# CHAPTER NINE
# VISUAL BASIC IN EXCEL (VBA)

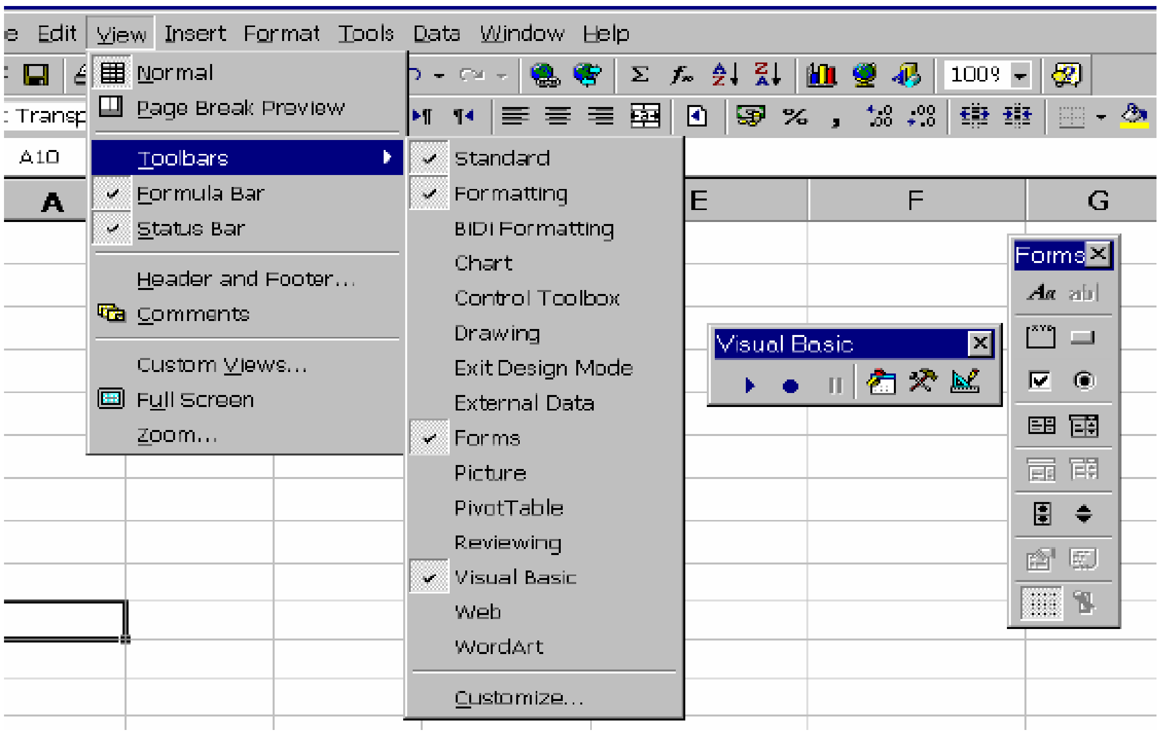
**What is VBA?**

. VBA stands for Visual Basic for Application.

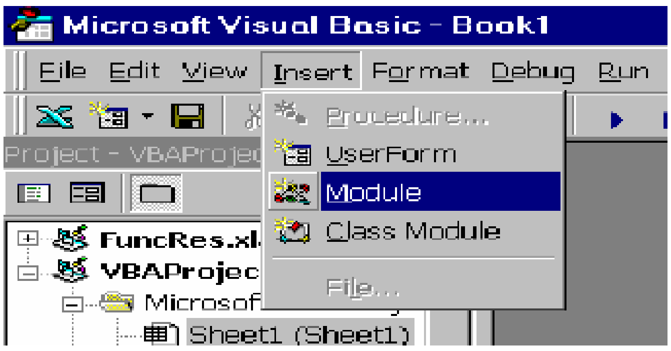
. VBA is the tool you use to develop *macros* (or *programs*) that control Excel in many different ways.

**<u>An introduction to Visual Basic Programming in excel</u>**

1. Choose **View** ⇨ **Toolbars** submenu ⇨ activate **Forms** and **Visual Basic** tool bars.
2. In the **Visual Basic** tool bar the icon to the the LEFT of the "hammer and wrench" is the **Visual Basic Editor.**



3. Click on this to enter the Visual Basic Environment.
4. Go to the **Insert** pull down menu
5. Choose **Module** from the list of options



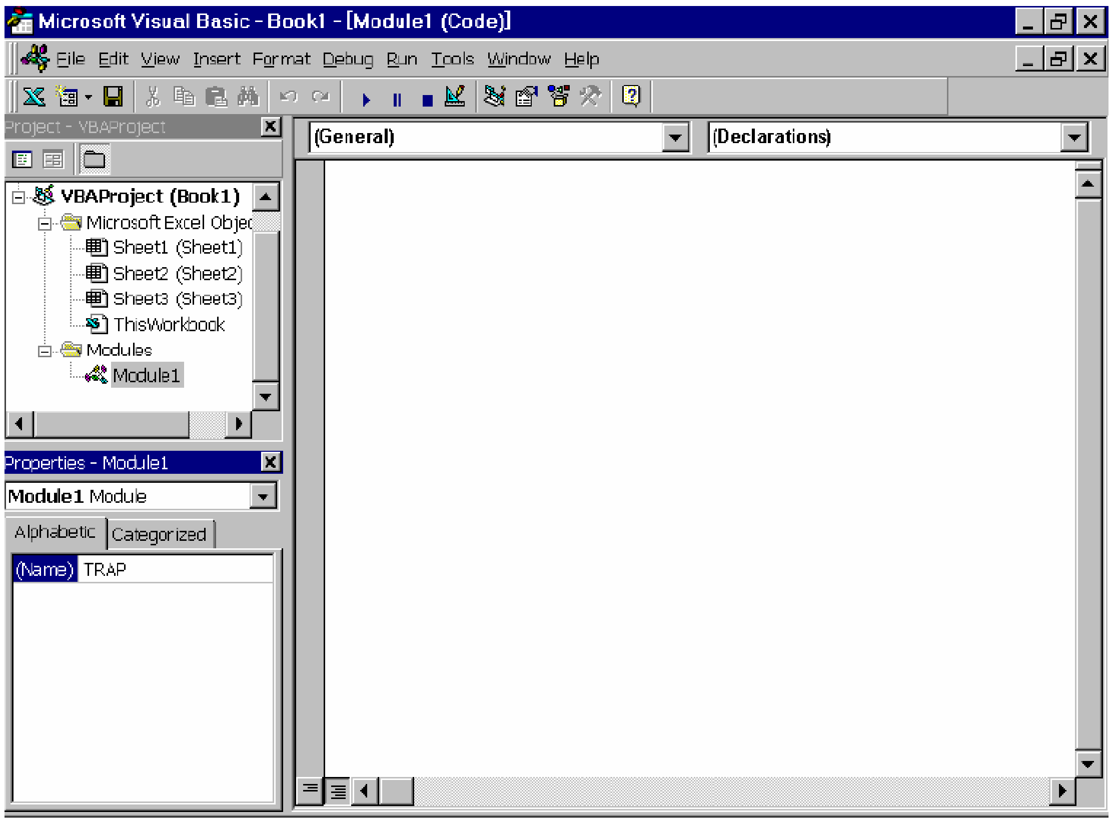
6. This will arrive at a multi part window:

• On the left-hand side you will see two small windows. The top left window gives an overview of all the elements in the visual basic environment

• The bottom left window gives the properties of what ever is highlighted in the top left window

Make sure **Module1** is highlighted in the **Project-VBA Project** sub-window.

• Go to the **(Name)** field in the bottom window and change the name to something more appropriate i.e., TRAP



(Name) TRAP

## <u>Programming:</u>

***STEP(1)***

The first and second lines of your visual basic program should be (1) The name of the program and (2) The End statement--- i.e.

**Sub Trap()**

**End Sub**

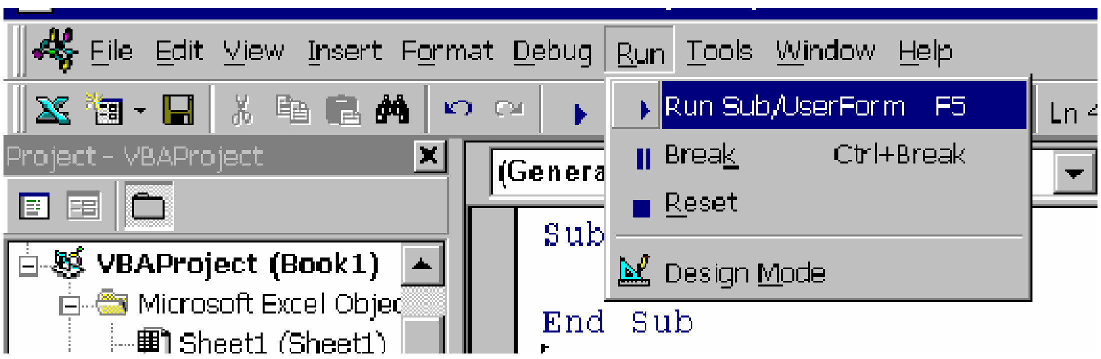
The first line says that you are going to write a *subroutine* called "Trap". The second line will be added automatically.

***STEP (2). RUNNING THE PROGRAM***

Although it will not do anything we can run our program

*Method 1. Development Mode.*

• Choose **Run** ⇨ **Run Sub/User Form**



• This is a good option when you are testing the program

• IF YOU MAKE AN ERROR IN the LOGIC, THE PROGRAM will stop running and a dialog box will appear.

. SELECTING the **debug** option in this box will point out the line that is wrong. You can then correct the line and reset the program.

• DO this by going to the **run** and selecting the **reset**

Method 2. Normal Mode.

• Return to **Excel** (in the **Visual Basic Editor**) click on the X icon on the far left of the second menu bar.

__**TIP**__ It is a good idea to rename the sheet that you will use to communicate with the visual basic program. Data will be entered in the cells in that sheet and read by the visual basic program. In turn, after processing the program will return data assigned cells in the spreadsheet.

To name control sheet

- select the sheet **tab**
- hit the right mouse button
- choose the rename option
- name the sheet.

***STEP (3)***

Next type some COMMENTS these are little memos to yourself. To type a comment at the start of the line place a single quote (‘).

**Sub Trap ()**

' This program was written in 2007

' Its purpose is to evaluate an integral using the trapezoidal rule

**End Sub**

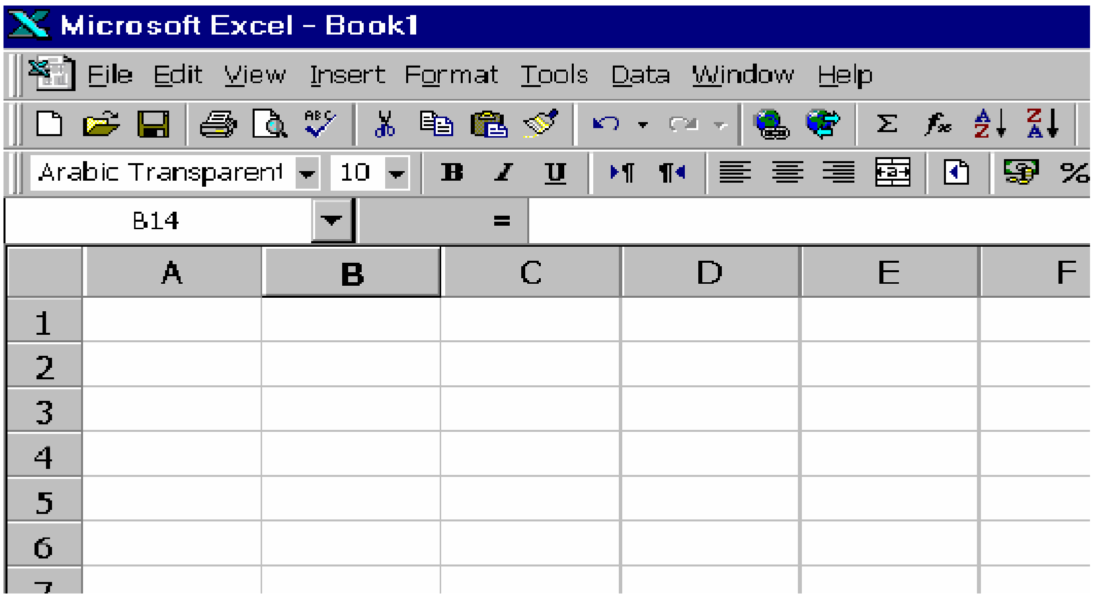
***STEP (4). Communicating between a sheet and a program***

• The key step in using visual basic in excel is the ability to pass information from a sheet to the program and visa versa.

