MAME and CEUS present,

Excel's VBA for Complete Beginners

Presented by Charles Cossette

Authored by Ken Carney
Extended and Modified (a lot) by Charles C. Cossette

Table of Contents

1 Introduction – What does VBA do?

'VBA' stands for "Visual Basic for Applications." In simple terms, it's a pretty easy programming language that Microsoft added to all their Office products, so that you can write programs that interact its features. This is especially useful in Excel, where you can use VBA to crunch numbers from hundreds of thousands of rows of data, from several different Excel files and beyond. It largely extends Excel's abilities, and certainly becomes a very valuable skill when working with any large company who's got a huge Excel list of anything.

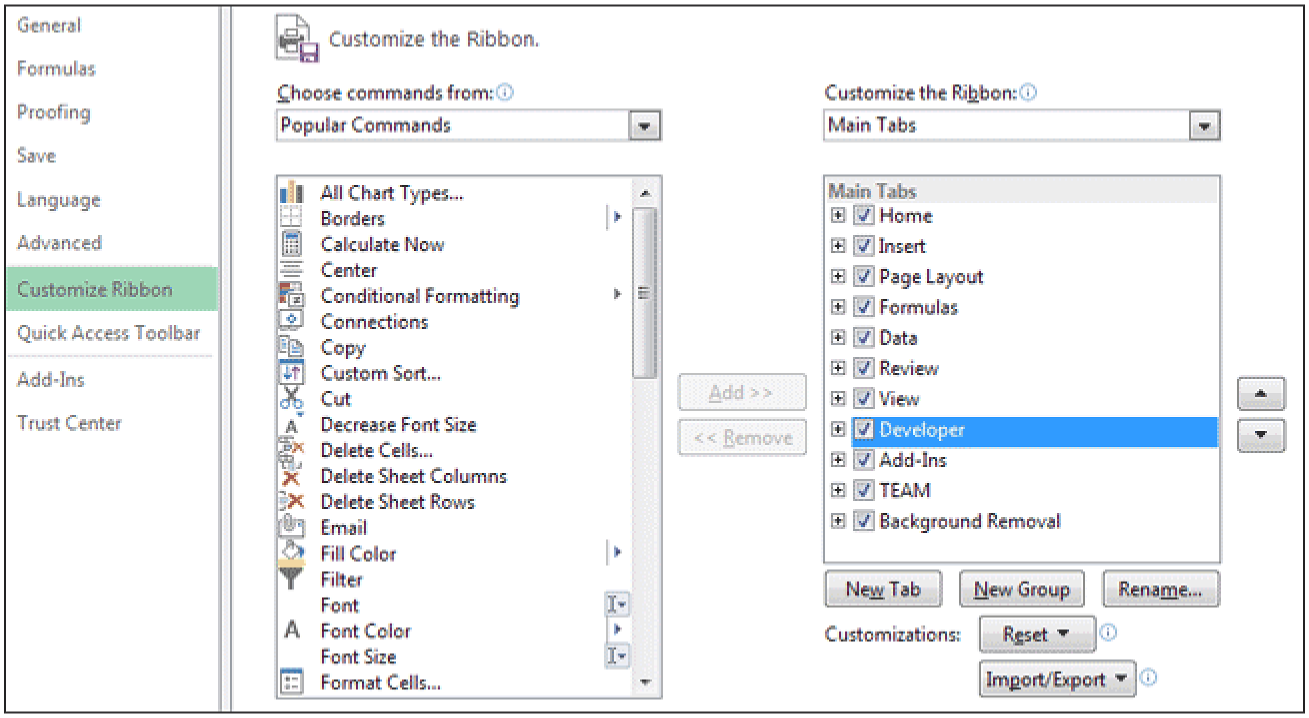
Anyways, you'll see. Now, to the point – this document is meant for people who have an average knowledge of Excel, but know nothing about VBA.

2 Getting Started

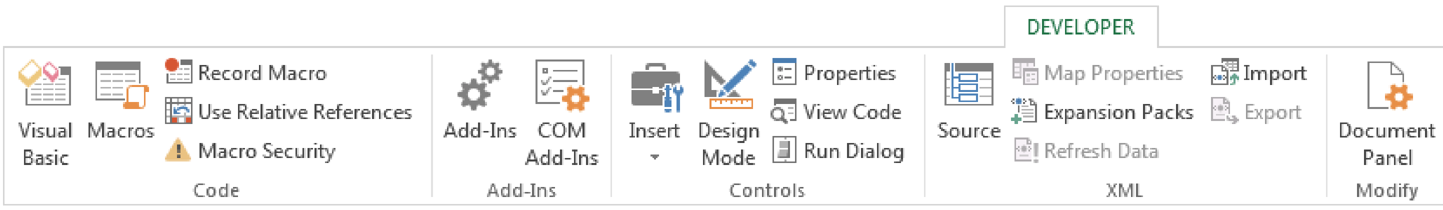
2.1 Adding the 'Developer' Toolbar

Before you can start, you need to add the "Developer" ribbon to the top of Excel.

In Excel 2010 and 2013 click the **File** menu then select **Options**. From the dialogue box, click on **Customize Ribbon** on the left side. From the right hand side you'll then see an area called "Customize the Ribbon". Under "Main Tabs" check the box for **Developer**:



When you have the developer toolbar, you'll see the following tab in the Ribbon (this is from Excel 2013, so you may not have all the items below):



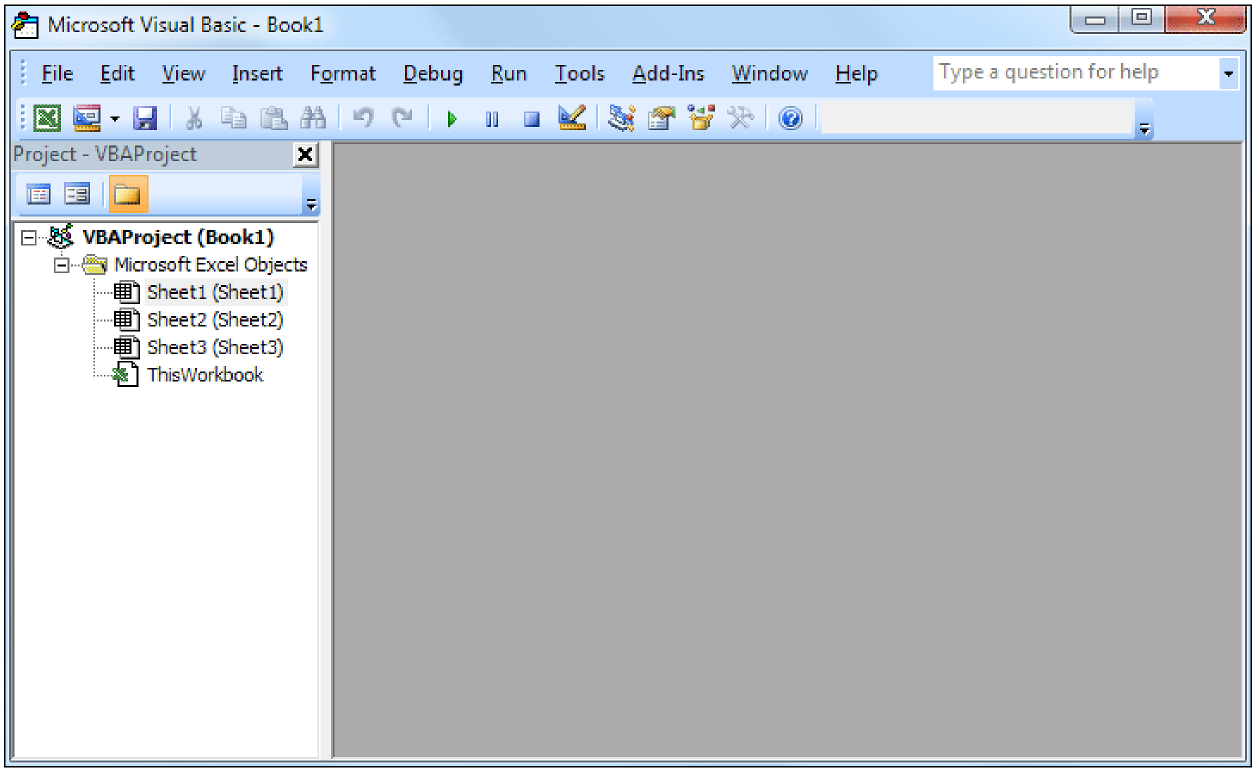
In order to run macros without any annoying security warnings, click on **Macro Security**, on the **Code** panel. Select the option for **Enable all macros**. Then make sure that "Trust access to the VBA object model" is checked:

NOTE: If you're worried about macro security then you can always bring this box up again and disable the macros before you exit Excel.

