Functions and Sub

Created by:	Marek Prazak	Reviewed by:	Marek Prazak	Create Date:	29 March 2019

Contents

1. PURPOSE

- This lesson will introduce you to VBA functions and Subs and highlight the difference between a **function** and a **sub** in **Excel VBA**.
- The lesson should take approximately 1 hour to finish.
- As the lessons are simplified to allow you to grasp the basics of VBA functionality, do not despair if you won't understand all on your first walkthrough, we highly encourage you to use additional sources provided by us or to go online and research more in depth. The most important is to try out all examples highlighted in this lesson as that is the best way to understand the logic.

2. DEFINITION

- Sub and VBA functions are procedures that contain the main body of the macros code.
- The difference between a function and a sub in Excel VBA is that a function can return a value while a sub cannot. Functions and Subs become very useful as program size increases.
- An array is a group of variables. In Excel VBA, you can refer to a specific variable (element) of an array by using the array name and the index number. As you'll get more familiar with the coding you will encounter cases where referencing variables one by one takes too much time and slows your code instead you reference whole "array" of variables.

3. OVERVIEW

3.1. SUB

- If you want Excel VBA to perform some actions, you can use a sub.
- As you've seen in previous lessons Sub will hold the VBA code that works as instruction for your macro.
- Sub and Functions can hold arguments, which function as an input. This can be best visible in moment when you reference Call/Reference sub and you want to give it values/arguments to work with.

    - Place a sub into a module (In the Visual Basic Editor, click Insert, Module).

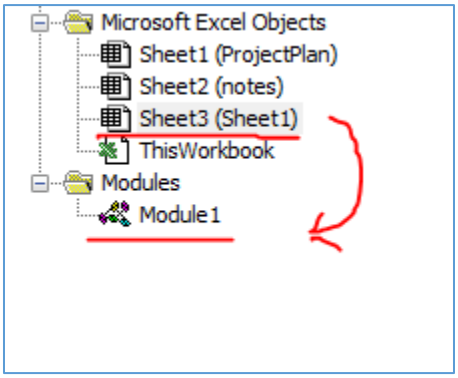
```
Sub Area(x As Double, y As Double)

    MsgBox x * y

End Sub
```

- Sub can contain any number of arguments with, different variable types. In this case Sub has two arguments (of type Double). **Sub does not have a return type!** You can refer to this sub by using **Call** from somewhere else in your code. To achieve this simply use the name of the sub and give it a value for each argument,

- Place a command button on your worksheet and add the following code line.

```
Area 3, 5
```

- Result when you click the command button on the sheet, it will reference the Sub in Module.



- You can use same logic in Modules only without use of Sheets.
- When using Arguments always make sure that you're using same type of variable.

```
Sub MainCode()

    Dim firstNum As Double, secondNum As Double

    firstNum = InputBox("Give me first num")
    secondNum = InputBox("Give me first num")


    Call area(firstNum, secondNum)

End Sub
```

```
Sub area(x As Double, y As Double)

    MsgBox x * y

End Sub
```

- This is very useful for **shortening your code**, if part of your **code is used multiple times** with only variables changing, instead of writing the whole section multiple times, you can **write** it **once** as a separate sub and "call it". Imagine having a scenario where you use IF/Else to decide whether to delete row of employees details or continue with your calculation, throughout your code you will have to make this decision multiple times and so instead of Writing 200 rows of code that is repeating itself you will write only 10 that will be called upon multiple times.

3.2. FUNCTION

- If you want Excel VBA to perform a task that returns a result (**has return value**), you can use a function.
    - Place a function into a module (In the Visual Basic Editor, click Insert, Module).

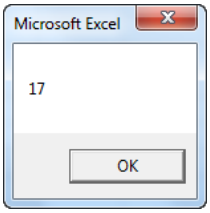
```
Function Area(x As Double, y As Double) As Double

    Area = x * y

End Function
```

    - This function has two arguments (of type Double) and a return type (the part after **As** also of type Double). You can use the name of the function (Area) in your code to indicate which result you want to return (here x * y).
    - You can now refer to this function (in other words **call the function**) from somewhere else in your code by simply using the name of the function and giving a value for each argument.

- Place a command button on your worksheet and add the following code lines to the button:

```
Private Sub CommandButton1_Click()

    Dim z As Double

    z = Area(3, 5) + 2

    MsgBox z


End Sub
```

- The function returns a value so you have to 'catch' this value in your code. You can use another variable (z) for this. Next, you can add another value to this variable (if you want). Finally, display the value using a MsgBox.

