existence only as long as the procedure in which it is declared is running. Usually, when the procedure is finished running, the values of the procedure's local variables are not preserved, and the memory allocated to those variables is released. The next time the procedure is executed, all of its local variables are reinitialized.

For Example the following subroutine has been created in Module1 code.

```
Sub disp()
Dim s As string
s="hello"
MsgBox s
End Sub
```

Run the subroutine "disp" in Module1 and it will display the message "Hello" in the message box.

Now the following subroutine has been created in Sheet1 code to call the disp() subroutine from Module1.

```
Sub Button1_Click()
disp
End Sub
```

This will generate an error since the subroutine disp() and the variable s are local to Module1 and cannot be accessed from elsewhere.

Static:

A local variable declared with the Static statement remains in existence the entire time Visual Basic is running. The variable is reset when any of the following occur:

- The macro generates an untrapped run-time error.
- Visual Basic is halted.
- You quit Microsoft Excel.
- You change the module.

For example, in the FindTotal example, the Accumulate variable retains its value every time it is executed. The first time the module is run, if you enter the number 2, the message box will display the value "2." The next time the module is run, if the value 3 is entered, the message box will display the running total value to be 5.

```
Sub FindTotal()
Static Total
Dim n as integer
n =InputBox("Enter a number: ")
Total = Total + n
MsgBox "The total is " &n
End Sub
```

Module scope

A variable that is recognized among all of the procedures on a module sheet is called a "module-level" variable. A module-level variable is available to all of the procedures in that module, but it is not available to procedures in other modules. A module-level variable remains in existence while Visual Basic is running until the module in which it is declared is edited. Module-level variables can be declared with a Dim or Private statement at the top of the module above the first procedure definition.

At the module level, there is no difference between Dim and Private. Note that module-level variables cannot be declared within a procedure.

Note If you use Private instead of Dim for module-level variables, your code may be easier to read (that is, if you use Dim for local variables only, and Private for module-level variables, the scope of a particular variable will be more clear).

In the following example, two variables, A and B, are declared at the module level. These two variables are available to any of the procedures on the module sheet. The third variable, C, which is declared in the Example3 macro, is a local variable and is only available to that procedure.

Note that in Test4, when the macro tries to use the variable C, the message box is empty. The message box is empty because C is a local variable and is not available to Test4, whereas variables A and B are.

```
Dim A As Integer       ' Module-level variable.

Private B As Integer    ' Module-level variable.

Sub Test1()

A = 10

B = A * 10

End Sub

Sub Test2()

MsgBox "The value of A is " & A

MsgBox "The value of B is " & B

End Sub

Sub Test3()

Dim C As Integer    ' Local variable.

C = A + B

MsgBox "The value of C is " & C

End Sub

Sub Test4()

MsgBox A

MsgBox B

MsgBox C

 ' The message box is blank since C is a local variable.

End Sub
```

Project Scope

Project scope variables are those declared using the Public keyword. These variables are accessible from any procedure in any module in the project. In Excel, a Project is all of the code modules, userforms, class modules, and object modules (e.g. ThisWorkbook and Sheet1) that are contained within a workbook.

In order to make a variable accessible from anywhere in the project, you must use the Public keyword in the declaration of the variable. However, this makes the variable accessible to any other project that reference the project containing the variable. If you want a variable to be accessible from anywhere within the project, but not accessible from another project, you need to use Option Private Module as the first line in the module (above and outside of any variable declaration or procedure). This option makes everything in the module accessible only from within the project. The project variables that should not be accessible to other projects should be declared in a module that has the Option Private Module directive. Variables that should be accessible to other project should be declared in a different module that does not use the Option Private Module directive. In both cases, however, you need to use the Public keyword.

Global Scope

Global scope variables are those that are accessible from anywhere in the project that declares them as well as any other project that references the first project. To declare a variable with global scope, you need to declare it using the Public keyword in a module that does not use the Option Private Module directive. In order to access variables in another project, you can simply use the variable's name. If, however, it is possible that the calling project also has a variable by the same name, you need to prefix the variable name with the project name. For example, if Project1 declares a global variable named x, and Project2 references Project1, code that is in Project2 can access x with either of the following lines of code:

```
x = 78

Project1.x = 78
```

If both Project1 and Project2 have variables with at least project scope, you need to include the project name with the variable. For clarity and maintainability, you should always include the project name when accessing a variable that is declared in another project. Even if this is not necessary, it makes the code more readable and maintainable.

There is no way to give some variables project, but not global, scope and give others in the same module global scope. Project versus global scope is handled only at the module level, not at the variable level.

The Access Specifiers

One of the techniques in object-oriented programming is encapsulation. It concerns the hiding of data in a class and making them available only through its methods. Most programming languages implementing OOPS allow you to control access to classes, methods, and fields via so-called access modifiers. The access to classes, constructors, methods and fields are regulated using access modifiers i.e. a class can control what information or data can be accessible by other classes. The VBA access specifiers are:

1 Private

2 Public

A Public procedure is accessible to all code inside the module and all code outside the module, essentially making it global. A VBA Private Sub can only be called from anywhere in the Module in which it resides. A Public Sub in the objects, ThisWorkbook, ThisDocument, Sheet1, etc. cannot be called from anywhere in a project. However, if you declare a Module level variable with the Public Keyword it can be used anywhere in the project and retains its value.

If you exclude the key word private in your declaration then by default the procedure is public. So Sub MySub() and Public Sub MySub() are exactly the same thing.

Public [variable] means that the variable can be accessed or used by subroutines in outside modules. These variables must be declared outside of a subroutine (usually at the very top of your module). You can use this type of variable when you have one subroutine generating a value and you want to pass that value on to another subroutine stored in a separate module.

A Private procedure is only available to the current module. It cannot be accessed from any other modules, or from the Excel workbook.Private Sub sets the scope so that subroutines from outside modules cannot call that particular subroutine. This means that a sub in Module 1 could not use the Call method to initiate a Private Sub in Module 2.

Private [variable] means that the variable cannot be accessed or used by subroutines in other modules. In order to be used, these variables must be declared outside of a subroutine (usually at the very top of your module). You can use this type of variable when you have one subroutine generating a value and you want to pass that value on to another subroutine in the same module.

Dim[variable] is used to state the scope inside of a subroutine (you cannot use Private in its place). Dim can be used either inside a subroutine or outside a subroutine (using it outside a subroutine would be the same as using Private).

Example of Public, Private Variables and Procedures.

Module 1 code

```
Dim x As Integer ' This is a Private Variable since it is declared using Dim.

Public y As Integer

Sub First_Sub()

  x = 10

  y = 20

 Call Third_Sub()

End Sub

Private Sub Second_Sub()

MsgBox "Gone through First, Second and Third Subroutines !"

End Sub
```

Module 2 code

```
Sub Third_Sub()

Debug.Print x

Debug.Print y

Call Second_Sub()

End Sub
```

The two variables x and y that are declared outside a subroutine. This means that their values can carry over into other macros. The variable x has a private scope so only subroutines in the same module can access its value. The variable y has a public scope, meaning that subroutines inside and outside its module can access its value.

The First_Sub() assigns values to x and y and then initiates the Third_Sub().

Third_Sub() can be called even though it is not in the same module because it is a Public Sub.

The Third_Sub() has been designed to display the values of x and y in the immediate window. When you try to print variable x it outputs nothing. This is because x does not exist in Module 2. Therefore, a new variable x is created in Module 2 and since we did not give this new x a value, nothing is printed for the statement Debug.Print x

When we print the value of the variable y, 12 is displayed in the Immediate Window. This is because Module 2 subroutines have access to the public variables declared in Module 1.But the statement "Call Second_Sub()" in the Third_Sub() will result in an error. This is because we are trying to call a private subroutine " Second_Sub" from here. The following changes can be done to avoid this:

1 We could remove the word "Private" from Display_Message

2 We could replace "Private" with "Public" in Second_Sub()

3 We can use the Application level and instead of using Call we could write Application.Run "Second_Sub " (this method serves as an override in case we wanted to keep Second_Sub private for subroutines outside the module.)

Create and Edit Macros

Objective: At the end of this lesson you shall be able t0
• **explain about Macros in VBA.**

Macros offer a powerful and flexible way to extend the features of Excel. They allow the automation of repetitive tasks such as printing, formatting, configuring, or otherwise manipulating data in Excel. In its' simplest form, a macro is a recording of your keystrokes. While macros represent one of the stronger features found in Excel, they are rather easy to create and use. There are six major points that I like to make about macros as follows.

1 **Record, Use Excel, Stop Recording**

To create a macro, simply turn on the macro recorder, use Excel as you normally do, then turn off the recorder. Presto – you have created a macro. While the process is simple from the user's point of view, underneath the covers Excel creates a Visual Basic subroutine using sophisticated Visual Basic programming commands.