Now that you have the developer tab added to the Ribbon you can do things like bring up the Visual Basic Editor, run macros, record macros, and insert form objects onto your spreadsheets. First, let's have a look at the Visual Basic Development Environment. This is, after all, where you'll be writing all your code.

2.2 The Visual Basic Editor

To open the Visual Basic editor, go to the “Code” panel on the Developer tab and click on the **Visual Basic** button. This should open a window like this.



It might seem like a lot at first, but you'll get accustomed to it soon enough.

The big grey area is where you'll write your code. The reason it's grey above is because no coding window has been opened yet. To open up a coding screen, click on **Insert >**

Module. The new white area is where you'll write your code.

You can try running the classical first program by typing in the following code into this window:

```
Sub HelloWorld()

    MsgBox ("Hello World!")

End Sub
```

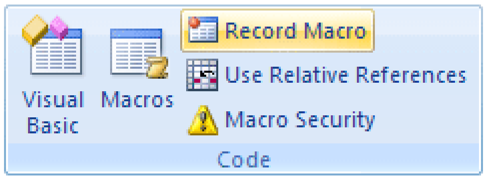
To run your program, locate the play button in the toolbar above, , and click "Run Sub/Userform."

Congratulations on running the first VBA program. In the case of Visual Basic *for Applications,* these programs that you'll be writing are called **Macros**.

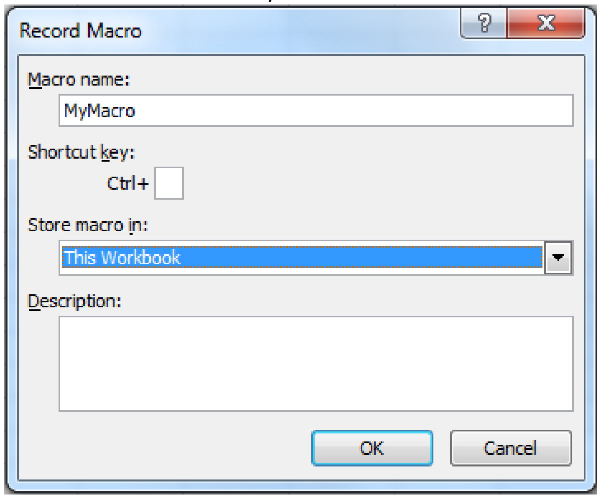
2.3 Recording a Macro

This feature is your new best friend. "Record Macro" is a tool that will record whatever actions you use in Excel (selecting cells, typing, creating graphs etc.). It will then create a new module, and have everything you just did translated into VBA code.

Let's start with something simple. Go to the developer tab and click on the **Record Macro** button.



Name it whatever you want, and make sure the macro will be stored in "This Workbook".


Record Macro

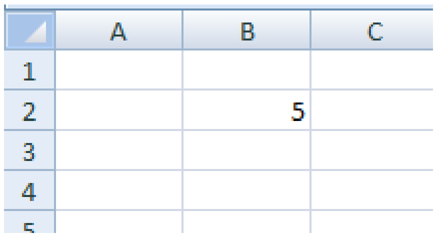
Macro name: MyMacro

Shortcut key: Ctrl+

Store macro in: This Workbook

Description:

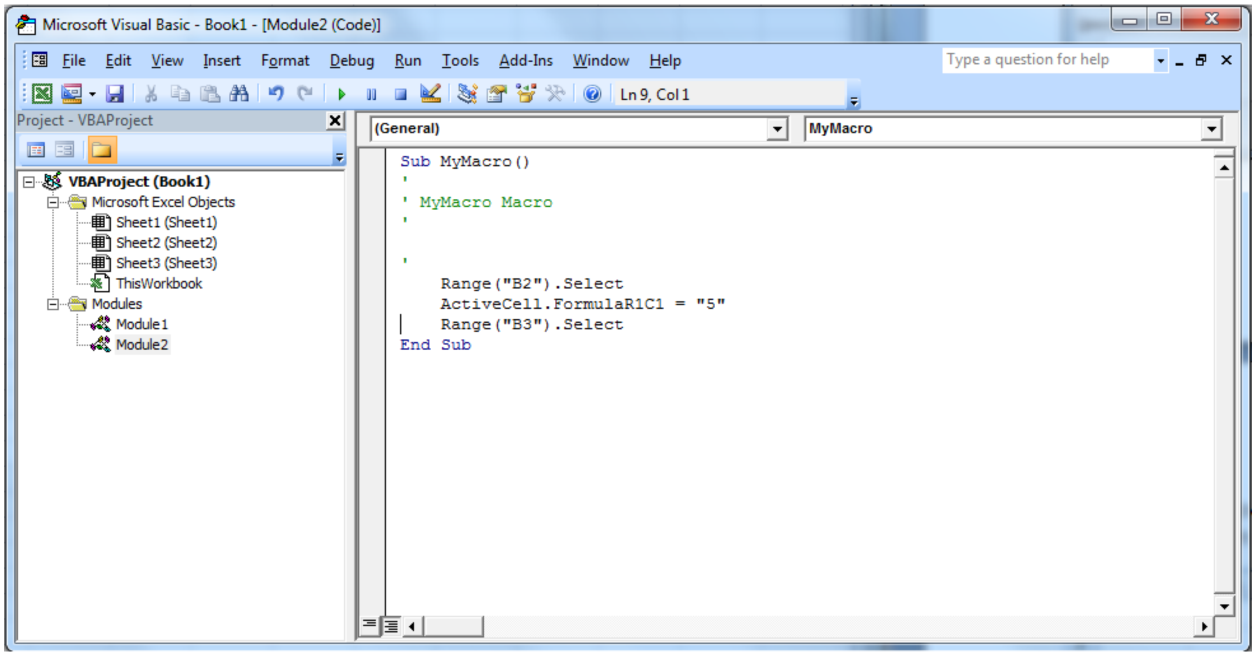
Press OK. You're now free to do anything, and it will be recorded in a new module. Let's just grab a random cell, and type something in. We'll type "5" into cell B2.



	A	B	C
1			
2		5	
3			
4			

Go back to the "Record Macro" button, and click **Stop Recording.**

To view the actions you just did in code, open the VB Editor. You'll see that there is a new module after the one you created before. Double click on this module to see the code:



```
Sub MyMacro()
'
' MyMacro Macro
'

'
    Range("B2").Select
    ActiveCell.FormulaR1C1 = "5"
    Range("B3").Select
End Sub
```

2.3.1 EXERCISE 1 - Macro Recording

Record a new macro to make some random text a different font, a different font size, and a different colour.

When you complete the exercise above, you should find that Excel adds quite a lot of code for what seems like simple job. However, you should bear in mind that most of what Excel adds is not needed, since it tends to overdo it - your own VBA code will be a lot shorter!

2.4 Excel Dot Notation

Excel VBA uses dot notation to separate the various things you can access and manipulate with the programming language. Dot notation is hierarchical, and usually starts with an object. (In Excel, an object is the thing you're trying to manipulate, such as a worksheet.) After the object, you type a dot. You then specify what you want to do with this object, or what you want to manipulate. The doing is called a method. The manipulating is done via properties or parameters.

If all this is confusing, let's try and clear it up.