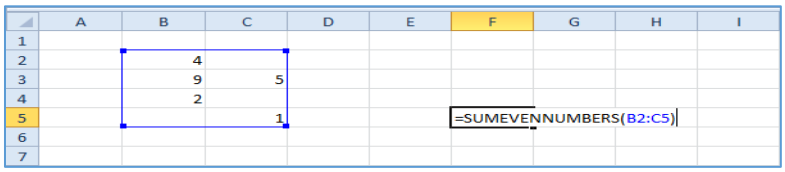


Conclusion*: The function returned the value 15. We added the value 2 to this result and displayed the final result. When we called the sub we had no more control over the result (15) because a sub cannot return a value!*

3.2.1. User Defined Function

- Excel has a large collection of pre-defined functions (vloopup, sum, round, etc.). In most situations those functions are sufficient to get the job done. If not, you can **create your own function** called **User Defined Function** or **custom Excel function**. You can access a User Defined Function just like any other Excel function by writing equal sign and name of function.

  - In this example we want to create a function called **SUMEVENNUMBERS** that sums up the even numbers of a randomly selected range.



  - When creating User defined functions, you have to **place them into a module**.
  - Open the Visual Basic Editor and click Insert, Module.
  - Add the following code line.

```
Function SUMEVENNUMBERS(rng As Range)


End Function
```

- The name of our Function is **SUMEVENNUMBERS**. The part between the brackets means we give Excel VBA a range as input. We name our range rng, but you can use any name.
- Next, we declare a Range object and call it cell.

```
Dim cell As Range
```

- We want to check each cell in a randomly selected range (this range can be of any size). In Excel VBA, you can use the **For Each Next** loop for this. Add the following code lines.

```
For Each cell In rng

Next cell
```

Note: *cell is randomly chosen; you can use any name.*

- Next, we check for each value in this range whether it is even or not. We use the **Mod** operator for this. The **Mod** operator gives the remainder of a division. So **7 mod 2** would give **1**. **7** is **divided** by **2** (3 times) to give a remainder of 1. Having said this, it is easy to check whether a number is even or not. Only if the remainder of a number divided by 2 is 0, the number is even. **8 mod 2** gives 0, 8 is divided by 2 exactly 4 times, and therefore 8 is even.

7 mod 2 = 1	7/2 = 3 -> 1 remains
8 mod 2 = 0	8/2 = 4 -> 0 remains

- Add the following If statement to the **For Each Next** loop

```
If cell.Value Mod 2 = 0 Then

End If
```

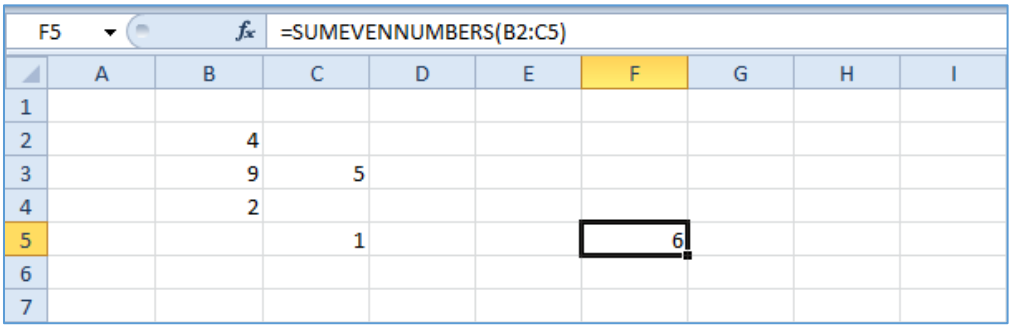
- Only if this statement is true, we add the value to **SUMEVENNUMBERS**. Add the following code line in the **If** statement.

```
SUMEVENNUMBERS = SUMEVENNUMBERS + cell.Value
```

- Don't forget to end the function (outside the loop).

```
Function SUMEVENNUMBERS(rng As Range)
    Dim cell As Range
    For Each cell In rng
        If cell.Value Mod 2 = 0 Then
            SUMEVENNUMBERS = SUMEVENNUMBERS + cell.Value
        End If
    Next cell

End Function
```

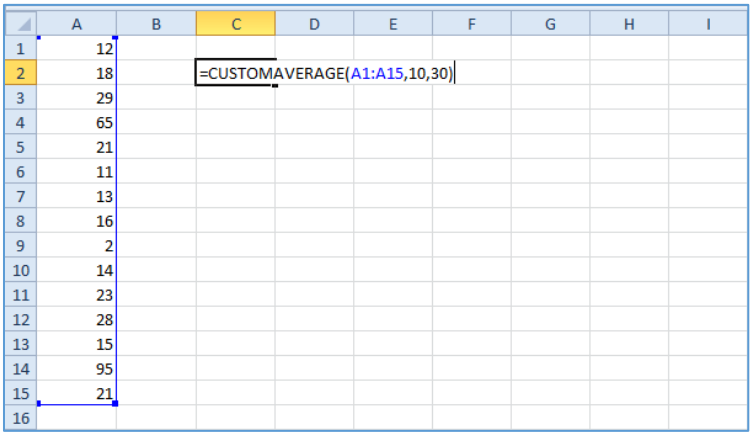
- Now you can use this function, just like any other Excel function, to find the sum of the even numbers of a randomly selected range.