2 **Macro Location**

Macros can be stored in either of two locations, as follows:

The workbook you are using, or the Personal Macro Workbook (which by default is hidden from view) If the macro applies to all workbooks, then store it in the Personal Macro Workbook so it will always be available in all of the Excel workbooks; otherwise store it in the current workbook. A macro stored in the current workbook will embedded and included in the workbook, even if you email the workbook to another user.

3 **Assign the Macro to an Icon, Text or a Button**

To make it easy to run your macro, you should assign it to a toolbar icon so it will always be available no matter which workbooks you have open. If the macro applies only to the current workbook, then assign it to Text or a macro Button so it will be quickly available in the current workbook.

4 **Absolute versus Relative Macros**

An "Absolute" macro will always affect the same cells each time whereas a "Relative" macro will affect those cells relative to where the cursor is positioned when invoke the macro. It is crucial that understand the difference.

5 **Editing Macros**

Once created, you can view and/or edit your macro using the View Macros option. This will open the macro subroutine in a Visual basic programming window and provide you with a plethora of VB tools.

6 **Advanced Visual Basic Programming**

For the truly ambitious CPA, in the Visual Basic Programming window, you have the necessary tools you need to build very sophisticated macros with dialog boxes, drop down menu options, check boxes, radio buttons – the whole works. To see all of this power, turn on the "Developer Tab" in "Excel Options". Presented below are more detailed comments and stepbystep instructions for creating and invoking macros, followed by some example macros.

Page Setup Macro Start recording a new macro called page setup. Select all of the worksheets and then choose Page Setup and customize the header and footers to include page numbers, date and time stamps, file locations, tab names, etc. Assign the macro to an Icon onthe toolbar or Quick Access Bar and insetting headers and footers will be a breeze for the rest of your life.

Print Macros Do you have a template that print frequently from? If so, insert several macro buttons to print each report, a group of reports, and even multiple reports and reporting will be snap in the future.

Delete Data Macro create a macro that visits each cell and erases that data, resetting the worksheet for use in a new set of criteria. Assign the macro to a macro button and will never again have old assumptions mixed in with your newer template

User forms and control in Excel VBA

Objectives: At the end of this lesson you shall be able to
- **define forms and controls in VBA**
- **describe the types of excel forms**
- **describe the properties, methods and events of forms.**

Introduction to Forms and Controls

A form is a document designed with a standard structure and format that makes it easier to enter, organize, and edit information. Forms contain labels, textboxes, drop down boxes and command buttons too.

By using forms and the many controls and objects that you can add to them, you can significantly enhance data entry on your worksheets and improve the way your worksheets are displayed.

Types of Excel forms

There are several types of forms that you can create in Excel: data forms, worksheets that contain Form and ActiveX controls, and VBA UserForms.

Data form

A data form provides a convenient way to enter or display one complete row of information in a range or table without scrolling horizontally. You may find that using a data form can make data entry easier than moving from column to column when you have more columns of data than can be viewed on the screen. Excel can automatically generate a built-in data form for your range or table.

Worksheet with Form and ActiveX controls

A worksheet can be considered to be a form that enables you to enter and view data on the grid.

For added flexibility, you can add controls and other drawing objects to the worksheet, and combine and coordinate them with worksheet cells. For example, you can use a list box control to make it easier for a user to select from a list of items. Or, you can use a spin button control to make it easier for a user to enter a number.

You can display or view controls and objects alongside associated text that is independent of row and column boundaries without changing the layout of a grid or table of data on your worksheet. Many of these controls can also be linked to cells on the worksheet and do not require VBA code to make them work. For example, you might have a check box that you want to move together with its underlying cell when the range is sorted. However, if you have a list box that you want to keep in a specific location at all times, you probably do not want it to move together with its underlying cell.

Creating VBA Forms

A VBA form can be created from the code window. To create a Form in VBA,click on Insert menu in the code window and then click 'UserForm'. A UserForm1 appears in the project window.

When you create or add a form, a module is also automatically created for it. To access the module associated with a form, you can right-click the form and click View Code.Double Clicking on the Form or pressing F7 will also open the Code window. Using Shift F7 will again switch back to the Design Window.

The design time properties of the Form can be set by right clicking on the form and selecting 'Properties'. The same can be achieved by Clicking "F4" or the properties button on the Form.Controls can be placed on the form from the ToolBox as per requirement.

In addition, Controls can be added on the Form programmatically / at run time using the "Add" method. Similarly the controls can be removed from the form at run time / programmatically using the "Remove" method. As an example to add a checkbox control, we can write

```
Set cb1 = Controls.Add("Forms.CheckBox.1")
```

Some of the events and methods connected with the form object are:

Events, Activate, Deactivate, Add Control, Remove Control, Click, DblClick, Initialize, KeyPress, Resize, Scroll, Terminate, Zoom etc.

Methods Copy, Paste, Hide, Move, Print Form, Repaint, Scroll, Show etc .

The code needed to perform various operations on Forms is given in Table 1.

A sample Form for data entry of students' details, marks and results is shown in Fig. 1.

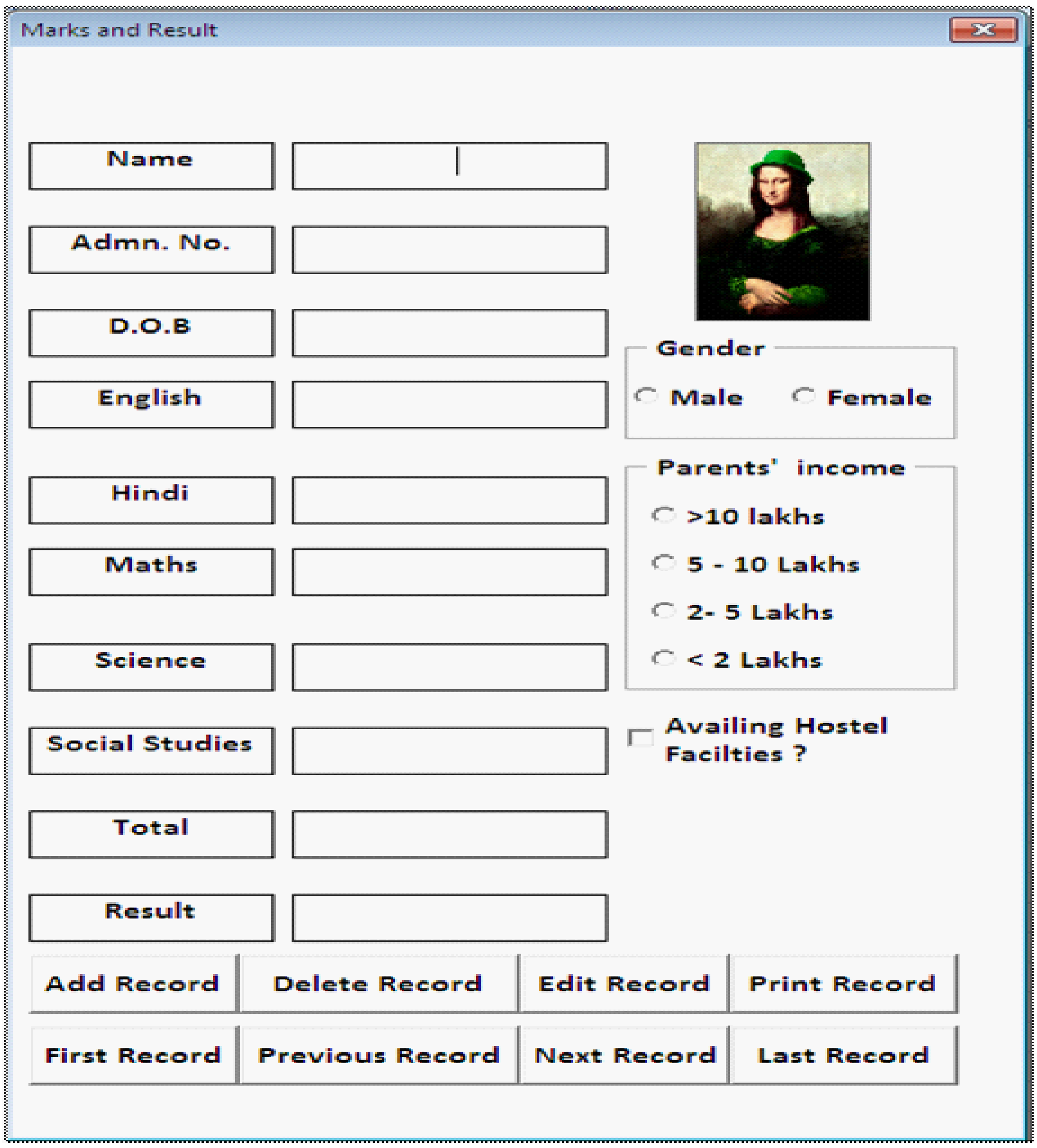
Necessary code can be attached to the Command Buttons and other controls shown. After the user enters the data, the total is calculated and the result is displayed. The records can then be stored appropriately.