Think of a television. This is an object. We can notate it like this:

tv

OK, all very simple so far. But you'll need some more information if you were going to buy a television. One thing you may want to know is how big the tv is. To add a size property, you'd do this:

tv.size

You'd want this to equal something, though, so add an equal sign and a size:

tv.size = "55 inch"

We now have an object (the tv) and a property (the size). We also have a value for the size (55 inch).

If we wanted to buy this tv then we'd be doing something (buying). We can call this "doing" a method. It is a method of the tv:

tv.buy

Example commands in Excel VBA is something like,

Range("A3").Value = "Bacon"

Which will write "Bacon" into cell A3. Likewise, the following command (a method) will delete the 3rd row in the currently active Excel sheet:

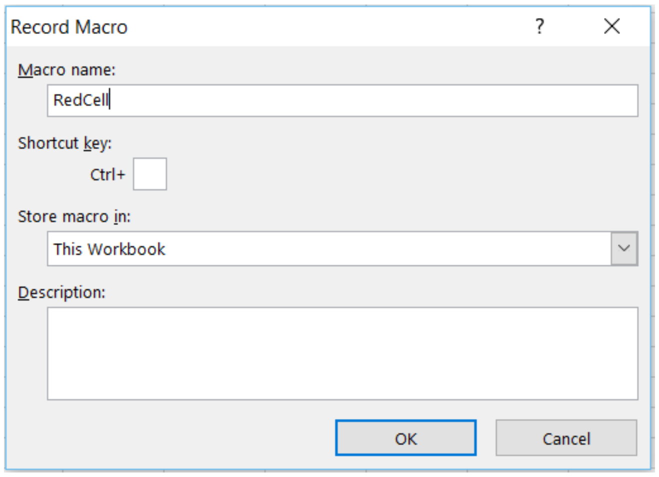
Rows(3).Delete

These are just examples, we'll get in to actually introducing these commands soon.

2.5 Adding a Button to a Spreadsheet

A fun feature in Excel is that you can assign macros to buttons on a spreadsheet, and turn the spreadsheet into a sort of user interface.

To demonstrate this, we'll start by creating a new macro. Let's record a macro that makes a cell red.



Record Macro

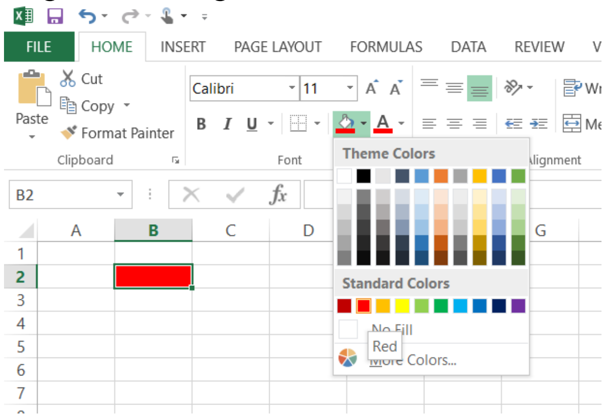
Macro name: RedCell

Shortcut key: Ctrl+

Store macro in: This Workbook

Description:

Begin recording a macro, name it whatever you want, and Press OK.



Pick any cell, make it red, and then go back to the Developer toolbar to stop recording the macro. An alternate way of viewing your code directly is by clicking on the **Macros** button on the developer toolbar. Press this button to access a list of all your macros, and select the one you just created. You should see a sub like this:

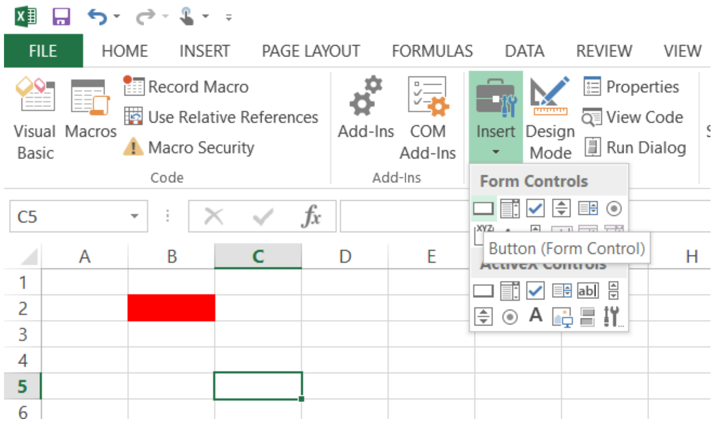
```
Sub RedCell()
'
' RedCell Macro
'

    Range("B2").Select
    With Selection.Interior
        .Pattern = xlSolid
        .PatternColorIndex = xlAutomatic
        .Color = 255
        .TintAndShade = 0
        .PatternTintAndShade = 0
    End With

End Sub
```

Remember that recording macros tends to overdo things. Because you initially picked the cell you wanted to color red, you might see the following line, depending on which cell you chose:

Range("B2").Select

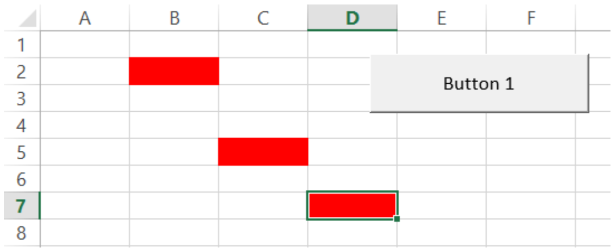
Delete this line. We're doing this because we don't want our macro to always color the same cell when we re-run it again. Now that the macro doesn't pick a cell itself, it will just execute the rest of the code on whatever cell was already selected.

To create a button, return to your spreadsheet, and go to the Developer toolbar. Click on the **Insert** button in the **Controls** panel, and select the small **Button** icon.



This will allow you to click-and-drag to create a button. Upon placing your button, you will be asked to pick the macro you want to execute when the button is clicked (in versions prior to Excel 2013, you have to right-click the button and choose **Assign Macro**).

Voila! This button will now take any cells you have selected, and make it red.



Note that you may also rename the label on the button by right-clicking it and pressing **Edit Text**.

2.5.1 EXERCISE 2 – Make a button that clears all colors.

Create another button that removes the fills from **all** the cells on the spreadsheet.

Hint: The small grey triangle at the top-left corner of the cells will select all the cells in the spreadsheet.

2.6 Comments

Comments are in green by default, and can be created with an apostrophe. All the code on a line after an apostrophe will be a comment.

You've now become familiar with how to create macros. You see that you do not need the damn semi-colons anymore, and that unlike C, new lines have significance. The following chapters will teach you the rest of the syntax for basic programming. You should know how programming works at this point, since this document will only serve as a brief reminder of programming methods, and syntax.

3 Variables

Often, in VBA, we'd like to dedicate a place in memory to a variable – a dedicated value your code can modify. These should always be declared at the beginning of a program. For example if we want to declare the variable "x" as an integer, we type in the following line:

Dim x As Integer

3.1 Types of Variables

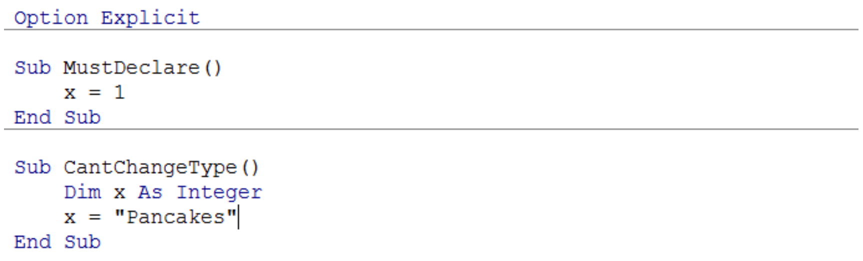
The following table summarizes the most common types of variables you'll need to declare:

Declaration Syntax	Variable Type	Description
Dim x As Integer	Integer	Can be any whole number from -32678 to 32676 (16 bits of memory are allocated)
Dim x As Long	Integer	Can be any whole number from -2,147,483,648 to 2,147,483,647 (32 bits of memory are allocated)
Dim x As Single	Float	This is what you use for numbers with decimals.
Dim x As Double	Float	This is what you use for really, really, really long decimals, numbers can be as big as 10^{300}
Dim x As String	Text	The variable you use if you want to assign a value of something like "Bacon" to 'x'.
Dim x As Boolean	Boolean	This variable can only have values of **TRUE** or **FALSE** (or 1 and 0, respectively). You would use them as such: **x = TRUE**
Dim x As Variant	All of them.	A variable that can change mid-code to be of whatever type it needs to be!