Note: *this function is only available in workbook that it was created in.*

3.2.2. Custom Average Function

- **Example** of User Defined Function that calculates the average of a randomly selected range excluding one or more values that are outliers and shouldn't be averaged.



	A	B	C	D	E	F	G	H	I
1	12								
2	18		=CUSTOMAVERAGE(A1:A15,10,30)						
3	29								
4	65								
5	21								
6	11								
7	13								
8	16								
9	2								
10	14								
11	23								
12	28								
13	15								
14	95								
15	21								
16									

- User defined functions need to be placed into a module
- Open the Visual Basic Editor and click Insert, Module
- Add the following code line.

```
Function CUSTOMAVERAGE(rng As Range, lower As Integer, upper As Integer)

End Function
```

- The name of our Function is **CUSTOMAVERAGE**. The part between the brackets means we give Excel VBA a range and two Integer variables as an input. We name our range rng, one Integer variable we call lower, and one Integer variable we call upper, but you can use any names.
- Declare a Range object and two variables of type Integer. We call the Range object cell. One Integer variable we call total and one Integer variable we call count.

```
Dim cell As Range, total As Integer, count As Integer
```

- We want to check each cell in a randomly selected range (this range can be of any size). In Excel VBA, you can use the **For Each Next** loop for this. Add the following code lines.

```
For Each cell In rng

Next cell
```

- **rng** and **cell** are randomly chosen here, you can use any names. Remember to refer to these names in the rest of your code.
- Check for each value in this range if it falls between the two values (lower and upper). If true, we add the value of the cell to the total and we add 1 to count. This will constantly change the value each time it runs through the For/Next.
- Add the following code lines to the loop.

```
If cell.Value >= lower And cell.Value <= upper Then

    total = total + cell.Value
    count = count + 1

End If
```

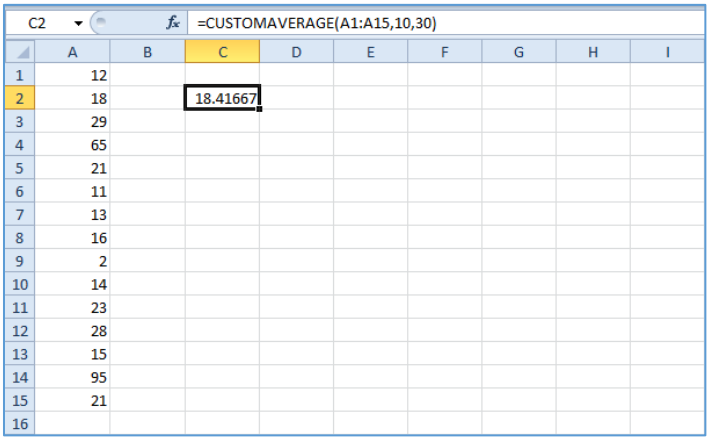
- To return the result of this function (the desired average), add the following code line outside the loop.

```
CUSTOMAVERAGE = total / count
```

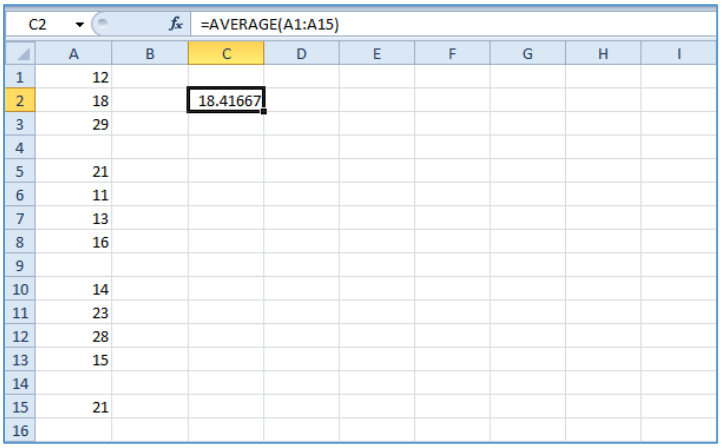
- Don't forget to end the function.

```
Function CUSTOMAVERAGE(rng As Range, lower As Integer, upper As Integer)

    Dim cell As Range, total As Integer, count As Integer

    For Each cell In rng

        If cell.Value >= lower And cell.Value <= upper Then

            total = total + cell.Value
            count = count + 1

        End If

    Next cell

    CUSTOMAVERAGE = total / count

End Function
```

- Now you can use this function just like any other Excel function to calculate the average of numbers that fall between two values.