• We use an array called CELLS (Row, Column)

• *For example*: on a sheet CELLS (5,11) refers to the cell with address D5.

• If you look at a given sheet the value in cell D5 is located in the 5th row and the 4th column.



***Example:***

1. Read a value from spread sheet “control” cell A1
2. Store that value in the program under the label N
3. Output the value of N back to the sheet “control” and place it in cell B10

**Sub Trap()**

' This program was written in 2007

' Its purpose is to evaluate an integral using the trapezoidal rule

' Read in number of steps from sheet "Control"

N = Sheets("Control").Cells(1, 1)

' The first element on the right hand side tells the program to look on the spreadsheet called "control"

' The second element on the right hand side tells the program to look in cell A1

' The program stores this value under the label N

' Output number of steps back to Spread sheet control

Sheets("Control").Cells(10, 2) = N

' This makes the program put the current value stored under the 'label N into the location B10 on the spreadsheet called control

**End Sub**

. This new code can be readily checked by clicking the “magic Button” on the control page.

***STEP (5) Variable types.***

**Understanding variables**

. A variable is simply a named storage location in your computer memory.

. Make the variable names as descriptive as possible

. You as sign a value to a variable using the equal sign operator

***Examples*:**

X=1

X= X +1

Username = “Bob Johnson”

**General Rules:**

• You can use letters, numbers and some characters, but the first character must be a letter.

• You cannot use any *spaces* or *periods* in a variable name.

• VBA does not distinguish between *uppercase* and *lowercase* letters.

• You cannot use the following characters: #, $, %, &, or !

**What are variable types?**

. Variable type refers to the manner in which a program stores data in memory – for example, as integer, real numbers, or strings.

. VBA has a variety of built-in data types. The following table lists the most common types of data that VBA can handle.

. Although VBA can take care of these details automatically, it does so at a cost.

. Letting VBA handle your data results in slower execution and inefficient use of memory especially with complex programs.

**<u>TIP</u>** when VBA is working with data, execution speed depends in the number of bytes VBA must handle. The fewer bytes used by the data, the faster VBA can access and manipulate data.

**VBA Built-In Data Types**

| *Data Type* | *Bytes Used* | *Range of Values* |
|---|---|---|
| Integer | 2 | -32,768 to 32,767 |
| Long | 4 | -2,147,483,648 to 2,147,483,647 |
| Single | 4 | -3.402823E38 to 1.401298E45 |
| Double (negative) | 8 | -1.797693134E308 to –4.94065645E-324 |
| Double (positive) | 8 | 4.94065645E-324 to 1.797693134E308 |
| String | 1 per character | Varies |
| Variant | Varies | Any data type |

## <u>Declaring variables</u>

If you don’t declare the data type for a variable, VBA assigns the default data type, variant.

. Variant causes VBA to repeatedly perform time- consuming checks and reserve more memory than necessary.

. If VBA knows a variable’s data type, it doesn’t have to investigate, and it can reserve just enough memory to store the data.

To force yourself to declare all variables you use, include the following at the first statement in your VBA module:

Option Explicit

This statement causes the program to stop whenever VBA founds a new variable that has not been declared. This is very useful if you misspell a variable so VBA will declare it as a new variable.

**. In our Trap program we will need:**

**<u>Two *integer* variable:</u>**

I (a counter) and N (the number of Strips);

**<u>Floating point (*real*) variables for:</u>**

. The“X” values

. The function heights “Fleft” and “Fright”

. The Area.

. At the start of the program we write the lines:

```
'Variable Declaration

 Dim I As Integer        ' a counter
 Dim N As Integer        ' the number of strips
 Dim Area As Single      ' Area under curve
 Dim Xleft As Single     ' value of x at left side of trap.
 Dim Xright As Single    ' value of x at right side of trap.
 Dim Fleft As Single     ' value of y at left side of trap.
 Dim Fright As Single    ' value of y at right side of trap.
```

***Step (6). Loop***

. Often in programming we need to Repeat operations many times.

.One way to do this is with – FOR LOOP,

*F – Next loops*

```
Dim I As Integer
Dim N As Integer

For  I = -3 To 1 Step 2
     N=N+1
Next I

Next Line
```

.This loop sets I = -3:

Carries out the equation N=N+1, so N=1

. Increments I by 2 to the value –1:

Repeats the equation, so N=2 now.

. Increments I to the value 1:

Repeats the equation, so N=3 now.

**GO TO NEXT LINE**

Tip

We can also step backwards, i.e.,

```
I = 4 To 1 Step –1
     N=N+1
Next I
```

Another way to do this is with – DO LOOP,

***Do*- Loops**

*Do- While and Do-Until loop*

Loop continues until a specified condition is met, e.g.,

```
N = 0
Do while I < 100
      N = N + I
Loop
```

Or

```
N = 0
Do Until I >= 100
      N = N + I
Loop
```

As long as I is less than 100, the loop will run.