By default, if you introduce a variable in your program without declaring it, Excel will automatically assume it to be a Variant. You might think this is OK, but Variants take up a lot of memory, and will make your program slower. Therefore, and for many other reasons, you should declare your variables. If you add the line:

Option Explicit

At the beginning of a module, you will be forced to declare all your variables, as Excel will no longer make them Variants for you. For example, the two following programs/subs will throw errors:



```
Option Explicit

Sub MustDeclare()
    x = 1
End Sub

Sub CantChangeType()
    Dim x As Integer
    x = "Pancakes"
End Sub
```

Including this statement is completely optional however, it is simply good programming practice.

3.2 Variable Practice – Properly Referencing the Spreadsheet

Once we declare a variable, we can simply assign it a value with the equals sign. For example, for example let's say we've declared an integer 'x'. Then we can assign it a value of 10:

x = 10

And that's all there is to it. Similarly, suppose we declare a string variable 'name'. Then we can assign it a value of "Harry":

name = "Harry"

Note that the quotation marks are necessary for strings. Once this is done, we probably want to do something with these variable, so we need to reference the worksheet. So far, we've done this with a line like:

Range("A1").Value = x

This would put the value 10 into cell A1. However, there are multiple "A1" cells in an Excel workbook – there's one on every sheet! By default, Excel will pick the "A1" that is on the active sheet, i.e. the sheet that is open on your screen.

3.2.1 The Worksheets Object

Often times, you'll have a program that interacts with multiple sheets, so we need to know how to reference a cell on another sheet. We do this with the 'Worksheets' object:

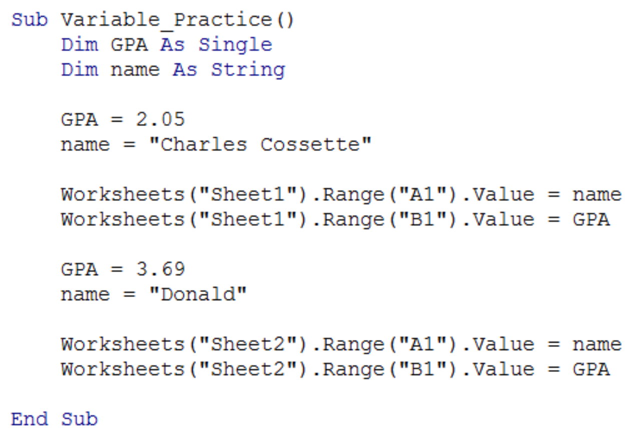
Worksheets(2).Range("C5").Value = 6

This will write the value 6, into cell C5, on the 2nd sheet from the left in the currently active Excel workbook. Suppose this sheet had the name "Sheet2" or you renamed it "Tuesday", then the following lines of code accomplish the same task as above, respectively:

Worksheets("Sheet2").Range("C5").Value = 6
Worksheets("Tuesday").Range("C5").Value = 6

Note that the quotation marks are necessary when referring to the sheet's name as opposed to its position on the tabs at the bottom of the spreadsheet (its index).

Let's summarize what we've done so far in an example program:



```
Sub Variable_Practice()
    Dim GPA As Single
    Dim name As String

    GPA = 2.05
    name = "Charles Cossette"

    Worksheets("Sheet1").Range("A1").Value = name
    Worksheets("Sheet1").Range("B1").Value = GPA

    GPA = 3.69
    name = "Donald"

    Worksheets("Sheet2").Range("A1").Value = name
    Worksheets("Sheet2").Range("B1").Value = GPA

End Sub
```

See what this does for yourself. Note that if "Sheet2" doesn't actually exist, it will throw an error.

3.2.2 The Workbooks Object

This, actually, also extends one level up to workbooks. If we have more than one Excel workbook open we use the 'Workbooks' object. The following lines of code write the value 5 into cell A1, on "Sheet1", in "My Excel File" and "My Other Excel File" respectively.

Workbooks("My Excel File.xlsx").Worksheets("Sheet1").Range("A1").Value = 5
Workbooks("My Other Excel File.xlsx").Worksheets("Sheet1").Range("A1").Value = 5

3.3 Mathematical Operators

The following table summarizes the basic mathematical operators, there's nothing really surprising about what they are or how to use them.

Operator	**Syntax**
Addition	**a = 4 + 3**
Subtraction	**a = 4 – 3**
Multiplication	**a = 4 * 3**
Division	**a = 4 / 3**
Exponentiation	**a = 4 ^ 3**
Brackets	**a = 4 * (2 + 1)**

Note that proper 'BEDMAS' mathematical hierarchy applies, and that those numbers up there could have been other variables too. For example, if you declared a variable 'x' and gave it a value of 5, then the line:

a = x + 2

Would assign 'a' the value of 7. Likewise, you can also use variables recursively as such:

i = i + 1

To increase the value of 'i' by 1.

3.4 Reading Values from the Spreadsheet

This is as simple as this:

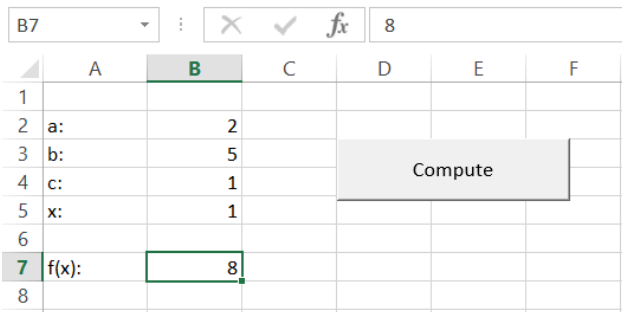
x = Worksheets("Sheet1").Range("A1").Value

This line would make 'x' whatever value was stored inside cell "A1". You can also switch the equality to fill that cell with the value of the variable.

3.5 EXERCISE 3 – Code a function

A quadratic equation is of the form: *$f(x) = ax^2 + bx + c$*

Have cells on a spreadsheet be dedicated to the values of 'a', 'b', 'c', and 'x'. Next, create a button that computes the result of the function, and displays it in another cell, as such:



	A	B	C	D	E	F
1						
2	a:	2				
3	b:	5				
4	c:	1				
5	x:	1				
6						
7	f(x):	8				
8						

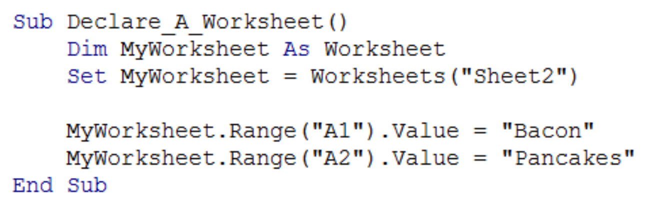
This might not seem very useful since we can just type in a formula directly into the cell, but we're getting there.

3.6 A Useful Shortcut – Declaring Worksheets and Workbooks as Variables

You might notice that writing

Worksheets("Sheet2").Range("A2").Value = "Pancakes"

Becomes a little cumbersome every time you have to fill a cell. VBA lets you replace the **Worksheets("Sheet2")** part in the following manner:



```
Sub Declare_A_Worksheet()
    Dim MyWorksheet As Worksheet
    Set MyWorksheet = Worksheets("Sheet2")

    MyWorksheet.Range("A1").Value = "Bacon"
    MyWorksheet.Range("A2").Value = "Pancakes"
End Sub
```

Which will become useful once you start writing bigger programs. Note that you can also do this to workbooks if you have different workbooks open:

Dim wb1 As Workbook
Dim wb2 As Workbook
Set wb1 = Workbooks(“My Workbook.xlsx”)
Set wb2 = Workbooks(“Book2.xlsx”)

wb1.Worksheets(“Sheet2”).Range(“A1”).Value = 5

4 Conditional Logic

4.1 If Statement

If statements have the following form:

If Condition_To_Test Then

'CODE HERE

End If

Where the Condition_To_Test is some sort of equality that must evaluate to true for the code to execute.