- As a check, you can delete all values that are lower than 10 and higher than 30 and use the standard Average function in Excel to see if Excel calculates the same average as our custom average function.



Note: *this function is only available in the workbook that was created in.*

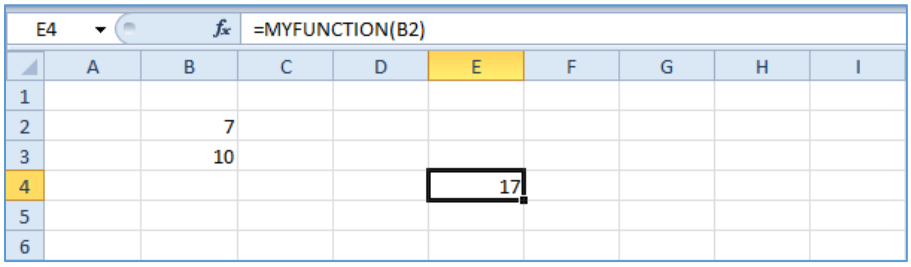
3.2.3. VOLATILE FUNCTIONS

- By default, UDF's (User Defined Functions) in Excel VBA are not volatile. They are only recalculated when any of the function's arguments change. A **volatile function** will be **recalculated** whenever **calculation occurs** in any cells on the worksheet. Let's take a look at an easy example to explain this a bit more.

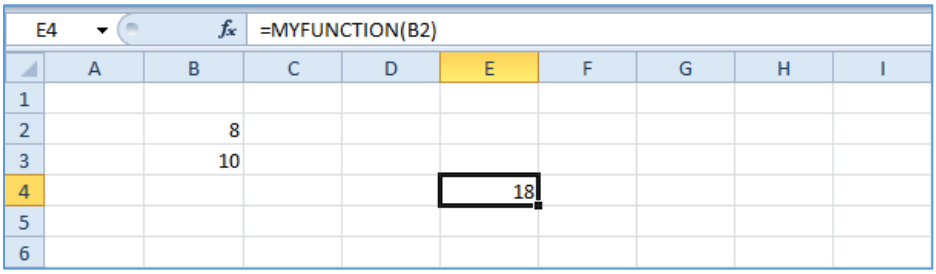
  - Open the Visual Basic Editor and click Insert, Module.
  - Create a function called **MYFUNCTION** which returns the sum of the selected cell and the cell below this cell.
  - Add the following code lines.

```
Function MYFUNCTION(cell As Range)

    MYFUNCTION = cell.Value + cell.Offset(1, 0).Value

End Function
```

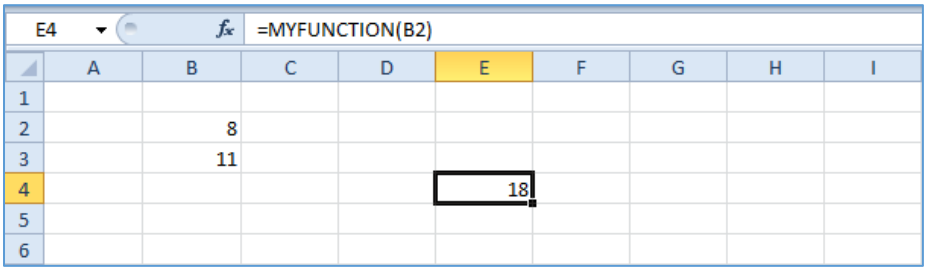
- Now you can use this function, just like any other Excel function.



- This is a non-volatile function. Non-volatile functions are only recalculated when any of the function's arguments change. Change the value of cell B2 to 8.



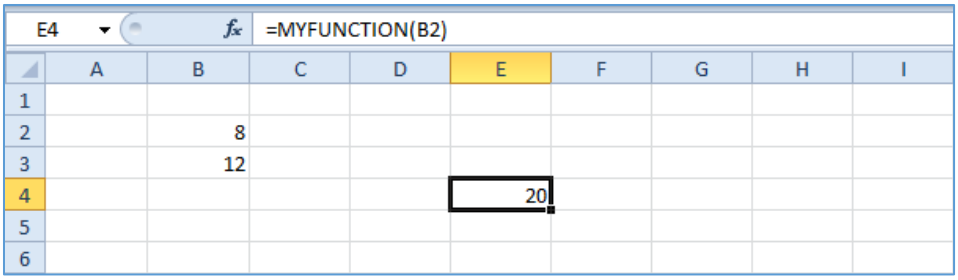
- Now change the value of cell B3 to 11.