Table 1

Userform Application	VBA Code	Action
To Display a UserForm	UserForm1.Show	Displays the UserForm with name UserForm1. This code should be inserted in a Standard VBA Module and not in the Code Module of the UserForm. You can create a button in a worksheet, then right click to assign macro to this button, and select the macro which shows the UserForm.
Load a UserForm into memory but do not display	Load UserForm1	Load statement is useful in case of a complex UserForm that you want to load into memory so that it displays quickly on using the Show method, which otherwise might take a longer time to appear.
Remove a User Form from memory / Close UserForm	Unload UserForm1	Note: The Hide method (UserForm1.Hide) does not unload the UserForm from memory. To unload the UserForm from memory, the Unload method should be used.
	Unload Me	Use the Me keyword in a procedure in the Code Module of the UserForm.
Hide a UserForm	UserForm1.Hide	Using the Hide method will temporarily hide the UserForm, but will not close it and it will remain loaded in memory.
Print a UserForm	UserForm1.PrintForm	The PrintForm method sends the UserForm directly for printing.
Display UserForm as Modeless	UserForm1.Show False	If the UserForm is displayed as Modeless, user can continue working in Excel while the UserForm continues to be shown. Omitting the Boolean argument (False or 0) will display the UserForm as Modal, in which case user cannot simultaneously work in Excel. By default UserForm is displayed as Modal.
Close a UserForm	Unload UserForm1	The Unload method closes the specified UserForm.
	Unload Me	The Unload method closes the UserForm within whose Code Module it resides.
	End	Use the End statement in the "Close" CommandButton to close the form. The "End" statement unloads all forms.
Specify UserForm Caption	UserForm1.Caption = "Bio Data"	Caption is the text which describes and identifies a UserForm and will display in the header of the Userform.
Set UserForm size	UserForm1.Height = 250	Set Height of the UserForm, in points.
	UserForm1.Width = 350	Set Width of the UserForm, in points.
		Set UserForm Position:
Left & Top properties	UserForm1.Left = 30 UserForm1.Top = 50	Distance set is between the form and the Left or Top edge of the window that contains it, in pixels.
Move method	UserForm1.Move 200, 50	Move method includes two arguments which are required - the Left distance and the Top distance, in that order.

Necessary code can be attached to the Command Buttons and other controls shown. After the user enters the data, the total is calculated and the result is displayed. The records can then be stored appropriately.

Fig 1



Methods and Events in VBA

Objectives: At the end of this lesson you shall be able to
- **explain VBA methods and events.**

Methods and Events

Methods

A method is an action you perform with an object. A method can change an object's properties or make the object do something.

For example painting is a Method, building a new room is a method in building a new house.

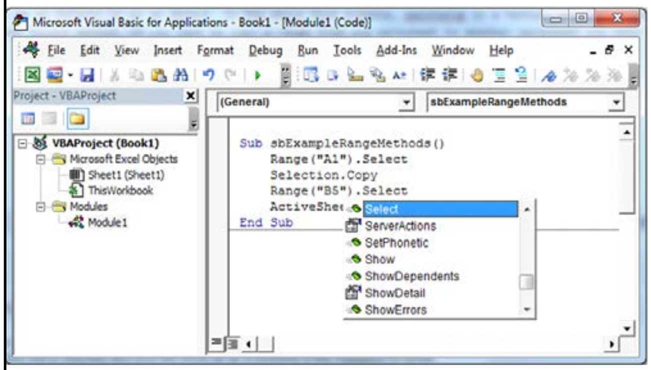
Similarly, if you want to select a range, you need Select method. If you want to copy a range from one worksheet to another worksheet you need Copy method to do it.

The following example Copies the data from Range A1 to B5.

Enter the following code in the Module1 as shown in Fig 1

```
Sub sbExampleRangeMethods()
Range("A1").Select
Selection.Copy
Range("B5").Select
ActiveSheet.Paste
End Sub
```

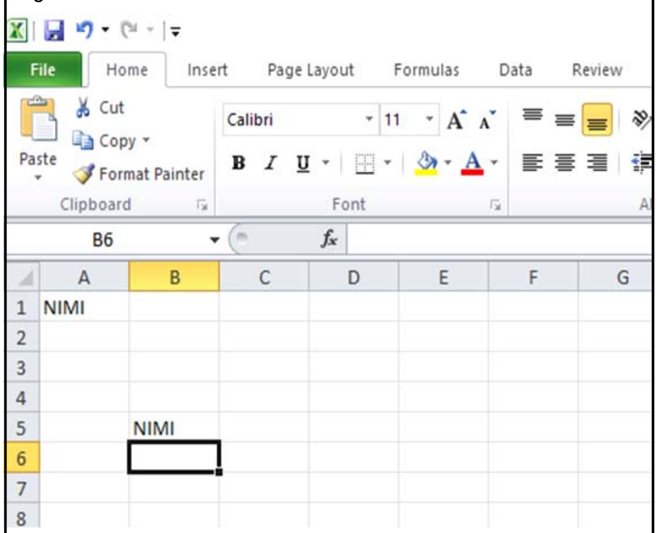
Fig 1



```
Sub sbExampleRangeMethods()
    Range("A1").Select
    Selection.Copy
    Range("B5").Select
    ActiveShee
End Sub
```

If the above code is executed the content of cell A1 is copied to Cell B5 as shown in the Fig 2.

Fig 2



	A	B	C	D	E	F	G
1	NIMI						
2							
3							
4							
5		NIMI					
6							
7							
8							

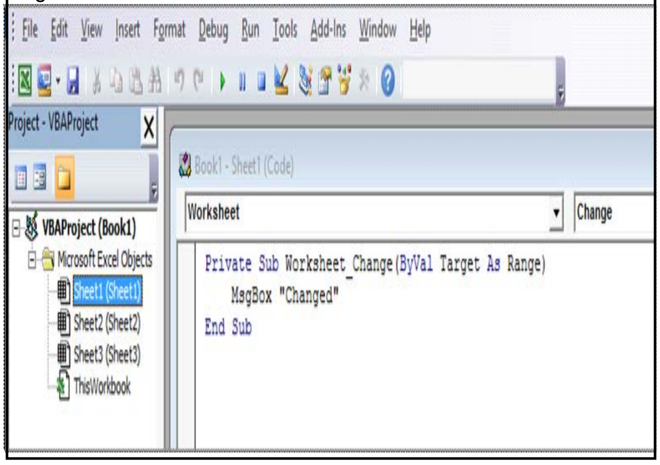
Events

An Event is an action initiated either by user action or by other VBA code. An Event Procedure is a Sub procedure that you write, according to the specification of the event, that is called automatically by Excel when an event occurs. For example, a Worksheet object has an event named Change. If you have properly programmed the event procedure for the Change event, Excel will automatically call that procedure, always named Worksheet_Change and always in the code module of the worksheet, whenever the value of any cell on the worksheet is changed by user input or by other VBA code (but not if the change in value is a result of a formula calculation). You can write code in the Worksheet_Change event procedure to take some action depending on which cell was changed or based upon the newly changed value.

Enter the following code in the Worksheet_Change event as shown in Fig 3.

```
Private Sub Worksheet_Change(ByVal Target
As Range)
MsgBox "Changed"
End Sub
```

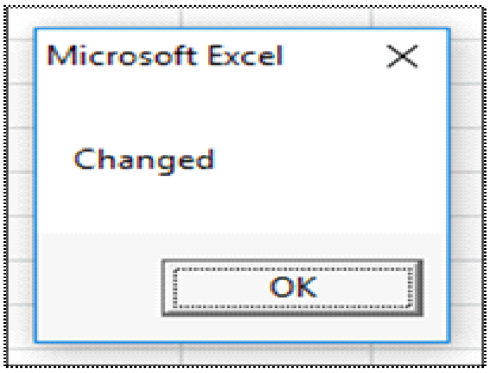
Fig 3



```
Private Sub Worksheet_Change(ByVal Target As Range)
    MsgBox "Changed"
End Sub
```

When we change content of any Cell the following message will be displayed as shown in Fig 4.