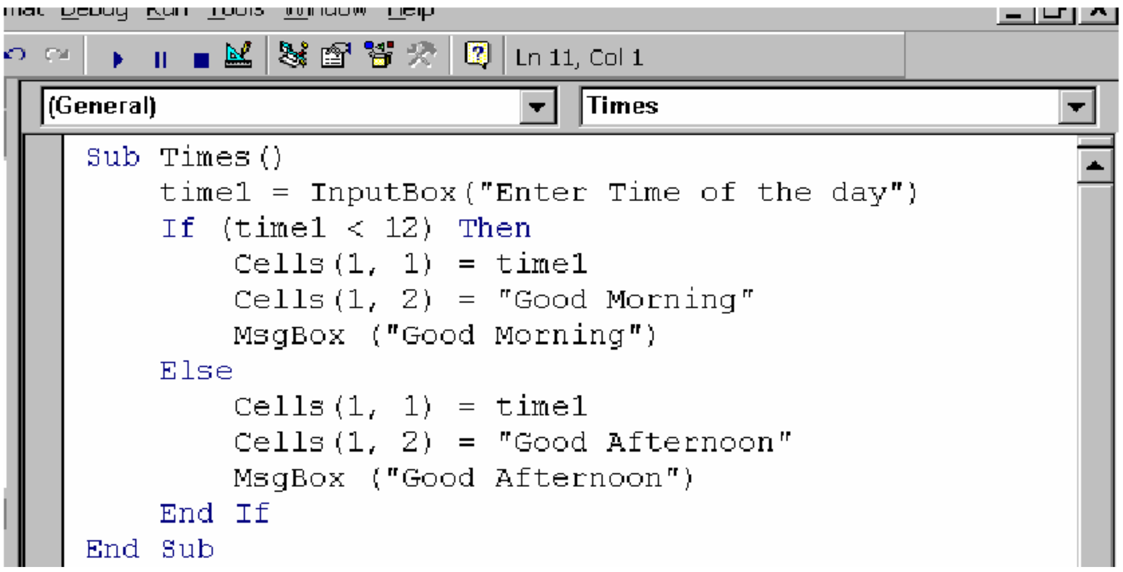
## ***<u>STEP (7) The If – Then structure</u>***

. Often we may only want to follow some instructions under a given condition. Consider:

## *If – Then - Else:*

```
If  (T > 1) Then

        Do this if " T > 1"
        Do this also

Else

        Do this if " T < 1"
        And this

End If
```

***Example:***

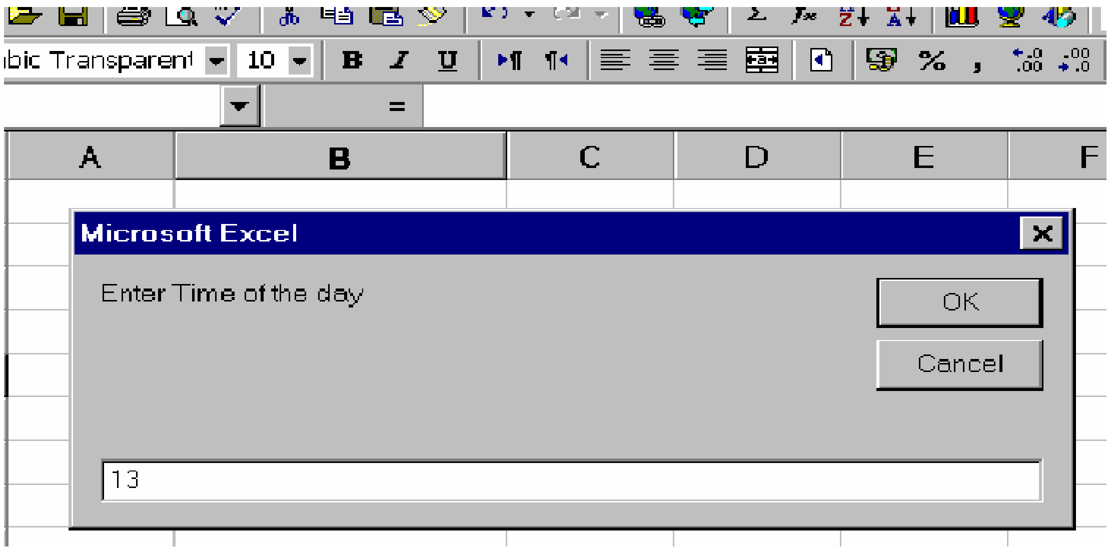


Ln 11, Col 1

(General) | Times

```
Sub Times()
    time1 = InputBox("Enter Time of the day")
    If (time1 < 12) Then
        Cells(1, 1) = time1
        Cells(1, 2) = "Good Morning"
        MsgBox ("Good Morning")
    Else
        Cells(1, 1) = time1
        Cells(1, 2) = "Good Afternoon"
        MsgBox ("Good Afternoon")
    End If
End Sub
```

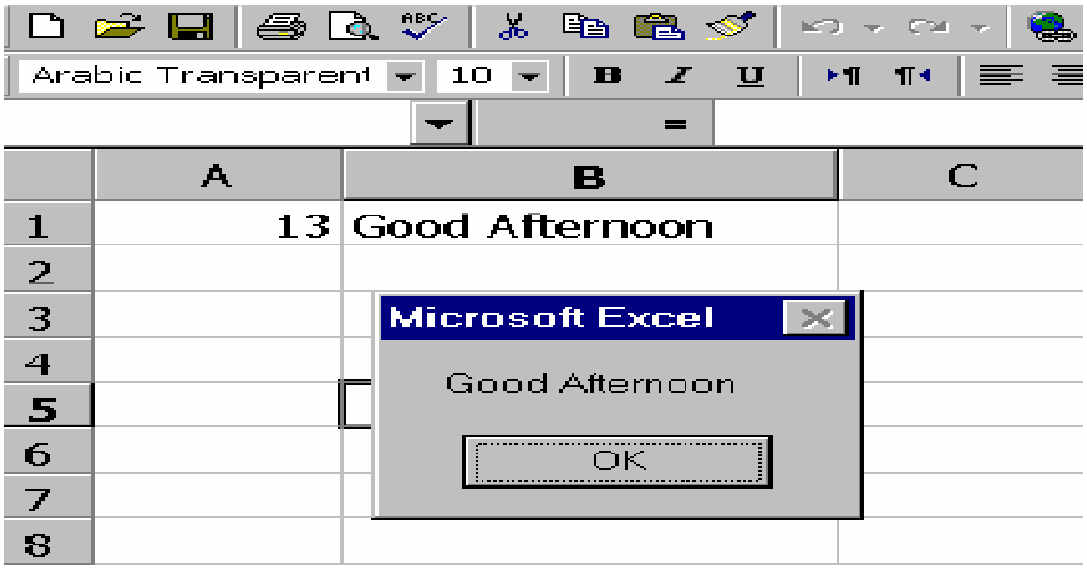
When the program runs, the figure below will appear asking for the time of the day;


Microsoft Excel

Enter Time of the day

13

Once the time is entered and you click OK, the program will put the value of "13" in cell "A1", the text "Good Afternoon" in cell "B1", and then displays a message box as shown below.


|  | A | B | C |
|---|---|---|---|
| 1 | 13 | Good Afternoon |  |

Microsoft Excel

Good Afternoon

**<u>Note:</u>**

. We used the operator < (less than) and > (bigger than) before.