4.2 ElseIf and Else Statements

These have the following form

If Condition_To_Test Then

‘CODE HERE

ElseIf Next_Condition_To_Test Then

‘SOME OTHER CODE HERE

Else

‘CODE IF NOTHING ABOVE IS TRUE

End If

Both of these are optional in an If Statement.

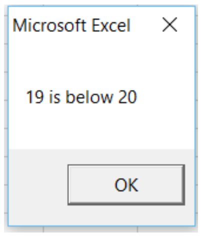
4.3 Conditional Operators

These are the equality/inequality symbols you should use in a logical condition.

Operator	Meaning
=	Has a value of
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
< >	Not equal to

The following example combines all the information above to show you a program that displays a different message box depending on how big the variable 'MyNumber' is.

```
Sub LogicTest()
    Dim MyNummber As Integer

    MyNumber = 19

    If MyNumber = 20 Then
        MsgBox (MyNumber & " is 20")
    ElseIf MyNumber > 20 Then
        MsgBox (MyNumber & " is Greater Than 20")
    Else
        MsgBox (MyNumber & " is below 20")
    End If

End Sub
```



19 is below 20

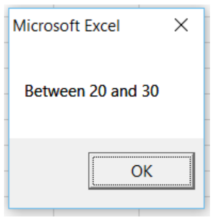
4.4 Logical Operators

The following can be used if you want to have more than one condition on the same line, in an If Statement.

Operator	Meaning
Not	Test if value is NOT something
And	Test for more than one condition
Or	Test if the value is either OR something
Xor	Test if one and only one value is true

The And, Or, and Xor statements must be put in between two logical conditions:

```
Sub LogicTest2()
    Dim Age As Integer

    Age = 21

    If Age > 20 And Age < 30 Then
        MsgBox "Between 20 and 30"
    Else
        MsgBox "Not Between 20 and 30"
    End If

End Sub
```



Between 20 and 30

Whereas the Not statement is used in a line as follows:

```
If Not Condition_To_Test Then
    'CODE HERE
End If
```

The not statement basically 'inverts' the If statement. So now the condition to test must evaluate to FALSE for the code to execute. For example, if the 'Condition_To_Test' was replaced by '10 = 20,' which obviously isn't true, then that code would actually end up executing.

4.5 Some Built-in Functions

4.5.1 ActiveCell Referencing

Before moving on to the next exercise, you need to know some more extremely useful built-in functions. **ActiveCell** is one of them, and you can use it to get information about the cell you have selected. For example, you can assign the selected cell's content to a variable with:

```
x = ActiveCell.Value
```

You can also assign the active cell's row and column numbers to variables:

```
currentrow = ActiveCell.Row
currentcolumn= ActiveCell.Column
```

Now that the **.Column** property returns a number whereas columns are letters. A corresponds to 1, B to 2, etc.

4.5.2 The 'Cells' Object

Recall that you have been using **Range("A2")** or something, to point to a cell. There is another very convenient way to do this:

```
Worksheets("Sheet1").Cells( [Row_Number] , [Column_Number] ).Value
```

For example, the two following lines accomplish the exact same thing:

```
Worksheets("Sheet1").Range("A2").Value
Worksheets("Sheet1").Cells( 2 , 1 ).Value
```

This particular method is handy because you can put mathematical expressions inside the row and column slots in the Cells object.

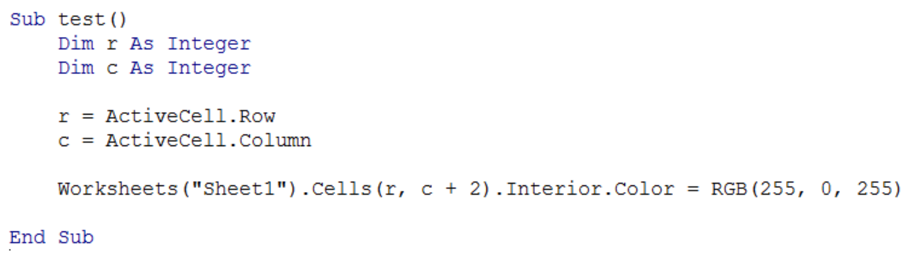
4.5.3 The RGB Function

To change the color of a cell, you would do something like this:

```
Cells( 4 , 3 ).Interior.Color = RGB( 0 , 255 , 0 )
```

The RGB function (stands for Red Green Blue) is a function that will return the value for the color once you specify its three arguments. The three arguments require a number from 0 to 255, and correspond to how much of a certain color is added to the 'mix'. Red, Green, and Blue correspond to first, second, and third arguments (or slots) respectively. For example, the line above would color in cell C4 pure green. Feel free to play around with the RGB function.

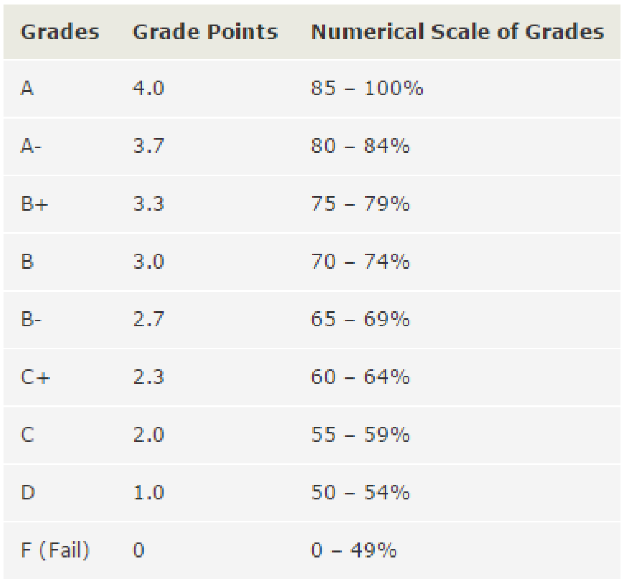
To illustrate all this, the following program will color in a cell purple, located two to the right of the active cell.



```
Sub test()
    Dim r As Integer
    Dim c As Integer

    r = ActiveCell.Row
    c = ActiveCell.Column

    Worksheets("Sheet1").Cells(r, c + 2).Interior.Color = RGB(255, 0, 255)

End Sub
```

4.6 EXERCISE 4 – Automatic Grader (UNFINISHED)

Go to the sheet in the Exercise Book called "EXERCISE 4." Create a button on that spreadsheet that will evaluate the grade of any student you have selected (i.e. you have selected a cell in the same row as the student), and then displays the appropriate grade in the "Grade" column. The letter grades can be assigned according to McGill's standard:



Grades	Grade Points	Numerical Scale of Grades
A	4.0	85 – 100%
A-	3.7	80 – 84%
B+	3.3	75 – 79%
B	3.0	70 – 74%
B-	2.7	65 – 69%
C+	2.3	60 – 64%
C	2.0	55 – 59%
D	1.0	50 – 54%
F (Fail)	0	0 – 49%

5 Loops

Loops are used to make your program do things over and over again.

5.1 For Loops

For loops require a variable which value gets changed automatically. For loops are of the form:

```
For j = [StartingValue] To [EndingValue] Step [ChangeAmount]
  'CODE THAT GETS REPEATED
Next
```

The **Step** statement is optional, and if absent defaults to 1. You can analyse this example to see how they work.

```
Sub LoopExample()

    Dim StartNumber As Integer
    Dim EndNumber As Integer

    EndNumber = 5

    For StartNumber = 1 To EndNumber

        MsgBox StartNumber

    Next StartNumber

End Sub
```

5.1.1 EXERCISE 5 – Factorial Evaluator

Write a program that reads a variable from a cell, in fills the cell next to it with the factorial of that number

n! = (n)*(n-1)*(n-2)*….*3*2*1

Make sure your program also works for the value of 0!, since this is equal to 1.