- The non-volatile function is not recalculated when any other cell on the sheet changes
- Update the function as follows to make the function volatile.

```
Function MYFUNCTION(cell As Range)

    Application.Volatile

    MYFUNCTION = cell.Value + cell.Offset(1, 0).Value

End Function
```

- Change the value of cell B3 to 12.



Note: *You need to enter the function again to make it volatile (or refresh it by placing your cursor in the formula bar and pressing enter).*

3.2.4. ByRef and ByVal

- You can pass arguments to a procedure (function or sub) by reference or by value. By default, Excel VBA passes arguments by reference. As always, we will use an easy example to make things more clear.

  - Place a command button on your worksheet and add the following code lines.

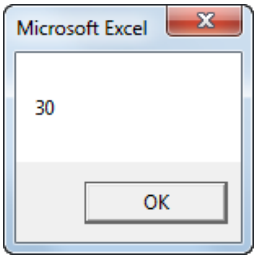
```
Private Sub CommandButton1_Click()

    Dim x As Integer
    x = 10

    MsgBox Triple(x)
    MsgBox x


End Sub
```

  - The code calls the function **Triple**. It's the result of the second MsgBox we are interested in. Functions need to be placed into a module.

- Open the Visual Basic Editor and click Insert, Module.
- Add the following code lines.

```
Function Triple(ByRef x As Integer) As Integer

    x = x * 3
    Triple = x

End Function
```

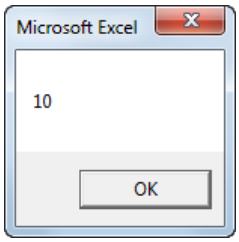
- Result when you click the command button on the sheet.



- Replace ByRef with ByVal;.

```
Function Triple(ByVal x As Integer) As Integer

    x = x * 3
    Triple = x

End Function
```

- **Result** when you click the command button on the sheet.



- When passing arguments by reference we are referencing the original value. The value of x (the original value) is changed in the function. As a result the second MsgBox displays a value of 30. When passing arguments by value we are passing a copy to the function. The original value is not changed. As a result the second MsgBox displays a value of 10 (the original value).

3.3. ARRAY

- An Array can be either One-dimensional Array or Two-dimensional Array
- Array is similar to range with the difference that each value can be directly referenced.

3.3.1. ONE-DIMENSIONAL ARRAY

- Place a command button on your worksheet and add the following code lines.

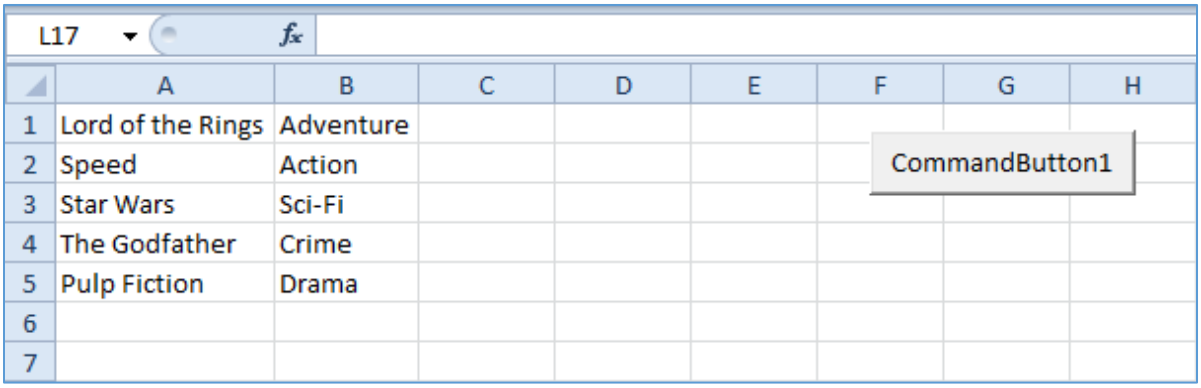
```
Private Sub CommandButton1_Click()

    Dim Films(1 To 5) As String

    Films(1) = "Lord of the Rings"
    Films(2) = "Speed"
    Films(3) = "Star Wars"
    Films(4) = "The Godfather"
    Films(5) = "Pulp Fiction"

    MsgBox Films(4)


End Sub
```

- When you click the command button on the sheet.

- The first code line declares a String array with name Films. The array consists of five elements.
- Next, we initialize each element of the array. Finally, we display the fourth element using a MsgBox.

3.3.2. TWO-DIMENSIONAL ARRAY

- This time we are going to read the names from the sheet.