. Other operator that may be used in the if statements are:

= (equal to) and <> (not equal to)

***Using ElseIf:***

. If the first condition in the if statement turns out to be false, you might want to explore other alternatives.

. The ElseIf allows you to add as many alternatives as you like.

```
Sub Times()

        time1 = InputBox("Enter Time of the day")
        If (time1 < 12) Then
               Cells(1, 1) = time1
               Cells(1, 2) = "Good Morning"
               MsgBox ("Good Morning")

        ElseIf (time1 >= 12 and time1 < 18) Then
               Cells(1, 1) = time1
               Cells(1, 2) = "Good Afternoon"
               MsgBox ("Good Afternoon")

        Else
                Cells(1, 1) = time1
               Cells(1, 2) = "Good Afternoon"
               MsgBox ("Good Evening")

        End If
End Sub
```

## <u>VBA FUNCTIONS AND ARRAYS</u>

### Subroutines versus Functions

. The VBA code that you write in Visual Basic module is known as a *procedure*.

. You can write two types of procedures:

**<u>A subroutine procedure:</u>** A group of VBA statements that perform actions with Excel, i.e. like the one we used in trapezoidal rule.

**<u>A function procedure:</u>** A group of VBA statements that perform a calculation and return a *single* value.

### <u>Common Types of Functions in Excel</u>

***Built-in Functions:***

. Excel includes many worksheet *functions* that we often use.

. Examples include SUM, AVERAGE, MEAN, SIN, … etc.

. Each function takes one or more arguments and returns a single value.

***VBA or User-defined Functions:***

. Similar to what a build-in function can do but can do more.

. Simply, it is a piece of programming code that gives you a value back whenever you use it.

**Looking at Functions**

. Every function must start with the keyword *Function* and end with an *End Function* statement.

```
Function Multiply (x , y)
        Statements
        Multiply = x * y
End Function
```

. This function, named "*Multiply*", takes two arguments (named *x* and *y*).

. *Arguments* are a list of variables representing values that are passed to the function.

. Functions can have no argument, one argument, or multiple arguments.

. *Statements* are various line of code, one of which will be of the form "*Multiply = expression", i.e. Multiply = x*y.*

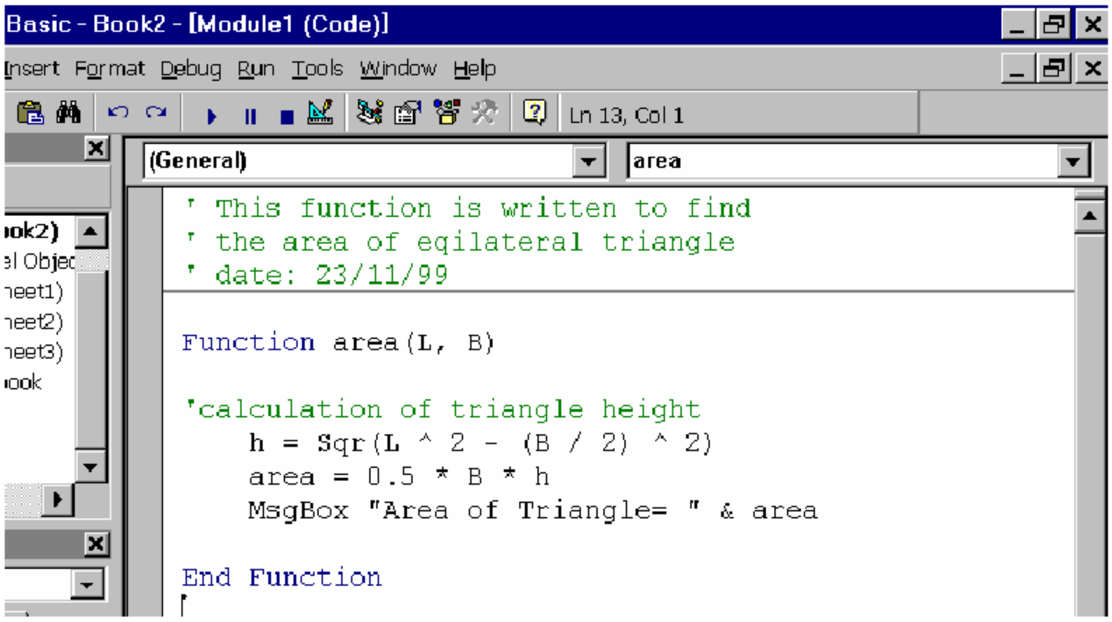
. This is *the same name as on the first line* and the value of the expression is assigned to it. This is becomes the return value of the function

. When you run the function, it returns a single value that is the value of 'x' multiplied by '*y*'.

You can’t use the Excel macro recorder to record a function. You must manually enter every function that you create.

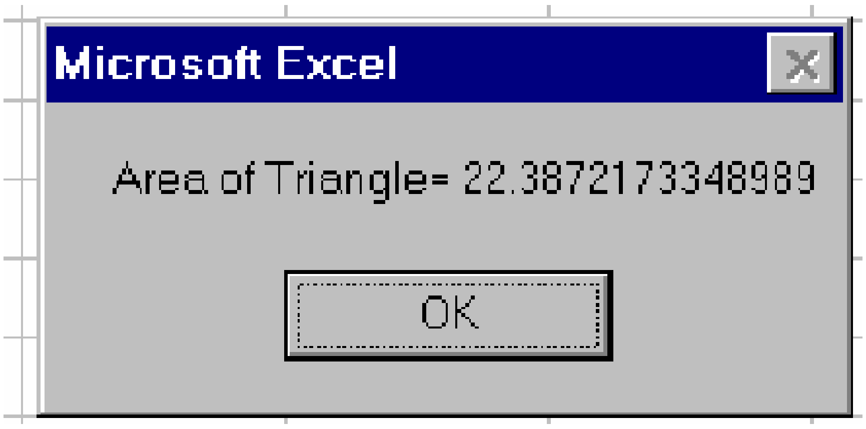
***Example:***

Create a function that determines the area of equilateral triangle with side length ‘L’ and base ‘B’.