Fig 4



Microsoft Excel

Changed

Debugging Techniques in VBA

Objectives: At the end of this lesson you shall be able to
- **explain about VBA debugging**
- **explain how to set and clear the Breakpoints**
- **describe use of immediate window**
- **explain about watch window.**

VBA Debugging

In Excel 2010, VBA's debugging environment allows the programmer to momentarily suspend the execution of VBA code so that the following debug tasks can be done:

1 Check the value of a variable in its current state.

2 Enter VBA code in the Immediate window to view the results.

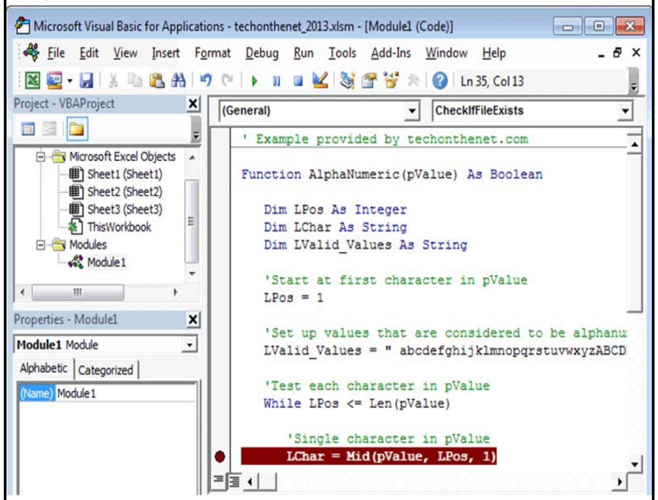
3 Execute each line of code one at a time.

4 Continue execution of the code.

5 Halt execution of the code.

These are just some of the tasks that you might perform in VBA's debugging environment. (Fig 1)

Fig 1



```
' Example provided by techonthenet.com

Function AlphaNumeric(pValue) As Boolean

   Dim LPos As Integer
   Dim LChar As String
   Dim LValid_Values As String

   'Start at first character in pValue
   LPos = 1

   'Set up values that are considered to be alphanu
   LValid_Values = " abcdefghijklmnopqrstuvwxyzABCD

   'Test each character in pValue
   While LPos <= Len(pValue)

      'Single character in pValue
      LChar = Mid(pValue, LPos, 1)
```

Breakpoint in VBA

In Excel 2010, a breakpoint is a selected line of code that once reached, the program will momentarily become suspended. Once suspended, and to use VBA's debugging environment to view the status of program, step through each successive line of code, continue execution of the code, or halt execution of the code.

And create as many breakpoints in the code as you want. Breakpoints are particularly useful when suspend the program where you suspect a problem/bug exists.

Setting a Breakpoint

First, you need to open the VBA environment. The quickest way to do this is by pressing Alt+F11 while the Excel database file is open.

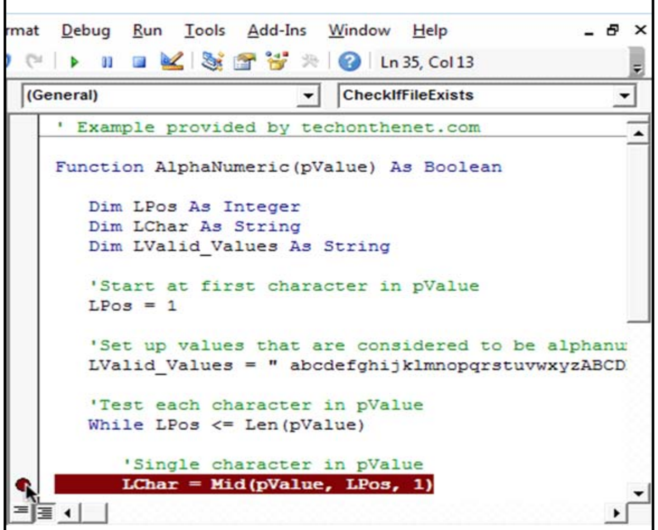
To set a breakpoint, find the line of code where to suspend your program. Left-click in the grey bar to the left of the code. A red dot should appear and the line of code should be highlighted in red.

Clear Breakpoint in VBA

A breakpoint in VBA is indicated by a red dot with a line of code highlighted in red.

To clear a breakpoint in Excel 2010, left-click on the red dot next to the line of code that has the breakpoint. (Fig 2)

Fig 2



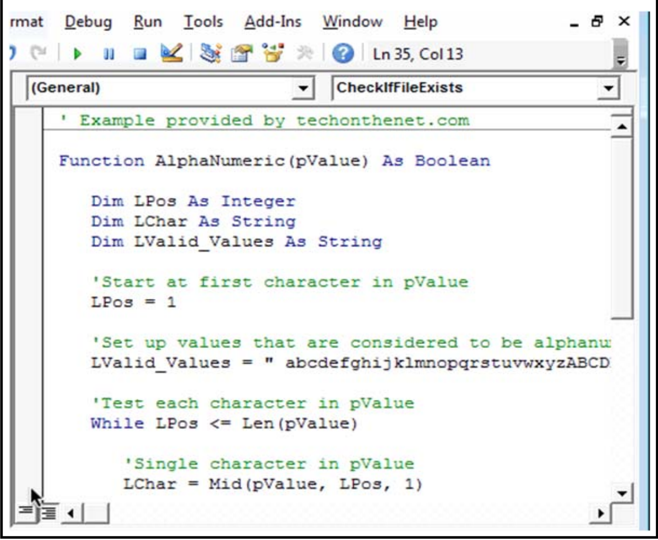
```
' Example provided by techonthenet.com

Function AlphaNumeric(pValue) As Boolean

   Dim LPos As Integer
   Dim LChar As String
   Dim LValid_Values As String

   'Start at first character in pValue
   LPos = 1

   'Set up values that are considered to be alphanu
   LValid_Values = " abcdefghijklmnopqrstuvwxyzABCD

   'Test each character in pValue
   While LPos <= Len(pValue)

      'Single character in pValue
      LChar = Mid(pValue, LPos, 1)
```

In this example, we want to clear the breakpoint at the following line of code:

LChar = Mid(pValue, LPos, 1) (Fig 3)

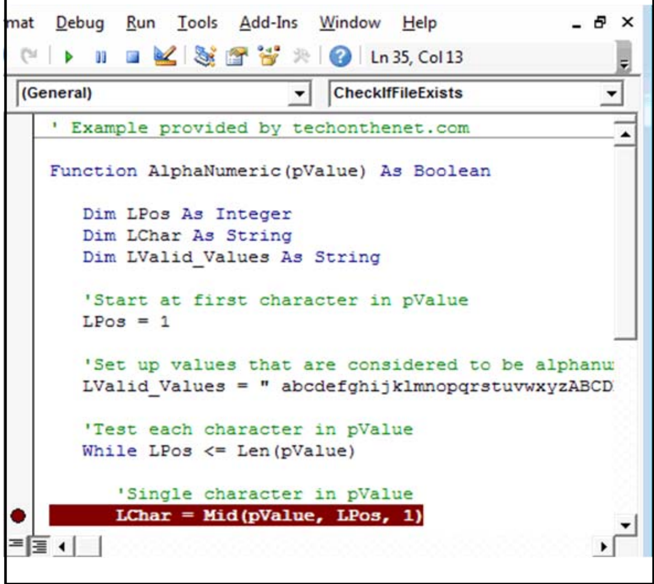
Now, the breakpoint is cleared and the line of code should look normal again. (Fig 4)

Fig 3



```
' Example provided by techonthenet.com

Function AlphaNumeric(pValue) As Boolean

   Dim LPos As Integer
   Dim LChar As String
   Dim LValid_Values As String

   'Start at first character in pValue
   LPos = 1

   'Set up values that are considered to be alphanu
   LValid_Values = " abcdefghijklmnopqrstuvwxyzABCD

   'Test each character in pValue
   While LPos <= Len(pValue)

      'Single character in pValue
      LChar = Mid(pValue, LPos, 1)
```

Fig 4



```
' Example provided by techonthenet.com

Function AlphaNumeric(pValue) As Boolean

   Dim LPos As Integer
   Dim LChar As String
   Dim LValid_Values As String

   'Start at first character in pValue
   LPos = 1

   'Set up values that are considered to be alphanu
   LValid_Values = " abcdefghijklmnopqrstuvwxyzABCD

   'Test each character in pValue
   While LPos <= Len(pValue)

      'Single character in pValue
      LChar = Mid(pValue, LPos, 1)
```

In this example, we've created a breakpoint at the following line of code:

LChar = Mid(pValue, LPos, 1)

Now, the breakpoint is cleared and the line of code should look normal again

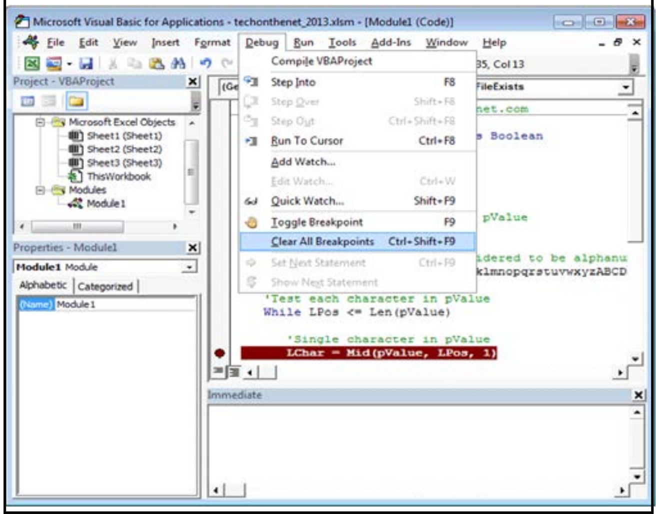
Clearing all Breakpoints

If user use as many breakpoints as you want in Excel 2010, and can save time by clearing all breakpoints in the VBA code at once.