- Place a command button on your worksheet and add the following code lines.

```
Private Sub CommandButton1_Click()

    Dim Films(1 To 5, 1 To 2) As String
    Dim i As Integer, j As Integer
    For i = 1 To 5
        For j = 1 To 2
            Films(i, j) = Cells(i, j).Value
        Next j
    Next i
    MsgBox Films(4, 2)


End Sub
```

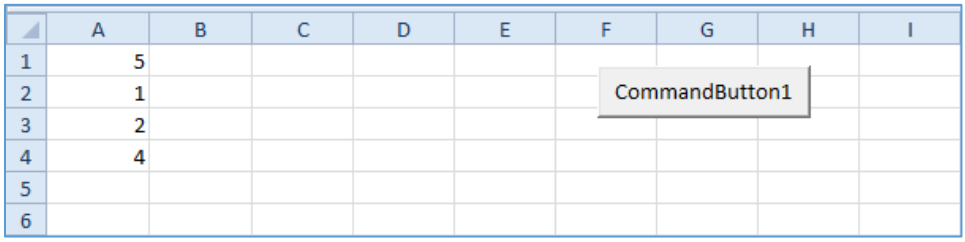
- When you click the command button on the sheet.

- The first code line declares a String array with name Films. The array has two dimensions. It consists of 5 rows and 2 columns. The other two variables of type Integer are used for the Double Loop to initialize each element of the array. Finally, we display the element at the intersection of row 4 and column 2.
- **Tip**: rows go first, then columns.

3.3.3. USE CASES FOR ARRAY

- **Dynamic Array** - If the size of your array increases and you don't want to fix the size of the array, you can use the ReDim keyword. Excel VBA then changes the size of the array automatically.

  - Add some numbers to column A.



- Place a command button on your worksheet and add the following code lines.

```
Private Sub CommandButton1_Click()

    Dim numbers() As Integer, size As Integer, i As Integer

End Sub
```

- the array has no size yet. **numbers**, **size** and **i** are randomly chosen here, you can use any names. Remember to refer to these names in the rest of your code.
- Next, we determine the size of the array and store it into the variable size. You can use the worksheet function CountA for this. Add the following code line.

```
size = WorksheetFunction.CountA(Worksheets(1).Columns(1))
```

- We now know the size of the array and we can redimension it. Add the following code line.

```
ReDim numbers(size)
```

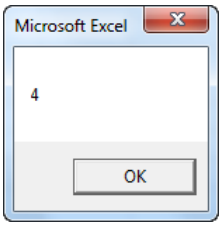
- Next, we initialize each element of the array. We use a loop.

```
For i = 1 To size
    numbers(i) = Cells(i, 1).Value
Next i
```

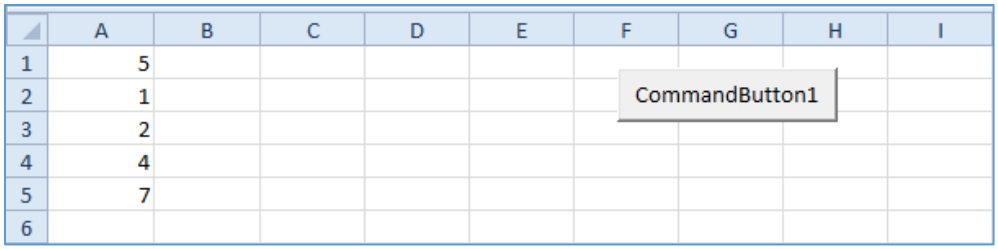
- We display the last element of the array using a MsgBox.

```
Private Sub CommandButton1_Click()
    Dim numbers() As Integer, size As Integer, i As Integer
    size = WorksheetFunction.CountA(Worksheets(1).Columns(1))
    ReDim numbers(size)
    For i = 1 To size
        numbers(i) = Cells(i, 1).Value
    Next i
    MsgBox numbers(size)
End Sub
```

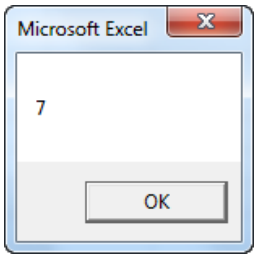
- Exit the Visual Basic Editor and click the command button on the sheet.



- Now to clearly see why this is called a dynamic array, add a number to column A.

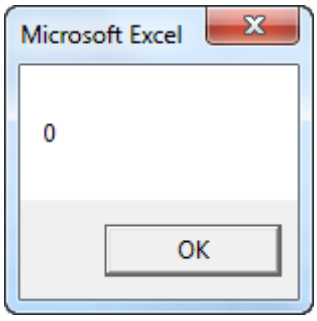


- Click the command button again.