5.2 For Each Loops

For Each loops are written in the following form:

```
For Each variable_name In collection_name
    'CODE HERE
Next variable_name
```

For loops can be used to go through each element in a collection of things. A common example of a collection is a range of cells. Using a **For each** loop for such a case would make it look like:

```
For Each MyCell In Range("A2:A6")
     'CODE HERE
Next MyCell
```

5.2.1 EXERCISE 6 – Factorial Evaluator Part 2

Take the code you just made for the previous for loop exercise, and put it in a **For Each** loop that evaluates the factorial of several cells.

5.3 Do While loops

These are essentially just a regular while loop:

```
Do While [CONDITION]
    'CODE HERE
Loop
```

These need a condition to become false eventually, or the loop will go on forever, and Excel will crash.

Try coding the factorial program with a **Do While** loop now!

5.4 EXERCISE 7a – Toss a coin 1000 times.

Simulate the tossing of a coin 1000 times, and see how many heads/tails you get. To do this, you will need the random number function. The line

```
x = Rnd
```

Will give the value of x a random decimal between 0 and 1. You can say that if x is less than 0.5, it is considered a head. Otherwise, it is tails.

5.5 EXERCISE 7b – Toss 10 coins 1000 times

Even better practice. Write a code that simulates tossing 10 coins 1000 times, and counting how many of those 10 coins are heads each time. The goal is to see how many times out of 1000 you get 2 heads, 3 heads, 4 heads etc.. Put your results in a table in a spreadsheet, so you can create a graph that has "Number of Heads" on the x-axis, and "Number of Times that many heads has shown up" on the y-axis.

The result should resemble the binomial distribution.

6 Strings and String Functions

"Strings," for our purposes, are just text. Numbers have certain operations, like adding and subtracting. In VBA, strings also have some functions that lets us play around with them. You would declare a string variable in the following way:

```
Dim MyString As String
MyString = "Some Text"
```

6.1 LCase and UCase

These convent the entire text to lower case and upper case letters respectively:

```
MyNewString = LCase(MyString)
```

Would return **"some text"**.

```
MyNewString = UCase("Some Text")
```

Would return "**SOME TEXT".**

6.2 Trim, Len, and Space

Trim is used to delete unwanted white space at the beginning or end of your string.

MyString = Trim(" some text   ")

Would return **"some text."**

Len returns the length of the string (how many characters, including spaces)/

x = Len("Hello")

Would return **5.**

Space is used to add spaces to an array

MyString = "some text" & Space(5)

Would put 5 spaces after the string to give **"some text   ".**

6.3 Replace

The replace function will replace parts of a certain string.

MyString = Replace("Micrasaft", "a" , "o")

Would return **"Microsoft"**. Notice that 3 arguments are necessary

6.4 InStr, InStrRev, StrReverse

InStr returns to you the position of a certain with within a string.

x = InStr("charles@mail.mcgill.ca", "@")

Would return to you the number 8. It searches the string from left to right. **InStrRev** does the same thing, except searches from right to left.

StrReverse is simple, it just reverses a string.

MyString = StrReverse("Hello")

Would return **"olleH".**

6.5 The Left, Right, and Mid Functions

These are the functions that are used if you want to snip out certain sections of a string.

The **Left** function grabs characters from the left of a string.

MyString = Left("Bacon Pancakes", 5)

Would return **"Bacon".**

Right does the same thing, just from the end of the string

MyString = Right("Bacon Pancakes", 8)

Would return **“Pancakes”.**

The **Mid** function is similar, it acts like the **Left** function, but can start grabbing characters from a specified position in the string, rather than at the beginning. It has the form:

Mid(string_to_search, start_position, number_of_characters_to_grab)

For example:

MyString = Mid(“Blues Pub”, 6 , 3)

Would return **“Pub”.**

Armed with the knowledge of loops and strings. It’s time to do some real exercises

6.6 EXERCISE 8 – String Practice

Suppose you have a cell in an Excel sheet with a product code: **PD-23-23-45**

However, the notation is incorrect. The “PD” should be a “PDC” instead, and there should be no hyphens. The final result should be **PDC232345**.

Write a VBA code to make this change. Display the result.

6.7 EXERCISE 9 – Email Identifier

The goal of this exercise is to create a program that goes through a list of emails, and highlights the cells which have an email that is **not** a valid McGill email. (i.e. those that don’t end with “@mcgill.ca” or “@mail.mcgill.ca”).

The list of emails can be found in the sheet called “EXERCISE 9” of the Exercise Book.

6.8 EXERCISE10 – E-Week Registration Trend Graph

It would be useful for the E-Week to coordinators to notice when exactly people registered. This way, they can see how people responded to certain promotions. Each registration comes with a “Time Stamp” which is given as a string (see “EXERCISE 10” of the Exercise Book).

Write a program that uses the data given to you to produce a graph, with days on the x-axis (i.e. Day 1, Day 2, Day 3 etc, but these can be actual dates), and the number of people that registered that day on the y-axis.

Note: You don’t have to get your program to actually create the graph, but at least populate a range of cells from which you can easily create one yourself.

7 Arrays

Sometimes, you might need to declare a lot of variables at once. This is what an array as useful for. You can declare an array like this:

Dim MyArray(4) As Integer

This created 5 different variables of the integer type:

```
MyArray(0)
MyArray(1)
MyArray(2)
MyArray(3)
MyArray(4)
```

If you don't want your array to start at 0, you can change the range of numbers which index your array like this:

```
Dim MyArray(1 to 5) As Integer
```

7.1 Multi-Dimensional Arrays

You can create an array with several dimensions as well:

```
Dim MyArray(1 to 5, 1 to 2) As Integer
```

This would create the following set of variables:

```
MyArray(1,1)        MyArray(1,2)
MyArray(2,1)        MyArray(2,2)
MyArray(3,1)        MyArray(3,2)
MyArray(4,1)        MyArray(4,2)
MyArray(5,1)        MyArray(5,2)
```

7.2 The Split Function

A very convenient function – the **Split** function dices up a string by a "delimiter" and store the division into an array. Suppose you had the string **"17/4/2016"**, you can use the split function to isolate the 3 numbers.

```
Sub Example()

    Dim MyDate As Variant

    MyDate = Split("17/4/2016", "/")

End Sub
```

This would produce an array (which is why MyDate must be declared as a variant). The array would have the following values:

```
MyDate(0) = 17
MyDate(1) = 4
MyDate(2) = 2016
```

7.3 EXERCISE 11 – Use a For-Loop to Load an Array

Go to sheet "EXERCISE 11&12" of the Exercise Book. Here, you will see 1000 rows of data. Fill a [1000 x 3] two-dimensional array with this full data set. There is no need to split the date into separate elements of an array, just load the cells directly into an array. Prove that you did this correctly by then printing the contents of your array into the cells adjacent to our data.

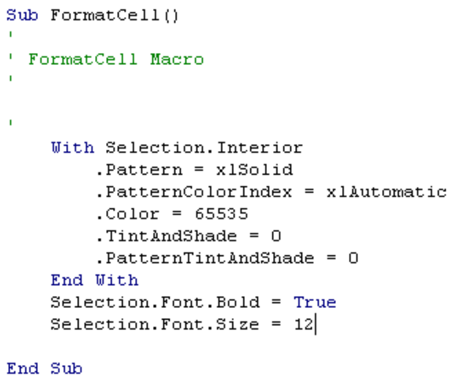
You will be using this data, and probably this code in a later Problem!