To clear all breakpoints in the program, select "Clear All Breakpoints" under the Debug menu. (Fig 5)

This will remove all breakpoints from the VBA code, so that you don't have to individually remove each breakpoint, one by one.

Fig 5



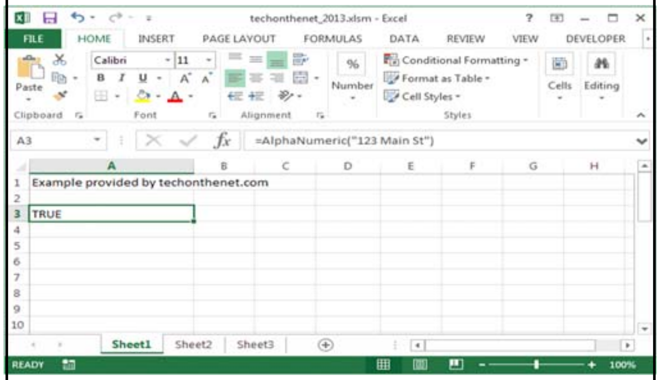
```
   'Test each character in pValue
   While LPos <= Len(pValue)

      'Single character in pValue
      LChar = Mid(pValue, LPos, 1)
```

Debug Mode

Now that we know how to set and clear breakpoints in Excel 2010, let's take a closer look at the debug mode in VBA.

In our example, we've set our breakpoint and entered our AlphaNumeric function as a formula in a cell. This will cause the VBA code to execute. (Fig 6)

Fig 6

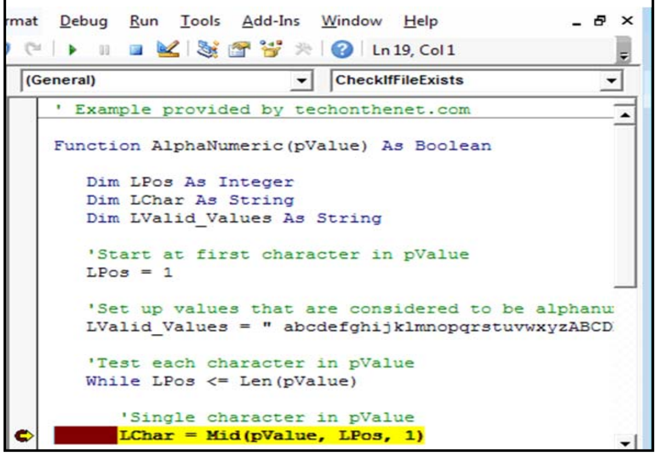

techonthenet_2013.xlsm - Excel
FILE HOME INSERT PAGE LAYOUT FORMULAS DATA REVIEW VIEW DEVELOPER
Calibri 11
Conditional Formatting
Format as Table
Cell Styles
Paste Number Cells Editing
Clipboard Font Alignment Styles
Sheet1 Sheet2 Sheet3
READY 100%

A3 =AlphaNumeric("123 Main St")

	A	B	C	D	E	F	G	H
1	Example provided by techonthenet.com							
2								
3	TRUE							
4								
5								
6								
7								
8								
9								
10								

When the breakpoint is reached, Excel will display the Microsoft Visual Basic window and highlight the line (in yellow) where the code has been suspended. (Fig 7)

Fig 7



```
' Example provided by techonthenet.com

Function AlphaNumeric(pValue) As Boolean

   Dim LPos As Integer
   Dim LChar As String
   Dim LValid_Values As String

   'Start at first character in pValue
   LPos = 1

   'Set up values that are considered to be alphanu
   LValid_Values = " abcdefghijklmnopqrstuvwxyzABCD

   'Test each character in pValue
   While LPos <= Len(pValue)

      'Single character in pValue
      LChar = Mid(pValue, LPos, 1)
```

Now we are in debug mode in our Excel spreadsheet. Now we can do any of the following:

1 Check the value of a variable in its current state.

2 Enter VBA code in the Immediate window to view the results.

3 Execute each line of code one at a time.

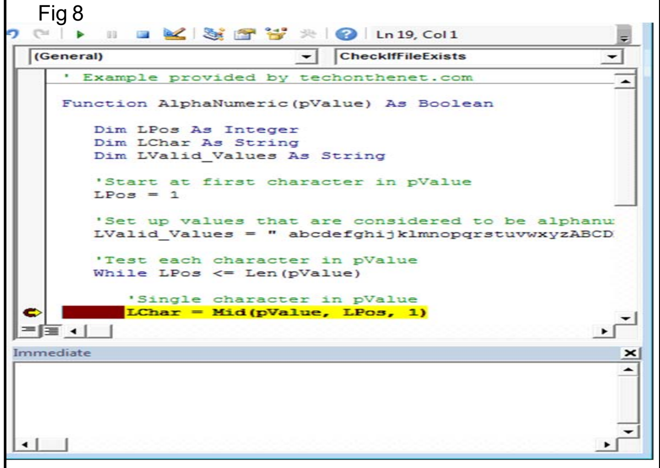
4 Continue execution of the code.

5 Halt execution of the code.

Using the Immediate Window

In Excel 2010, the Immediate window can be used to debug your program by allowing you to enter and run VBA code in the context of the suspended program. (Fig 8)

Fig 8

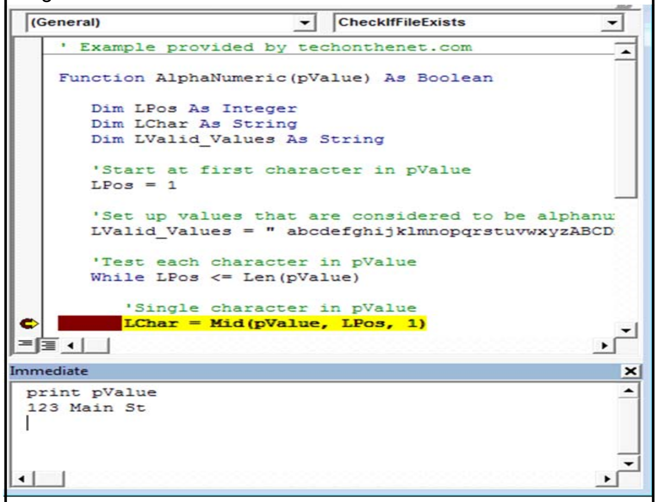


```
' Example provided by techonthenet.com

Function AlphaNumeric(pValue) As Boolean

   Dim LPos As Integer
   Dim LChar As String
   Dim LValid_Values As String

   'Start at first character in pValue
   LPos = 1

   'Set up values that are considered to be alphanu
   LValid_Values = " abcdefghijklmnopqrstuvwxyzABCD

   'Test each character in pValue
   While LPos <= Len(pValue)

      'Single character in pValue
      LChar = Mid(pValue, LPos, 1)
```

We've found the Immediate window to be the most help when we need to find out the value of a variable, expression, or object at a certain point in the program. This can be done using the print command.

For example, if you wanted to check the current value of the variable called pValue, you could use the print command as follows: (Fig 9)

Fig 9



```
' Example provided by techonthenet.com

Function AlphaNumeric(pValue) As Boolean

   Dim LPos As Integer
   Dim LChar As String
   Dim LValid_Values As String

   'Start at first character in pValue
   LPos = 1

   'Set up values that are considered to be alphanu
   LValid_Values = " abcdefghijklmnopqrstuvwxyzABCD

   'Test each character in pValue
   While LPos <= Len(pValue)

      'Single character in pValue
      LChar = Mid(pValue, LPos, 1)
```

```
print pValue
123 Main St
```

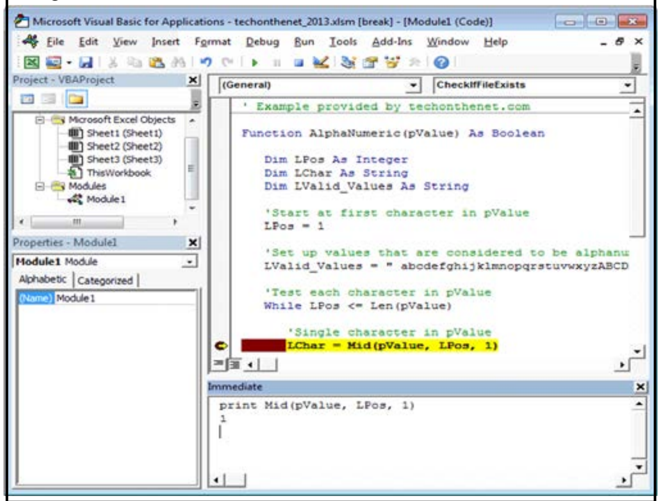
In this example, we typed **print pValue** in the Immediate window and pressed ENTER.