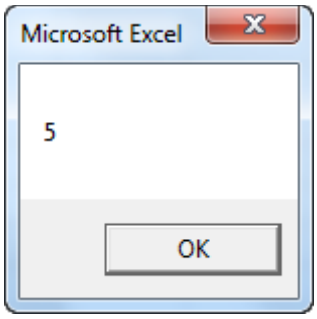
- Excel VBA has automatically changed the size of this dynamic array.
- When you use the ReDim keyword, you erase any existing data currently stored in the array. For example, add the following code lines to the previously created code.

```
ReDim numbers(3)
MsgBox numbers(1)
```



- The array is empty.
- When you want to preserve the data in the existing array when you redimension it, use the Preserve keyword.

```
ReDim Preserve numbers(3)
MsgBox numbers(1)
```



- **Array Function -** The **Array function** in **Excel VBA** can be used to quickly and easily initialize an array. Place a command button on your worksheet and use the following code lines.
  - First, create a variable named departments of type Variant.

```
Dim departments As Variant
```

  - Use the Array Function to assign an array to the variable departments. Add the following code line.

```
departments = Array("Sales", "Production", "Logistics")
```

  - To show the element with index 1, add the following code line.

```
Private Sub CommandButton1_Click()
    Dim departments As Variant
    departments = Array("Sales", "Production", "Logistics")
    MsgBox departments(1)
End Sub
```

- By default, the element's index of the array starts from 0.
- Add Option Base 1 to the General Declarations section if you want the index to start from 1.

```
Option Explicit

Option Base 1

Private Sub CommandButton1_Click()

Dim departments As Variant
departments = Array("Sales", "Production", "Logistics")

MsgBox departments(1)

End Sub
```

- Result when you click the command button again.

- **Month Names** - Below we will look at a program in Excel VBA which creates a User Defined Function that uses the Array function to return the names of the months.
  - User defined functions need to be placed into a module.
  - Open the Visual Basic Editor and click Insert, Module.
  - Add the following code line.

```
Function MONTHNAMES()


End Function
```

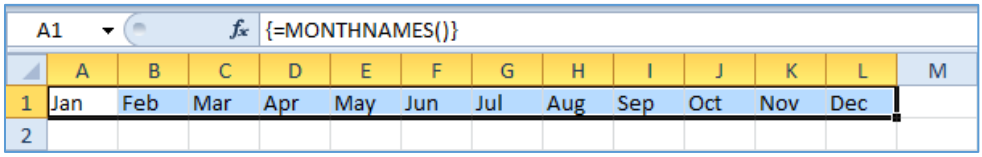
  - The name of our Function is **MONTHNAMES**. The empty part between the brackets means we give Excel VBA nothing as input.
  - The Array function allows us to assign values to a Variant array in one line of code.

```
MONTHNAMES = Array("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")
```

  - Don't forget to end the function.

```
Function MONTHNAMES()
    MONTHNAMES = Array("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")
End Function
```

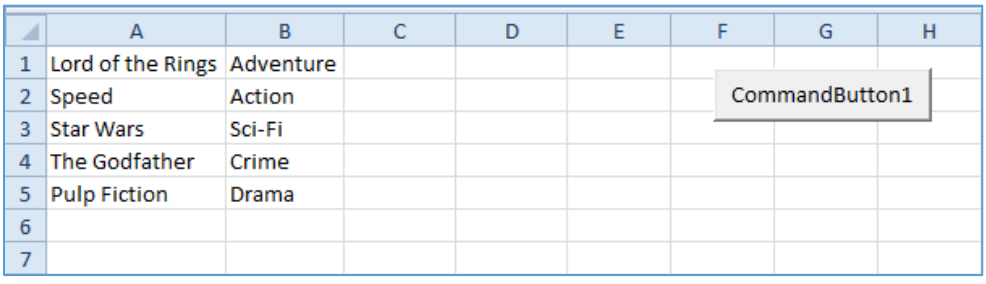
- Now you can use this function, just like any other Excel function, to return the names of the months. Select twelve horizontal cells, enter the function **=MONTHNAMES()** and press CTRL + SHIFT + ENTER.



