SIM 1005: FUNDAMENTALS OF SPREADSHEETS

SIN 1002: INTRODUCTION TO WORKSHEETS

SIM 1005: FUNDAMENTALS OF SPREADSHEETS

SIN 1002: INTRODUCTION TO WORKSHEETS

SIM 1005: FUNDAMENTALS OF SPREADSHEETS

SIN 1002: INTRODUCTION TO WORKSHEETS

SIM 1005: FUNDAMENTALS OF SPREADSHEETS

SIN 1002: INTRODUCTION TO WORKSHEETS

SIM 1005: FUNDAMENTALS OF SPREADSHEETS

SIN 1002: INTRODUCTION TO WORKSHEETS

ASSESSMENT METHODS :

A.	**CONTINUOUS ASSESSMENT (TBC)** **i. WRITTEN TEST and/or** **ii. LAB-TEST and/or** **iii. Assignment.**	**:**	**50%**
B.	**FINAL (LABORATORY)**	**:**	**50%**
	TOTAL	**:**	**100%**

Contents :

1.4 Formatting data

1.4.1 Cell formatting
1.4.2 Formatting rows and columns
1.4.3 Hiding rows and columns
1.4.4 Keeping row and column headings in view

1.5 Formulas

1.5.1 Creating a formula
1.5.2 How formula are evaluated
1.5.3 Relative cell referencing
1.5.4 Absolute cell referencing

1.6 Functions

1.6.1 Using AutoSum
1.6.2 Basic functions
1.6.3 The IF() function
1.6.4 Nested functions

1.7. Chart

1.7.1 Create a chart
1.7.2 Modify a chart
1.7.3 Copy a chart to Word

1.8. Printing

1.7.1 Print preview
1.7.2 Preparing to print
1.7.3 Printing a worksheet

CHAPTER 2 : MATRIX OPERATIONS

CHAPTER 3 : SYSTEM OF LINEAR EQUATIONS

CHAPTER 4 : EXCEL SOLVER

CHAPTER 5 : FINDING THE ROOTS OF EQUATIONS

CHAPTER 6 : NUMERICAL INTEGRATION

6.1 Introduction
6.2 Rectangular Rule
6.3 Trapezoidal Rule
6.4 Simpson's Rule

CHAPTER 7 : ORDINARY DIFFERENTIAL EQUATIONS

7.1 Introduction
7.2 Euler's method
7.3 Taylor methods

CHAPTER 8 : EXCEL VISUAL BASIC OF APPLICATIONS

8.1 INTRODUCTIONS
- 8.1.1 What is Excel VBA?
- 8.1.2 Course Requisites
- 8.1.3 What software do I need?
- 8.1.4 Do I need any programming skills?

8.2 How to Add the Developer Toolbar to Excel
8.3 The Excel Visual Basic for Applications Development Environment
8.4 Watch as Excel records a Macro
8.5 Excel Dot Notation
- 8.5.1 The Range Property
- 8.5.2 The Worksheets Object