Print pValue

The Immediate window displayed the result in the next line. In this case, the print pValue command returned **123 Main St**.

You can also type more complicated expressions in the Immediate window. (Remember to press ENTER.) For example: (Fig 10)

Fig 10



```
' Example provided by techonthenet.com

Function AlphaNumeric(pValue) As Boolean

   Dim LPos As Integer
   Dim LChar As String
   Dim LValid_Values As String

   'Start at first character in pValue
   LPos = 1

   'Set up values that are considered to be alphanu
   LValid_Values = " abcdefghijklmnopqrstuvwxyzABCD

   'Test each character in pValue
   While LPos <= Len(pValue)

      'Single character in pValue
      LChar = Mid(pValue, LPos, 1)
```

```
print Mid(pValue, LPos, 1)
1
```

In this example, we typed **print Mid(pValue, LPos, 1)** in the Immediate window and pressed ENTER.

print Mid(pValue, LPos, 1)

The Immediate window displayed the result of **1** in the next line.

The Immediate window can be used to run other kinds of VBA code, but bear in mind that the Immediate window can only be used when debugging so any code that you run is for debugging purposes only. The code entered in the Immediate window does not get saved and added to the existing VBA code

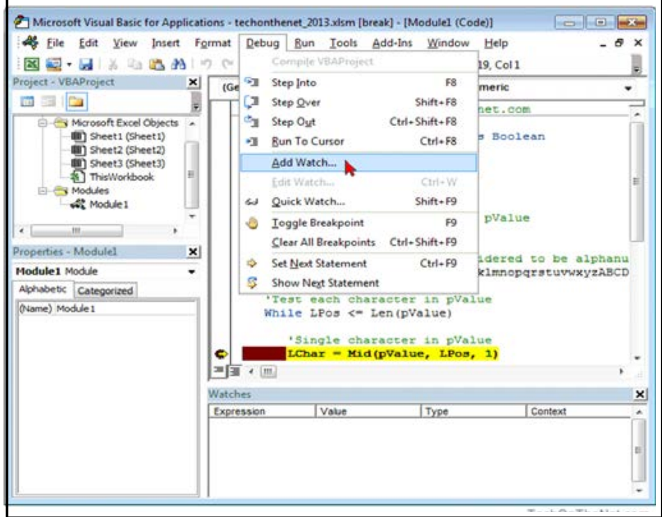
Adding a Watch Expression

The Watch Window displays the value of a watched expression in its current state. This can be extremely useful when debugging VBA code. Let's explore how to add an expression to the Watch Window.

To add a Watch expression, select **Add Watch** under the **Debug** menu. (Fig 11)

When the *Add Watch* window appears, enter the expression to watch and click the OK button when you are done. (Fig 12)

Fig 11

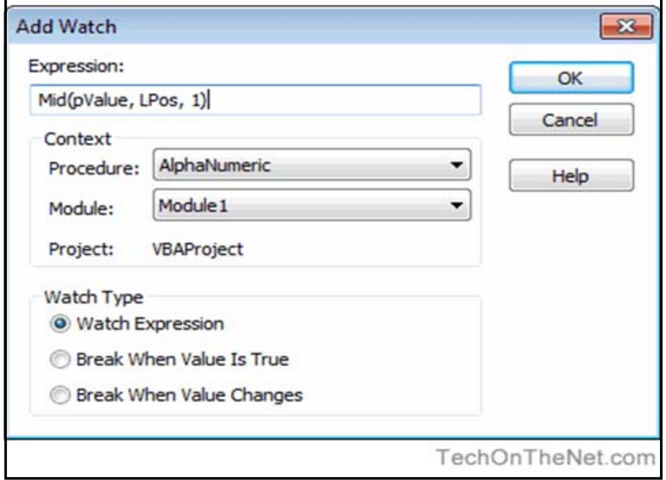

Microsoft Visual Basic for Applications - techonthenet_2013.xlsm [break] - [Module1 (Code)]
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
Compile VBAProject
Step Into F8
Step Over Shift+F8
Step Out Ctrl+Shift+F8
Run To Cursor Ctrl+F8
Add Watch...
Edit Watch... Ctrl+W
Quick Watch... Shift+F9
Toggle Breakpoint F9
Clear All Breakpoints Ctrl+Shift+F9
Set Next Statement Ctrl+F9
Show Next Statement
Project - VBAProject
Properties - Module1
Watches
Expression Value Type Context

```
    'Test each character in pValue
    While LPos <= Len(pValue)

        'Single character in pValue
        LChar = Mid(pValue, LPos, 1)
```

In this example, we've entered the following watch expression in the **Expression** field:

Mid(pValue, LPos, 1)

Fig 12



Expression: Mid(pValue, LPos, 1)

Context
Procedure: AlphaNumeric
Module: Module1
Project: VBAProject

Watch Type
[✓ Watch Expression]
Break When Value Is True
Break When Value Changes

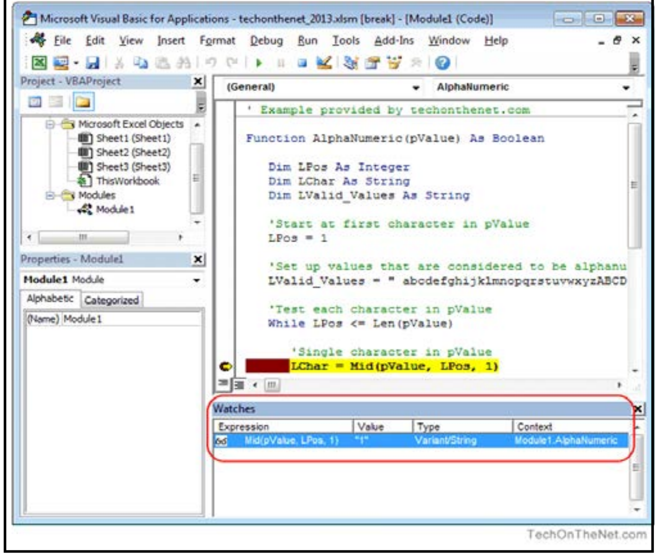
Next, we've selected AlphaNumeric as the **Procedure** and Module1 as the **Module** when setting up the **Context** for the watched expression.

Finally, we've selected *Watch Expression* as the **Watch Type** but there are 3 options to choose from:

Watch Type	Description
Watch Expression	To display the value of the watched expression in its current state
Break When Value Is True	To stop the execution of the code when the value of the watched expression is True
Break When Value Changes	To stop the execution of the code when the value of the watched expression changes

When return to the VBA window, the Watch Window will automatically appear if it was previously hidden. Within the Watch Window, all of the watched expressions should be listed including the one that we just added. (Fig 13)

Fig 13


Microsoft Visual Basic for Applications - techonthenet_2013.xlsm [break] - [Module1 (Code)]
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
Project - VBAProject
Properties - Module1
(General)
AlphaNumeric
TechOnTheNet.com

```
' Example provided by techonthenet.com

Function AlphaNumeric(pValue) As Boolean

    Dim LPos As Integer
    Dim LChar As String
    Dim LValid_Values As String

    'Start at first character in pValue
    LPos = 1

    'Set up values that are considered to be alphanu
    LValid_Values = " abcdefghijklmnopqrstuvwxyzABCD

    'Test each character in pValue
    While LPos <= Len(pValue)

        'Single character in pValue
        LChar = Mid(pValue, LPos, 1)
```

Watches

Expression	Value	Type	Context
Mid(pValue, LPos, 1)	"1"	Variant/String	Module1.AlphaNumeric

As you can see, the expression Mid(pValue, LPos, 1) now appears in the Watch Window with a value of "1". Adding a watch is a great way to keep track of variables or expressions of interest when debugging the VBA code.

Object Oriented Programming concepts, Concepts of classes, Objects, properties and Methods

Objectives: At the end of this lesson you shall be able to
- **explain Class and objects and its features**
- **explain VBA Class modules Versus VBA normal modules**
- **list out parts of a class module and its properties**
- **explain class module events.**

Introduction

VBA Class Modules allow the user to create their own objects. In languages such as C# and Java, **classes** are used to create objects. **Class Modules** are the VBA equivalent of these classes. The major difference is that VBA Class Modules have a very limited type of Inheritance* compared to classes in the other languages. In VBA, Inheritance works in a similar way to Interfaces in C#\Java.

In VBA we have built-in objects such as the Collection, Workbook, Worksheet and so on. The purpose of VBA Class Modules is to allow us to custom build our own objects.

Let's start this post by looking at why we use objects in the first place.