Basic - Book2 - [Module1 (Code)]
Insert Format Debug Run Tools Window Help
Ln 13, Col 1
(General)
area

```
' This function is written to find
' the area of eqilateral triangle
' date: 23/11/99

Function area(L, B)

'calculation of triangle height
    h = Sqr(L ^ 2 - (B / 2) ^ 2)
    area = 0.5 * B * h
    MsgBox "Area of Triangle= " & area

End Function
```

. When you run the function, it returns a value of the area and the following window will display the value of the area.



```
Area of Triangle= 22.3872173348989
```

**Executing Functions**

. Functions can be executed in only two ways:

By calling the function from another subroutine or function.

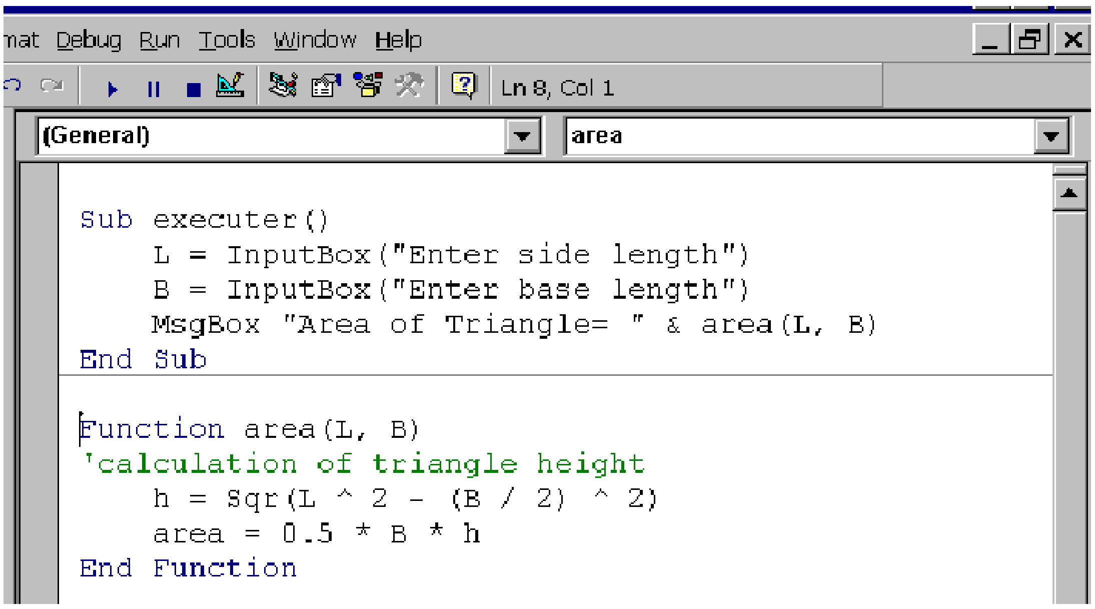
By using the function directly in a worksheet formula.

**Method 1: Calling the function from a subroutine**

Because you can't execute the function directly, you must call it from another subroutine as shown in the following figure

I. When running the executer subroutine, two consecutive windows will appear asking about L then B.

II. Once you gave the data needed, a message box will appear showing the resulting area as shown in the previous example.



```
Sub executer()
    L = InputBox("Enter side length")
    B = InputBox("Enter base length")
    MsgBox "Area of Triangle= " & area(L, B)
End Sub

Function area(L, B)
'calculation of triangle height
    h = Sqr(L ^ 2 - (B / 2) ^ 2)
    area = 0.5 * B * h
End Function
```

**<u>Method 2. Calling the function from a worksheet formula</u>**

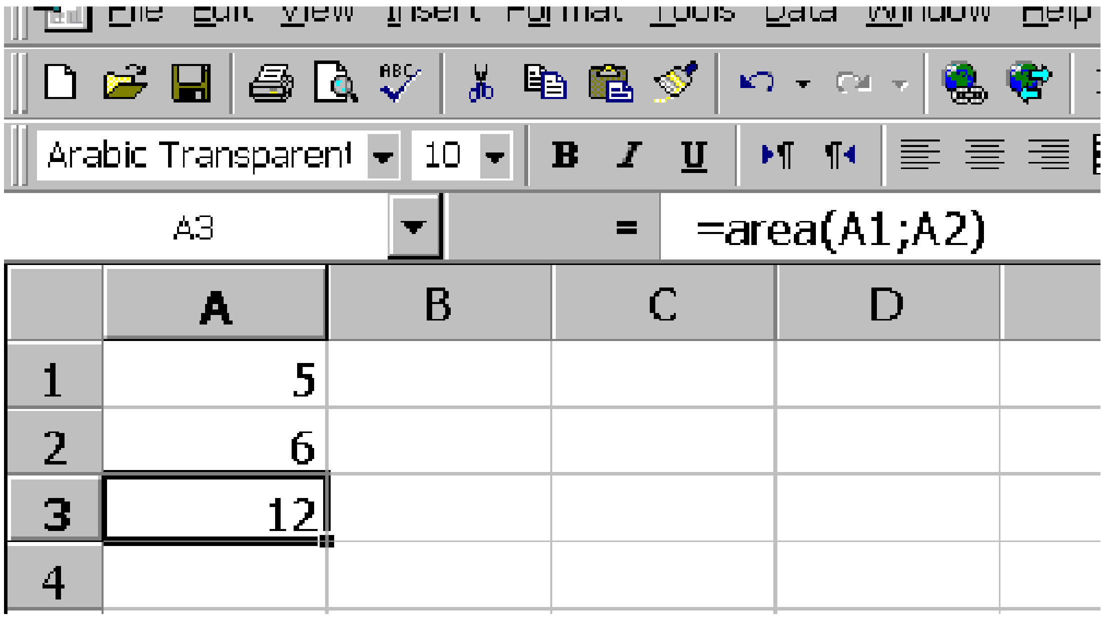
. Activate the worksheet in the same workbook that holds the "*area*" function.