8 Subs and Functions

8.1 Subroutines

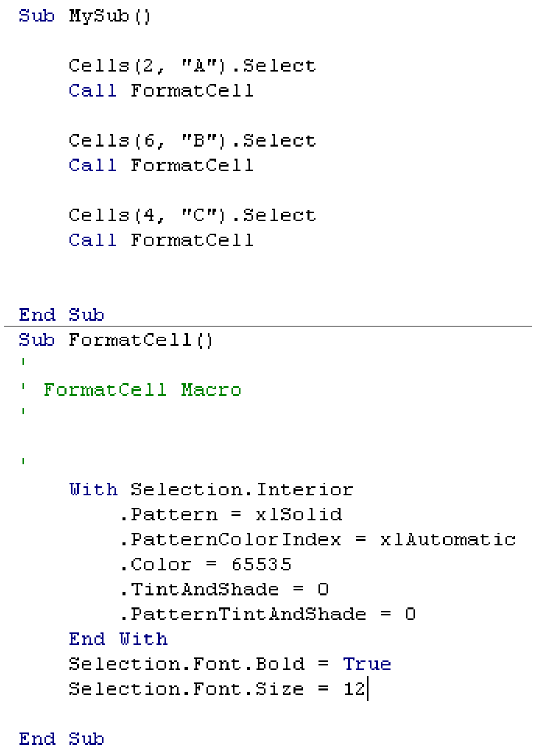
So far, everything you've been doing has been coding into subroutines. You can have several subroutines, and you can run one sub in the middle of the other. To do this, you would use the word **Call**.

Start by recoding a macro that highlights, bolds, and increases the font size of a cell. You should get something like this:



```
Sub FormatCell()
'
' FormatCell Macro
'

'
    With Selection.Interior
        .Pattern = xlSolid
        .PatternColorIndex = xlAutomatic
        .Color = 65535
        .TintAndShade = 0
        .PatternTintAndShade = 0
    End With
    Selection.Font.Bold = True
    Selection.Font.Size = 12

End Sub
```

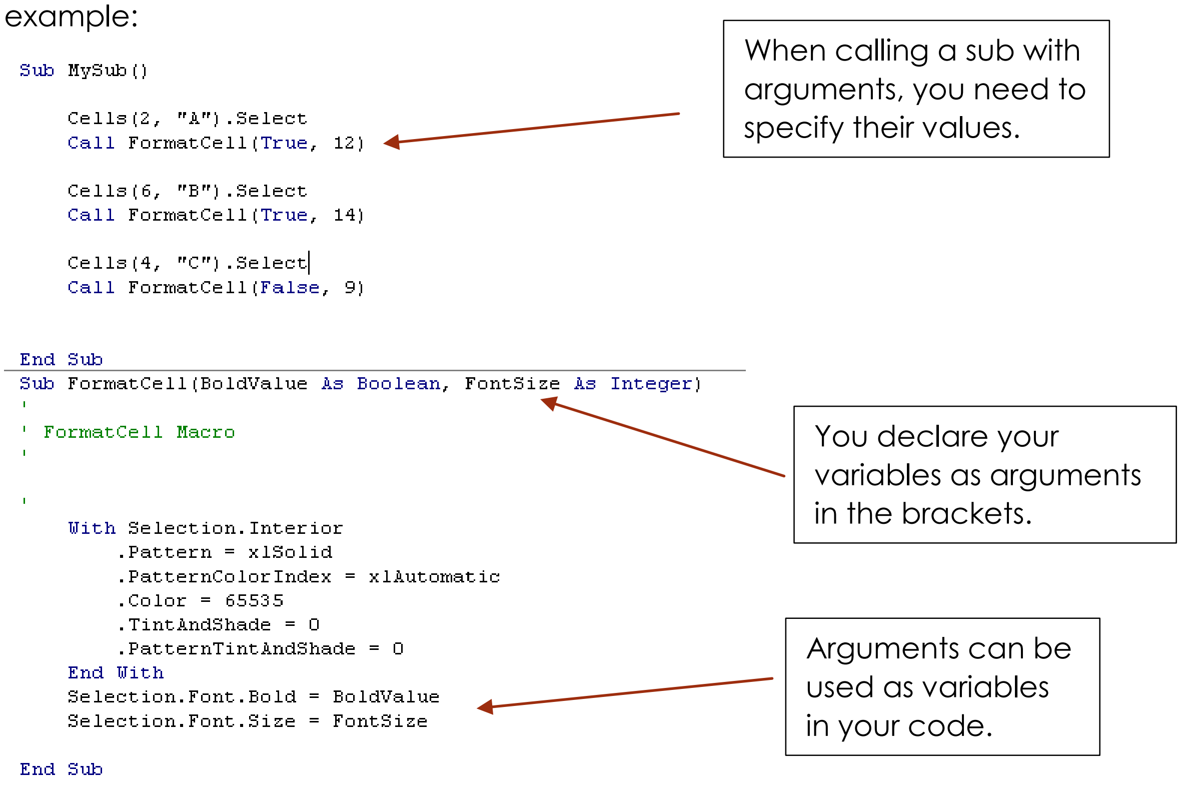
As you may notice, this will affect the cell you have currently selected. The following code will apply this formatting to cells A2, B6, and C4.



```
Sub MySub()

    Cells(2, "A").Select
    Call FormatCell

    Cells(6, "B").Select
    Call FormatCell

    Cells(4, "C").Select
    Call FormatCell


End Sub
Sub FormatCell()
'
' FormatCell Macro
'

'
    With Selection.Interior
        .Pattern = xlSolid
        .PatternColorIndex = xlAutomatic
        .Color = 65535
        .TintAndShade = 0
        .PatternTintAndShade = 0
    End With
    Selection.Font.Bold = True
    Selection.Font.Size = 12

End Sub
```

8.2 Passing Values to Subroutines

Ever wondering what those empty brackets were for? Let's go back to the previous example – edit the lines that make cells bold, and changes the font size. Replace the values with variables – they can be declared as arguments as per the following example:



```
Sub MySub()

    Cells(2, "A").Select
    Call FormatCell(True, 12)

    Cells(6, "B").Select
    Call FormatCell(True, 14)

    Cells(4, "C").Select
    Call FormatCell(False, 9)


End Sub
Sub FormatCell(BoldValue As Boolean, FontSize As Integer)
'
' FormatCell Macro
'

'
    With Selection.Interior
        .Pattern = xlSolid
        .PatternColorIndex = xlAutomatic
        .Color = 65535
        .TintAndShade = 0
        .PatternTintAndShade = 0
    End With
    Selection.Font.Bold = BoldValue
    Selection.Font.Size = FontSize

End Sub
```

When calling a sub with arguments, you need to specify their values.

You declare your variables as arguments in the brackets.

Arguments can be used as variables in your code.

However, you can obviously no longer run that sub by itself, since it will require values for its arguments.

8.3 Functions

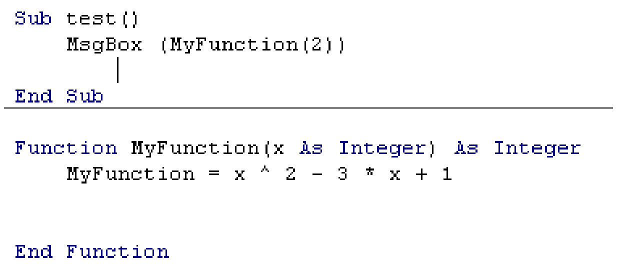
Functions work similar to subs, but have a slightly different declaration:

Function MyFunction (x as Integer) as Integer

Notice that the actual function is declared both with an argument, and as a variable type itself. Consequently, it needs to return a value:

MyFunction = x^2+3x-1

The following code will give you a message box displaying "-1".



```
Sub test()
    MsgBox (MyFunction(2))

End Sub

Function MyFunction(x As Integer) As Integer
    MyFunction = x ^ 2 - 3 * x + 1


End Function
```

8.4 Worksheet Functions

Worksheet functions refer to the functions that you're used to - the functions that you type into cells like **SUM, AVERAGE**, **COUNTIF, VLOOKUP** and many more. Turns out, these are all accessible in VBA. You can use them like this:

x = WorksheetFunction.Sum(Range("B1:B5"))

8.5 EXERCISE 12 – Client Payment Monitor

Revisit sheet "EXERCISE 11&12" from Exercise 11 in the Exercise Book. Write a program that counts how many times each client has been payed each year – indicated by a "YES" in the payment column. Then, display the results in the table shown in "EXERCISE 12" of the Exercise book.