Note: *You cannot delete a single month. To delete the months, select the range A1:L1 and press Delete. This function is only available in this workbook.*

- **Size of an Array** - To get the **size of an array** in **Excel VBA**, you can use the **UBound** and **LBound** functions
  - Place a command button on your worksheet and add the following code lines.
  - First, we need to declare the array. Our array has two dimensions. It consists of 5 rows and 2 columns. Also declare two variables of type Integer.

```
Private Sub CommandButton1_Click()
    Dim Films(1 To 5, 1 To 2) As String, x As Integer, y As Integer
End Sub
```



- Next, we get the size of the array. Add the following code lines

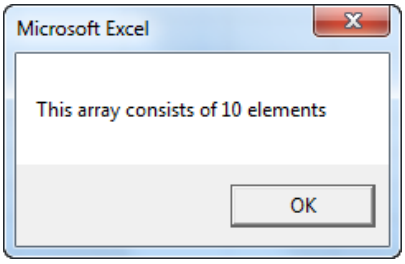
```
x = UBound(Films, 1) - LBound(Films, 1) + 1
y = UBound(Films, 2) - LBound(Films, 2) + 1
```

UBound(Films, 1) gives the upper limit of the first dimension, which is 5.
LBound(Films, 1) gives the lower limit of the first dimension, which is 1.

UBound(Films, 2) gives the upper limit of the second dimension, which is 2.
LBound(Films, 2) gives the lower limit of the second dimension, which is 1.

- As a result, x equals 5 and y equals 2.
- We use a MsgBox to display the number of elements of the array.

```
Private Sub CommandButton1_Click()

    Dim Films(1 To 5, 1 To 2) As String, x As Integer, y As Integer

    x = UBound(Films, 1) - LBound(Films, 1) + 1
    y = UBound(Films, 2) - LBound(Films, 2) + 1

    MsgBox "This array consists of " & x * y & " elements"

End Sub
```



3.4. DISABLE OF NOTIFICATION

- This optimization explicitly **turns off** Excel functionality you don't need to happen (over and over and over) **while your code runs**. Note that in the code sample below we grab the current state of these properties, turn them off, and then restore them at the end of code execution.
- One reason this helps is that if you're updating (via VBA) several different ranges with new values, or copy / pasting from several ranges to create a consolidated table of data, you likely do not want to have Excel taking time and resources to recalculate formulas, display paste progress, or even redraw the grid, especially after every single operation

(even more so if your code uses loops). Just one recalculation and one redraw at the end of your code execution is enough to get the workbook current with all your changes.

- Get current state of various Excel settings; put this at the beginning of your code

```
screenUpdateState = Application.ScreenUpdating
```

  - **We recommend to use mainly screen updating**

```
statusBarState = Application.DisplayStatusBar
```

  - Keep in mind that status bar might be necessary to show progress on power query or if you have custom progress

```
calcState = Application.Calculation
```

  - Responsible for calculation of formulas, if you disable, make sure that you don't need to use the calculation in cells during the macro.

```
eventsState = Application.EnableEvents

displayPageBreakState = ActiveSheet.DisplayPageBreaks 'note this is a sheet-level setting
```

- turn off some Excel functionality so your code runs faster

```
Application.ScreenUpdating = False
Application.DisplayStatusBar = False
Application.Calculation = xlCalculationManual
Application.EnableEvents = False
ActiveSheet.DisplayPageBreaks = False 'note this is a sheet-level setting

>>your code goes here<<
```

- after your code runs, **Do NOT forget to restore state**; put this at the end of your code

```
Application.ScreenUpdating = screenUpdateState
Application.DisplayStatusBar = statusBarState
Application.Calculation = calcState
Application.EnableEvents = eventsState
ActiveSheet.DisplayPageBreaks = displayPageBreaksState 'note this is a sheet-level setting
```

4. CLOSING EXERCISE

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

- Use source excel

1) create formula named =CUSTOMCALCULATION
2) make the formula check that AA has started between January, March and before 2017
3) Check that the AA is Regular
4) Check that his name is 4 and more characters' long
5) Check that the AA is working 40 hours per week
6) Check that his combined bonus salary is more than 5k
7) the formula will have multiple outcomes based on above checks

- "Date is not within scope"
- "AA is not Regular Full Time"
- "Not proper name"
- "Does not work 40h"
- "Not sufficient bonus"
- "OK" - if all conditions met

8) create three subs

- 1st checks each name length - three decision factors (3times possibility to call sub)
    - if the number of AAs who have name longer than 6 char then call second sub
    - if the number of AAs who have name is exactly 5char then call third sub
    - if the number of AAs who have name length is 3,4 char then call second sub
- 2nd sub
    - How many of those names are with OK check and how many are not (pop up message with the details)
- 3rd sub
    - Lvl higher than 4 then highlight the name in red (popup message "done")

9) Create form button to run it. Name it "Start" and assign the correct sub to it.

5. ADDITIONAL KNOWLEDGE

- Functions - https://support.office.com/en-ie/article/create-custom-functions-in-excel-2f06c10b-3622-40d6-a1b2-b6748ae8231f
- Array - https://docs.microsoft.com/en-us/dotnet/visual-basic/programming-guide/language-features/arrays/