• Then, enter the following formula into any cell:

= area (5,6)

• The cell displays 12, which is the area of equilateral triangle of side length = 5 and base length = 6.

• You can use a cell reference as the argument for the area function as we can see below;



A3 = =area(A1;A2)

| | A | B | C | D |
|---|---|---|---|---|
| 1 | 5 | | | |
| 2 | 6 | | | |
| 3 | 12 | | | |
| 4 | | | | |

**<u>ARRAYS</u>**

. An *array* is a group of variables that have a common name.

. You refer to specific variable in the array by using the array *name* and an *index* number.

**Array example:**

. Assume that you want to store the months of the year in your subroutine.

. You may define an array of 12 string variables to hold the months of the year.

. If the array name is *MonthNames*, you can refer to the first element of the array (i.e. January) as *MonthNames (1)*, the second element as *MonthNames (2)*, and so on.

**DeclaringArrays**

. You declare an array with a Dim statement, just like you declare regular variable.

. However, you also need to specify the number of elements in the array.

. You do this by specifying the first index number, the keyword to, and the last index number as shown in the following:

```
Dim MyArray (1 to 100) as Integer
```

. When you declare an array, you can specify only the upper index.

. VBA assumes that 0 is the lower index.

. Therefore, the following statements both declare the same array:

```
    Dim MyArray (0 to 100) as Integer
    Dim MyArray (100) as Integer
```

**Note:**

. If you want VBA to assume that 1 is the lower index for the arrays include the following statement before any sub or Function in your module

```
Option Base 1
```

. This statement forces VBA to use 1 as the first index number.

. If this statement is present, the following statements are identical:

```
Dim MyArray (1 to 100) as Integer
Dim MyArray (100) as Integer
```

**Declaring Multi-dimensional Arrays**

. The arrays created above are all one-dimensional arrays.

. Arrays you create in VBA can have as many as 60 dimensions.

. The following example declares a 100-integer array with two dimensions:

```
Dim MyArray (1 to 10, 1 to 10) as Integer
```

. You can think of this array as a 10 by 10 matrix.

. To refer to a specific element in this array, you need to specify two index numbers.

. Again, if you are thinking of the array in terms of a 10 by 10 matrix, the following example shows how you can assign a value to an element located in the *third row* and *fourth column* of the array:

```
MyArray (3, 4) = 125
```

***Dynamic arrays:***

. It can be hard to know ahead of time exactly how many items you are going to put onto an array.

. When you solve a continuous beam, for example, you don't know how many spans you are going to use. It could be 2 or even 10.

. So, a *dynamic array* is simply an array whose size is not predetermined. To use it, you specify a size, and then it acts just like any other array.

***Declaring Dynamic arrays:***

. A dynamic array is declared just like any other kind of array: with the Dim statement.

. You declare it with a blank set of parentheses:

```
Dim MyArray ( ) as Integer
```

. Before you can use this array, you must use the *ReDim* statement to tell VBA how many elements the array has.

. You can use the *ReDim* statement any number of times, changing the array's size as often as you need.

```
ReDim MyArray (1 to 10)
```

Warning

. When you redimension an array by using ReDim you delete the values stored in the array elements.

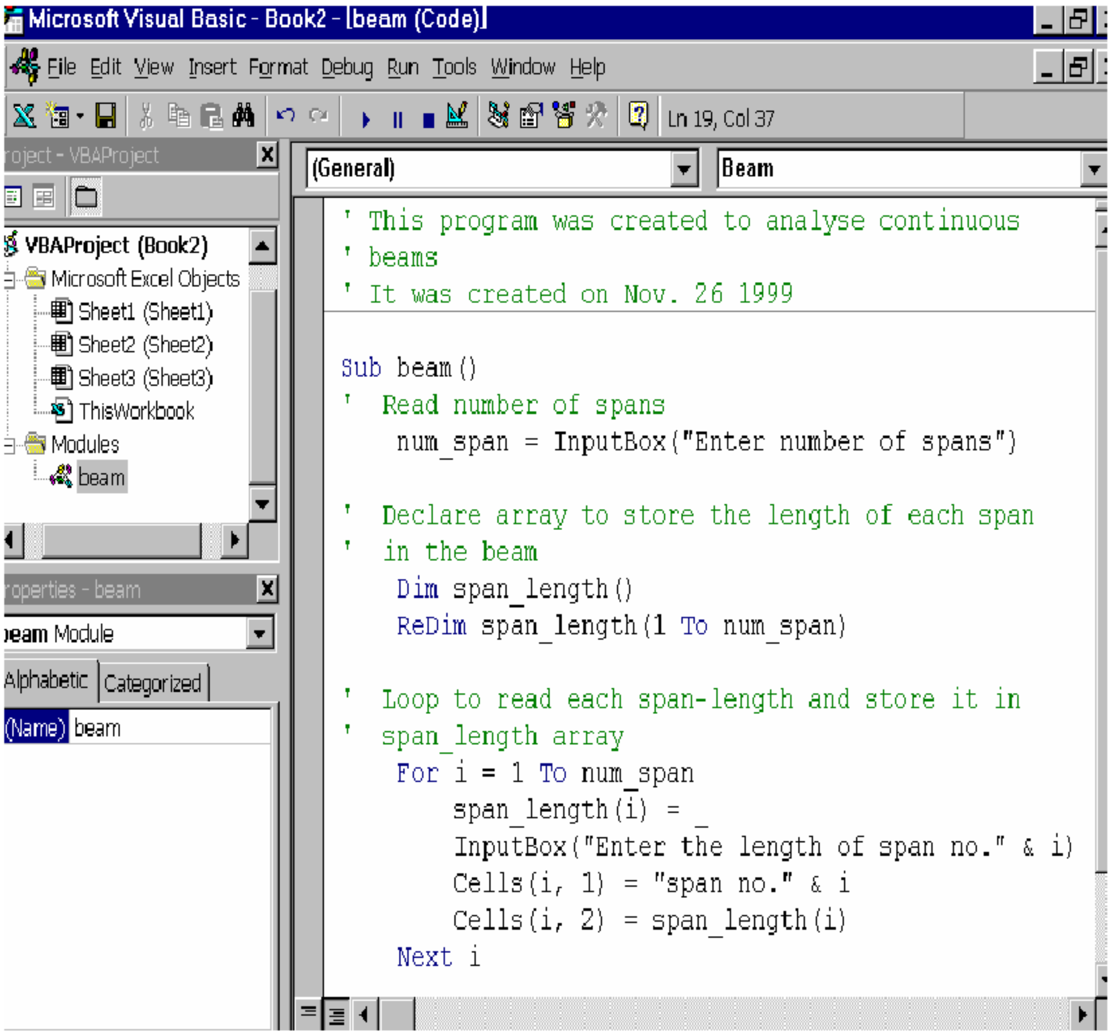
. You can avoid this by using the *Preserve* keyword.

```
        ReDim Preserve MyArray (1 to 20)
```

. If this array has 10 elements stored on it and you execute the preceding statement, the first 10 elements will remain unchanged, and the array will have room to store 10 more elements.