Inheritance is using an existing class to build a new class.
Interfaces are a form of Inheritance that forces a class to implement specifics procedures or properties.

Objects

Using objects allows us to build our applications like we are using building blocks.

The idea is that the code of each object is self-contained. It is completely independent of any other code in our application.

Advantages of Using Objects

Treating parts of our code as blocks provide us with a lot of advantages

1 It allows us to build an application one block at a time.

2 It is much easier to test individual parts of an application.

3 Updating code won't cause problems in other parts of the application.

4 It is easy to add objects between applications.

Disadvantages of Using Objects

With most things in life there are pros and cons. Using VBA class modules is no different. The following are the disadvantages of using class module to create objects

1 It takes more time *initially* to build applications*.

2 It is not always easy to clearly define what an object is.

3 People new to classes and objects can find them difficult to understand at first.

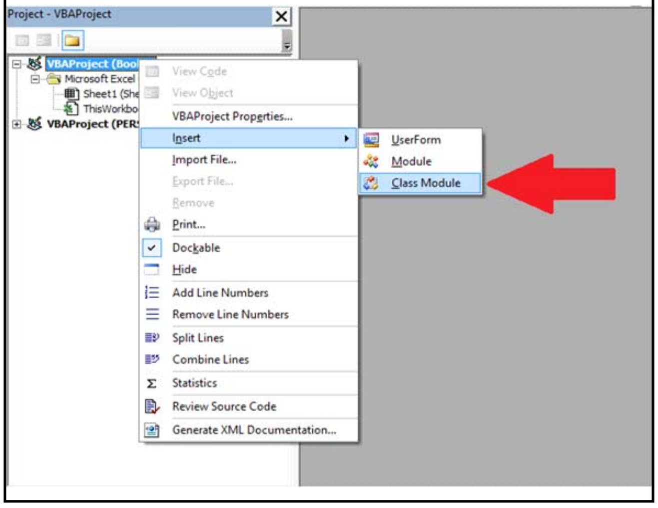
If create an application using objects it will take longer to create it initially have to spend more time planning and designing it. However, in the long run it will save a huge amount of time. The code will be easier to manage, update and reuse.

Creating a Simple Class Module

Let's look at a very simple example of creating a class module and using it in our code.

To create a class module we right-click in the Project window and then select **Insert** and **Class Module.** (Fig 1)

Fig 1



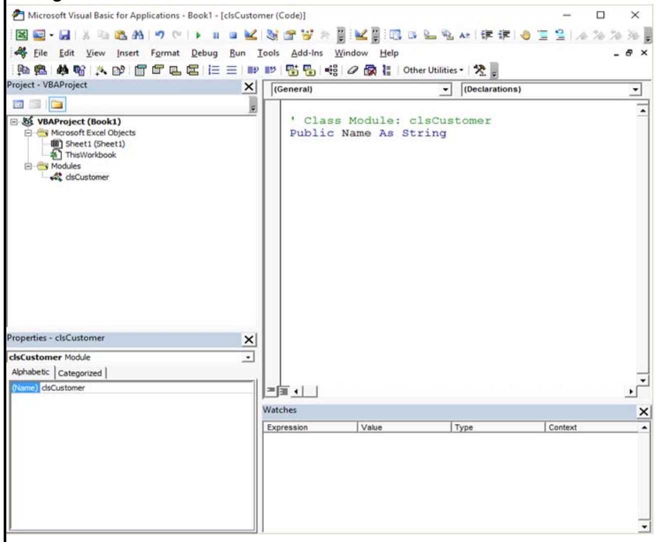
Adding a Class Module

Our new class is called **Class1**. We can change the name in the Properties window.

Let's change the name of the class module to **clsCustomer**. Then we will add a variable to the class module like this

Public Name **AsString (Fig 2)**

Fig 2



```
' Class Module: clsCustomer
Public Name As String
```

We can use now use this class module in any module(standard or class) in our workbook. For example

```
' Create the object from the class module

Dim oCustomer AsNew clsCustomer

' Set the customer name

oCustomer.Name = "John"

' Print the name to the Immediate Window(Ctrl + G)

Debug.Print oCustomer.Name
```

Class Module versus Objects

People who are new to using classes and VBA class modules, often get confused between what is a class and what is an object.

Let's look at a real-world example. Think of a mass-produced item like a coffee mug. A design of the mug is created first. Then, thousands of coffee mugs are created from this design.

This is similar to how class modules and objects work.

The **class module** can be thought of as the design.

The **object** can be thought of as the item that is created from the design.

The **New** keyword in VBA is what we use to create an object from a class module. For example

```
' Creating objects using new

Dim oItem AsNew Class1

Dim oCustomer1 AsNewclsCustomer

Dim coll AsNew Collection
```

Note: We don't use New with items such as Workbooks and Worksheets. See <u>When New is not required</u> for more information.

VBA Class Modules Versus VBA Normal Modules

Writing code in a class module is almost the same as writing code in a normal module. We can use the same code we use in normal modules. It's how this code is used which is very different.

Let's look at the two main differences between the class and normal module. These often cause confusion among new users.

Difference 1 – How the modules are used

If want to use a sub/function etc. from a class module must create the object first.

For example, imagine we have two identical **PrintCustomer** subs. One is in a class module and one is in a normal module…

```
' CLASS MODULE CODE - clsCustomer

Public Sub PrintCustomer()

Debug.Print "Sample Output"

End Sub

' NORMAL MODULE CODE

Public Sub PrintCustomer()

Debug.Print "Sample Output"

End Sub
```

You will note the code for both is exactly the same.

To use the **PrintCustomer** sub from the class module, you must first create an object of that type

```
' Other Module

Sub UseCustomer()

Dim oCust AsNew clsCustomer

oCust.PrintCustomer

EndSub
```

To use **Print Customer** from the normal module you can call it directly

‘ Other Module

Sub Use Customer()

Print Customer

End Sub

Difference 2 – Number of copies

Whencreate a variable in a normal module there is only one copy of it. For a class module, there is one copy of the variable for each object you create.

For example, imagine we create a variable **Student Name** in both a class and normal module..

‘ NORMAL MODULE

Public StudentName **As String**

‘ CLASS MODULE

Public Studen tName **As String**

For the normal module variable there will only be one copy of this variable in our application.

StudentName = “Ram”

For the class module a new copy of the variable **Student Name** is created each time a new object is created.

Dim student1 **As New** clsStudent

Dim student 2 **As New** clsStudent

student1.Student Name = “Bill”

student2.Student Name = “Ted”

When fully understand VBA class modules, these differences will seem obvious.

The Parts of a Class Module

There are four different items in a class module. These are

1 **Methods** – functions/subs.

2 **Member variables** – variables.

3 **Properties**– types of functions/subs that behave like variables.

4 **Events** – subs that are triggered by an event.

And can see they are all either functions, subs or variables.

Let’s have a quick look at some examples before we deal with them in turn

‘ CLASS MODULE CODE

‘ Member variable

Private dBalance **As Double**

‘ Properties

Property Get Balance () **AsDouble**

Balance = dBalance

EndProperty

Property Let Balance(dValue**As** Double)

dBalance = dValue

End Property

‘ Event - triggered when class created

Private Sub Class_Initialize()

dBalance = 100

EndSub

‘ Methods

Public Sub Withdraw (dAmount**As** Double)

dBalance = dBalance - dAmount

End Sub

Public Sub Deposit (dAmount**As** Double)

dBalance = dBalance + dAmount

EndSub

Now that we have seen examples, let’s look at each of these in turn.

Class Module Methods

Methods refer to the procedures of the class. In VBA procedures are subs and functions. Like member variables they can be Public or Private.

Let’s look at an example

‘ CLASS MODULE CODE

‘ Class name: clsSimple

‘ Public procedures can be called from outside the object

Public Sub PrintText (sText**As** String)

Debug.PrintsText

EndSub

Public Function Calculate (dAmount**As** Double) **As Double**

Calculate = dAmount - GetDeduction

End Function

‘ private procedures can only be called from within the Class Module

Private Function GetDeduction () **As Double**

GetDeduction = 2.78

EndFunction

We can use the **clsSimple** class module like this

Sub Class Members ()

Dim oSimple **As New** clsSimple

oSimple.PrintText “Hello”

Dim dTotal **As Double**

dTotal = oSimple.Calculate(22.44)

Debug.Print dTotal

EndSub

Class Module Member Variables

The member variable is very similar to the normal variable we use in VBA. The difference is we use **Public** or **Private** instead of **Dim**.

‘ CLASS MODULE CODE

Private Balance **AsDouble**

Public AccountID **As String**

Note: Dim and Private do exactly the same thing but the convention is to use Dim in sub/ functions and to use Private outside sub/ functions.