The speed advantage of arrays would become clear in this exercise. It would be a lot faster to load all the data in an array, and then to scan your array, rather than to scan the worksheet itself.

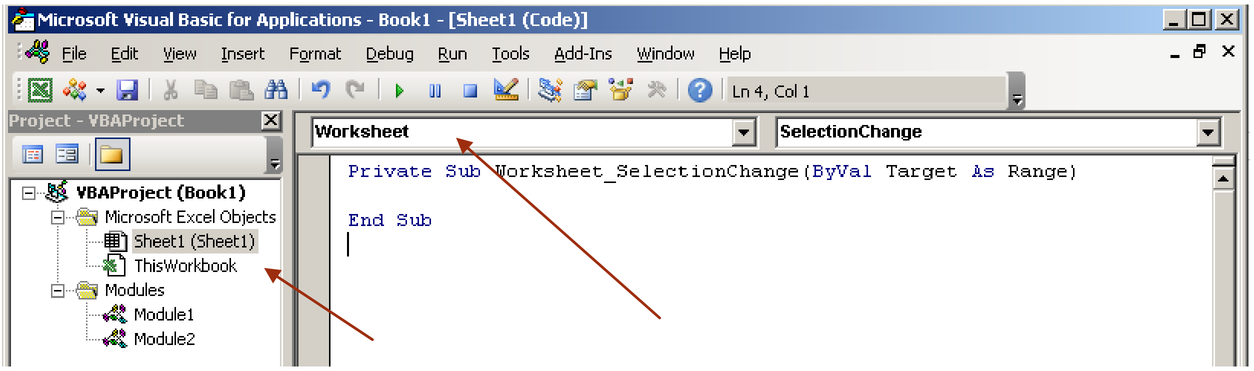
8.6 EXERCISE 13 – Consolidating Vendor Codes

See "EXERCISE 13" of the Exercise book. Here, you are given a list of vendor codes, along with the companies they represent. However, some companies have multiple vendor codes. Write a program that will go through this list, and create a consolidated list that has the company names in one column, and its associated vendor codes (separated by commas) in the other.

9 Events

This is what can make your Excel workbook truly come to life. So far, every single macro that we have run has been by us clicking on "Run". You can make your macro run automatically when specific events occur, such as a double click. There is, however, a finite list of events that Excel will recognize.

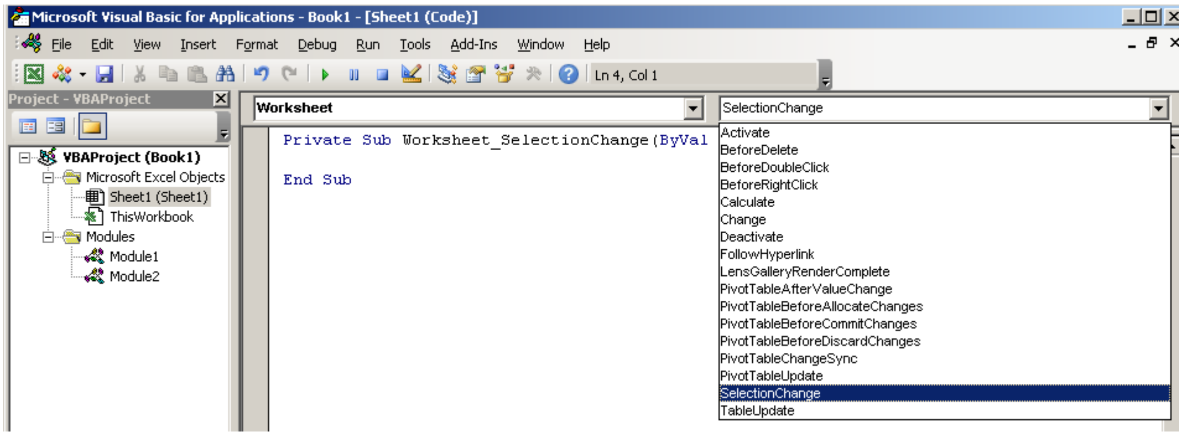
To access them, go to a Sheet or Workbook Module:



```
Private Sub Worksheet_SelectionChange(ByVal Target As Range)

End Sub
```

Click on the dropdown menu that says "General" and choose "Worksheet" (or "Workbook").

The default sub that shows up is a sub that will run every single time you change your selection, of anything. However, you can choose a different event if you want, by clicking on the dropdown menu that currently says “SelectionChange”.

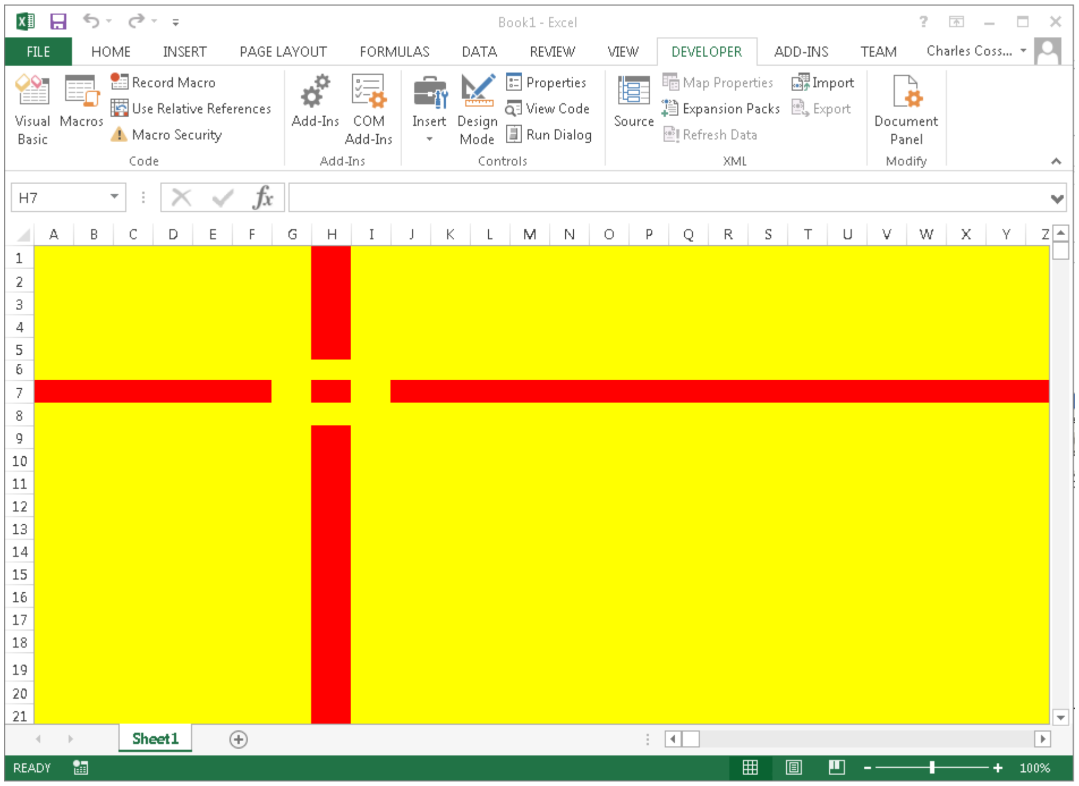


```
Private Sub Worksheet_SelectionChange(ByVal

End Sub
```

These are mostly left explanatory, and can be taken advantage of for creating very dynamic workbooks.

9.1 EXERCISE 13 – A Crosshair

Write a program using “SelectionChange” that will display a crosshair around whatever cell you have selected, as soon as you click it.



Bonus Extension: Upon double clicking, it highlights the selected cell blue, and it remains blue forever.

10 Final Remarks

With this, you are now armed with the basic knowledge of VBA. There are hundreds of functions that you have not been introduced to yet, but google will now do the remainder of the teaching.

Here are a few more useful things you can explore on your own, which can become especially useful in the industrial world:

- Opening and editing other Workbooks with code.
- Opening and navigating other Microsoft programs with code (i.e. automatically sending emails!)
- Userforms (these are little menus that you can create, made functional by code.)
- Writing code that writes code! (Getting quite advanced here).