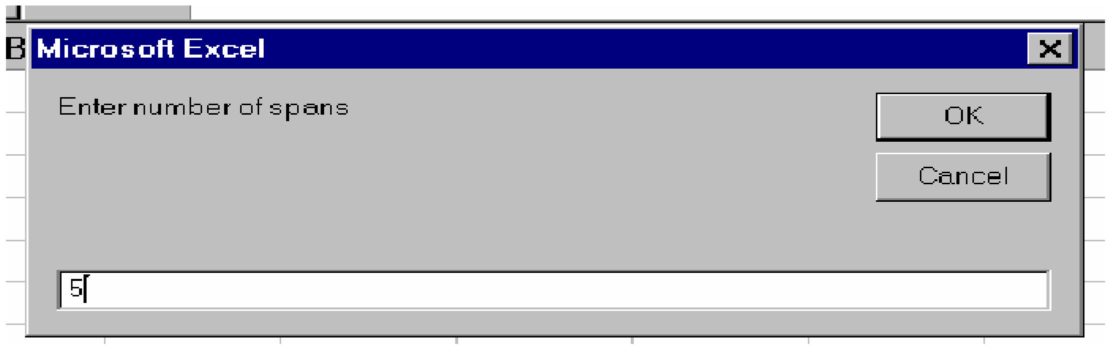
***Example:***

For a part of continuous beam analysis with any number of spans, create a Sub to read and store the length of each span



```
' This program was created to analyse continuous
' beams
' It was created on Nov. 26 1999

Sub beam()
'  Read number of spans
    num_span = InputBox("Enter number of spans")

'  Declare array to store the length of each span
'  in the beam
    Dim span_length()
    ReDim span_length(1 To num_span)

'  Loop to read each span-length and store it in
'  span_length array
    For i = 1 To num_span
        span_length(i) = _
        InputBox("Enter the length of span no." & i)
        Cells(i, 1) = "span no." & i
        Cells(i, 2) = span_length(i)
    Next i
```

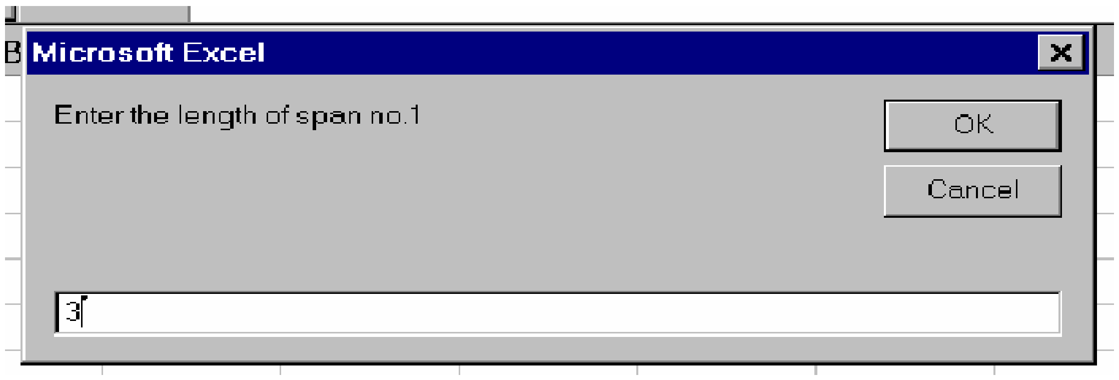
After running this sub the program will ask about the number of spans, as shown in the following figure:



Enter number of spans

5

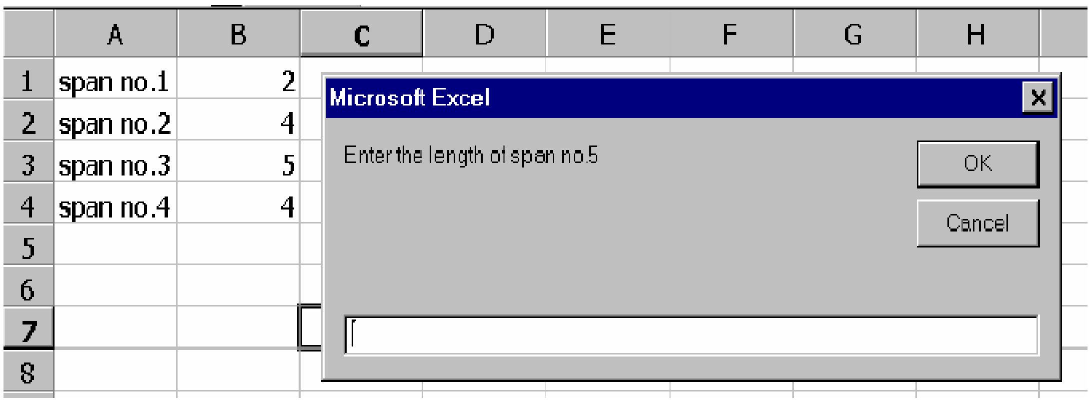
After entering the number of spans and clicking OK button, five consecutive windows will appear to ask about the length of each span. The first window, out of five, is shown below:



Enter the length of span no.1

3

By entering the last span length the sub should put these values in the work sheet, as shown below, and also stores these value in the span_length array



|  | A | B |
|---|---|---|
| 1 | span no.1 | 2 |
| 2 | span no.2 | 4 |
| 3 | span no.3 | 5 |
| 4 | span no.4 | 4 |

Enter the length of span no.5