The **Public** keyword means the variable can be accessed from outside the class module. For example

Dim oAccount **AsNew** clsAccount

‘ Valid - AccountID is public

oAccount.AccountID = “499789”

‘ Error - Balance is private

oAccount.Balance = 678.90

In the above example we cannot access **Balance** because it is declared as **Private**. We can only use a **Private** variable within the class module. We can use in a function/ sub in the class module e.g.

‘ CLASS MODULE CODE

Private Balance **As Double**

Public Sub SetBalance()

    Balance = 100

Debug.Print Balance

End Sub

It is considered poor practice to have public member variables. This is because the code allowing outside the object to interfere with how the class works. The purpose of the using classes is so that hide what is happening from the caller.

To avoid the user directly talking to the member variables we use Properties.

Class Module Properties

1 **Get** – returns an object or value from the class

2 **Let** – sets a value in the class

3 Set – sets an object in the class

Format of VBA Property

The normal format for the properties are as follows:

Public Property Get () **AsType**

End Property

Public Property Let (varname**AsType**)

End Property

Public PropertySet (varname**AsType**)

EndProperty

We have seen already that the Property is simply a type of sub. The purpose of the Property is to allow the caller to get and set values.

Use of Properties

Imagine we have a class that maintains a list of Countries. We could store the list as an array

‘ Use array to store countries

Public arrCountries **As Variant**

‘ Set size of array when class is initialized

Private Sub Class_Initialize()

ReDim arrCountries (1 **To** 1000)

End Sub

When the user wants to get the number of countries in the list they could do this

‘ NORMAL MODULE CODE

Dim oCountry **As New** clsCountry

‘ Get the number of items

NumCountries = UBound(oCountry.arrCountries) + 1

There are two major problems with the above code

1 To get the number of countries you need to know how the list is stored e.g. Array.

2 If we change the Array to a Collection, we need to change all code that reference the array directly.

To solve these problems we can create a function to return the number of countries

‘ CLASS MODULE CODE - clsCountryList

‘ Array

Private arrCountries () **As String**

Public Function Count () **AsLong**

  Count = UBound(arrCountries) + 1

End Function

We then use it like this

‘ MODULE CODE

Dim oCountries **As New** clsCountries

Debug.Print “Number of countries is “ &oCountries.Count

This code solves the two problems we listed above. We can change our Array to a Collection and the caller code will still work e.g.

‘ CLASS MODULE CODE

‘ Collection

Private collCountries() **As** Collection

Public Function Count() **AsLong**

  Count = collCountries.Count

End Function

The caller is oblivious to how the countries are stored. All the caller needs to know is that the **Count** function will return the number of countries.

As we have just seen, a sub or function provides a solution to the above problems. However, using a **Property** can provide a more elegant solution.

Using a Property instead of a Function/Sub

Instead of the creating a **Count** Function we can create a **Count** Property. As you can see below they are very similar

‘ Replace this

Public Function Count() **As Long**

  Count = UBound(arrCountries) + 1

End Function

‘ With this

Property Get Count () **As Long**

  Count = UBound(arrCountries) + 1

End Function

In this scenario, there is not a lot of difference between using the Property and using a function. However, there are differences. We normally create a **Get** and **Let** property like this

‘ CLASS MODULE CODE - clsAccount

Privated TotalCost **As Double**

Property Get TotalCost () **As Long**

Total Cost= dTotalCost

End Property

Property Let Total Cost (dValue **As** Long)

dTotal Cost = dValue

End Property

Using **Let** allows us to treat the property like a variable. So we can do this

oAccount.Total Cost = 6

The second difference is that using **Let** and **Get** allows us to use the same name when referencing the Get or Let property. So we can use the property like a variable. This is the purpose of using Properties over a sub and function.

oAccount.TotalCost = 6

dValue = oAccount.TotalCost

If we used a function and a sub then we cannot get the behaviour of a variable. Instead we have to call two different procedures e.g.

oAccount.SetTotalCost 6

dValue = oAccount.GetTotalCost

You can also see that when we used Let we can assigned the value like a variable. When we use **Set Total Cost**, we had to pass it as a parameter.

The Property in a Nutshell

1 The Property hides the details of the implementation from the caller.

2 The Property allows us to provide the same behaviour as a variable.

Types of VBA Property

There are three types of Properties. We have seen Get and Let already. The one we haven't looked at is **Set**.

Set is similar to **Let** but it is used for an object(see Assigning VBA Objects for more detail about this).

Originally in Visual Basic, the **Let** keyword was used to assign a variable. In fact, we can still use it if we like.

' These line are equivalent

Let a = 7

a = 7

So we use **Let** to assign a value to a variable and we use **Set** to assign an object to an object variable

' Using Let

Dim a **As Long**

Let a = 7

' Using Set

Dim coll1 **As** Collection, coll2 **As** Collection

Set coll1 = **New** Collection

Set coll2 = coll1

- **Let** is used to assign a value to a basic variable type.
- **Set** is used to assign an object to an object variable.

In the following example, we use **Get** and **Let** properties for a string variable

' CLASS MODULE CODE

' SET/LET PROPERTIES for a variable

Private m_sName **As String**

' Get/Let Properties

Property Get Name() **As String**

  Name = m_sName

End Property

Property Let Name (sName**As** String)

m_sName = sName

End Property

We can then use the **Name** properties like this

Sub Test Let Set()

Dim sName **As String**

Dim coll **As New** Collection

Dim oCurrency **As New** clsCurrency

' Let Property

oCurrency.Name = "USD"

' Get Property

sName = oCurrency.Name

End Sub

In the next example, we use **Get** and **Set** properties for an object variable

' CLASS MODULE CODE

Private m_collPrices **As** Collection

' Get/Set Properties

Property Get Prices() **As** Collection

Set Prices = m_collPrices

End Property

Property Set Prices (collPrices**As** Collection)

Set m_collPrices = collPrices

End Property

We can then use the properties like this

Sub Test Let Set ()

Dim coll1 **As New** Collection

Dim oCurrency **As New** cls Currency

‘ Set Property

Set oCurrency.Prices = coll1

‘ Get Property

Dim coll2 **As** Collection

Set Coll2 = oCurrency.Prices

EndSub

We use the Get property to return the values for both items. Notice that even though we use the **Get** Property to return the Collection, we still need to use the **Set** keyword to assign it.

Class Module Events

A class module has two events

1 **Initialize** – occurs when a new object of the class is created.

2 **Terminate** – occurrs when the class object is deleted.

In Object Oriented languages like C++, these events are referred to as the **Constructor** and the **Destructor**. In most languages, you can pass parameters to a constructor but in VBA you cannot. We can use a **Class Factory** to get around this issue as we will see below.

Initialize

Let’s create a very simple class module called clsSimple with **Initialize** and **Terminate** events

‘ CLASS MODULE CODE

Private Sub Class_Initialize()

Msg Box “Class is being initialized”

End Sub

Private Sub Class_Terminate()

Msg Box “Class is being terminated”

End Sub

Public Sub Print Hello ()

Debug.Print “Hello”

End Sub

In the following example, we use **Dim** and **New** to create the object.

In this case, **oSimple** is not created until we reference it for the first time e.g.

Sub Class Event sInit2 ()

Dim oSimple **As New** clsSimple

‘ Initialize occurs here

oSimple.PrintHello

EndSub

When we use **Set** and **New** together the behaviour is different. In this case the object is created when **Set** is used e.g.

Sub Class Events Init()

Dim oSimple **As** clsSimple

‘ Initialize occurs here

Set oSimple = **New** clsSimple

oSimple.PrintHello

End Sub

Note: For more information about the different between using New with Dim and using New with Set see Subtle Differences of Dim Versus Set

As said earlier, you cannot pass a parameter to **Initialize**. If you need to do this you need a function to create the object first

‘ CLASS MODULE - clsSimple

Public Sub Init (Price **As** Double)

EndSub

‘ NORMAL MODULE

PublicSubTest()

‘ Use CreateSimpleObject function

Dim oSimple **As** clsSimple

Set oSimple = CreateSimpleObject(199.99)

End Sub

Public Function CreateSimpleObject(Price **As** Double) **As** clsSimple

Dim oSimple **As New** clsSimple

oSimple.Init Price

Set CreateSimpleObject = oSimple

End Function

We will expand on this CreateSimpleObject in <u>Example 2</u> to create a **Class Factory**.

Terminate

The Terminate event occurs when the class is deleted. This happens when we set it to **Nothing**

Sub Class EventsTerm ()

Dim oSimple **As** clsSimple

Set oSimple = **New**clsSimple

‘ Terminate occurs here

Set oSimple = **Nothing**

End Sub

If we don’t set the object to **Nothing** then VBA will automatically delete it when it goes out of scope.

What this means is that if we create an object in a procedure, when that procedure ends VBA will delete any objects that were created.

Sub Class EventsTerm2()

Dim oSimple **As New** clsSimple

‘ Initialize occurs here

oSimple.PrintHello

‘ oSimple is deleted when we exit this Sub calling Terminate

EndSub