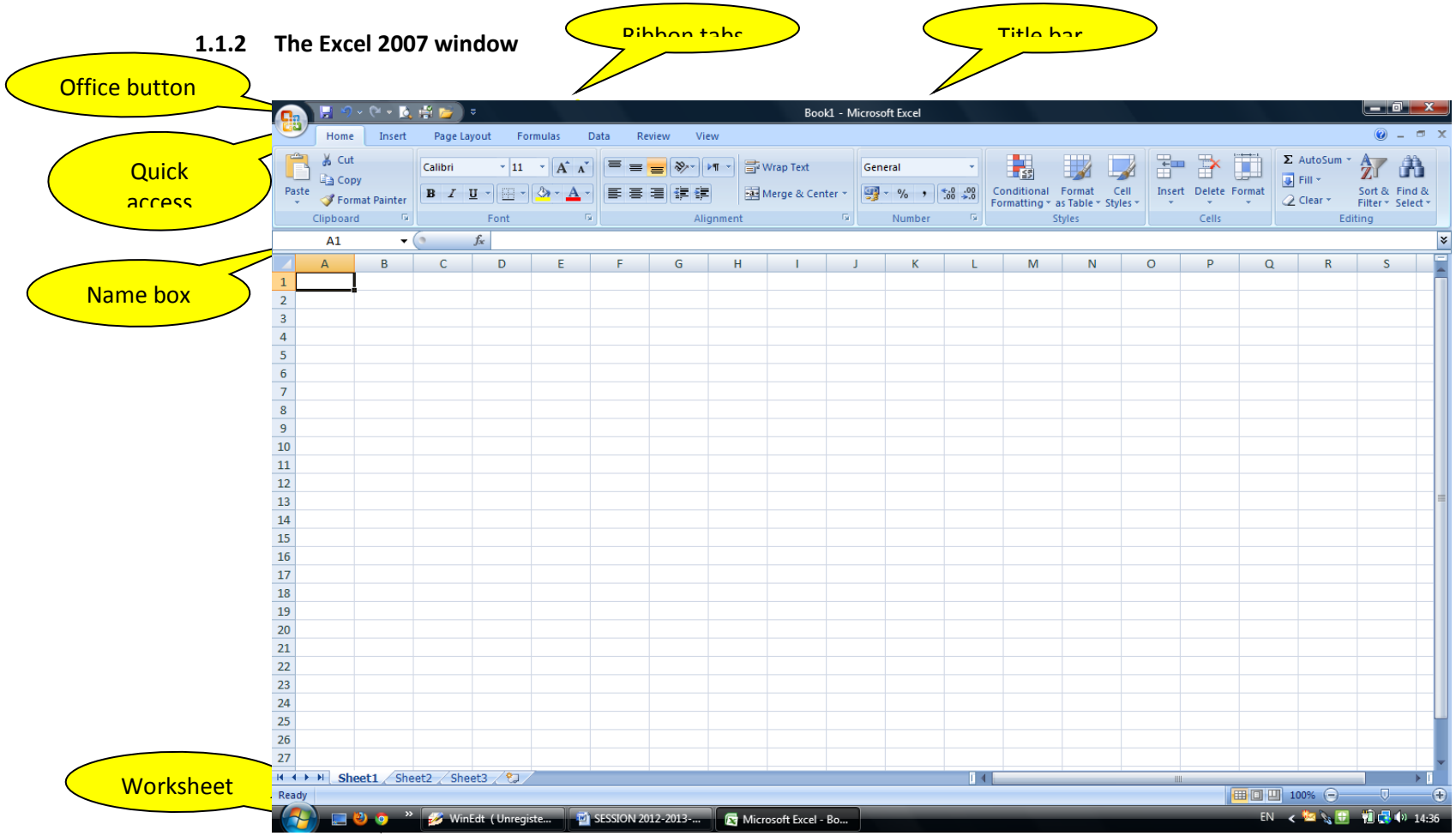
CHAPTER 1 : BASIC IN EXCEL

1.1 Introduction to Microsoft Excel 2007

1.1.1 What is Excel?

- Excel is all about numbers.
- There's almost no limit to what you can do with number in Excel, including sorting, advanced calculation and graphing.

1.1.2 The Excel 2007 window



Starting Excel

- If you have an icon on the desktop for Excel, then you have to double-click it to open Excel.
- Alternatively, click the Start button and then select All Programs, Microsoft Office, Microsoft Excel.

1.1.4 Closing Excel

- Close Excel by clicking the X on the far right of the title bar.

1.1.5 Selecting cells

- Using the mouse :
    1. Click on a cell to select it.
    2. You can select a range of adjacent cells by clicking on the first one, and then dragging the mouse over the others.
    3. You can select a set of non-adjacent cells by clicking on the first one, and then holding down the [CTRL] key as you click on the others.

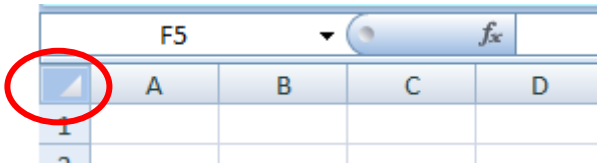
- Using the keyboard :
    1. Use the arrow keys to move the desired cell, which is automatically selected.
    2. To select multiple cells, hold down the [SHIFT] key while the first cell is active and then use the arrow keys to select the rest of the range.

1.1.6 Selecting rows or columns

- To select all the cells in a particular row :
    - Just lick on the row number (1, 2, 3, etc) at the left edge of the worksheet.

- To select all the cells in a particular column :
    - Just lick on the column heading (A, B, C, etc) at the top edge of the worksheet.

- You can quickly select all the cells in a worksheet by clicking the square to the immediate left of the Column A heading (just above the label for Row 1)



1.2 Entering data

1.2.1 First you need a workbook

- If you need to create a new workbook from inside Excel :
    1. Click on the Office button, select New and then Blank Workbook.
    2. Sheet 1 of a new workbook will be displayed on your screen, with cell A1 active.

- To open an existing workbook from inside Excel :
    1. Click on the Office button, select Open and then navigate to the drive and folder containing the file you want to open.
    2. Double-click on the required file name.

1.2.2 Data types

- It can be a number, text or formulas.

1.2.3 Data entry cell by cell

- To enter either numbers or text:
    1. Click on the cell where you want the data to be stored, so that the cell becomes active.
    2. Type the number or text.
    3. Press [ENTER] to move to the next row, or [TAB] to move to the next column. Until you've pressed [ENTER] or [TAB], you can cancel the data entry by pressing [ESC].

1.2.4 Deleting data

- Press delete button on your keyboard or right click on the cell that you want to delete and select delete.

1.2.5 Moving data

- Click on the cell that you want to move, drag to a new.

1.2.6 Copying data

- To copy existing cell contents to another area :
    1. Select the cells you want to move.
    2. Move the cursor to the border of the highlighted cells while holding down the [CTRL] key. When the cursor changes from a white cross to a hollow left pointing arrow (the copy pointer), hold down the left mouse button.
    3. Drag the selected cells to a second area of the worksheet, then release the mouse button.
- You can also copy the selected data using the ribbon icon or [CTRL]+[C], then click in the top left cell of the destination area and paste the data with the ribbon icon or [CTRL]+[V].

1.2.7 Saving a workbook

- As usual, you need to specify the file name, and its location (drive and folder).
    1. Click the office button and select ***Save,*** or click the Save icon on the Quick Access toolbar. If this workbokk has been saved before, then that's it-your worksbook will be save again with the same name and location.
    2. If it's the first time of saving this worksbook, then ***Save As*** dialogue box will open.
    3. type the new file name in the ***File Name*** field.
    4. Click ***Save*** button.

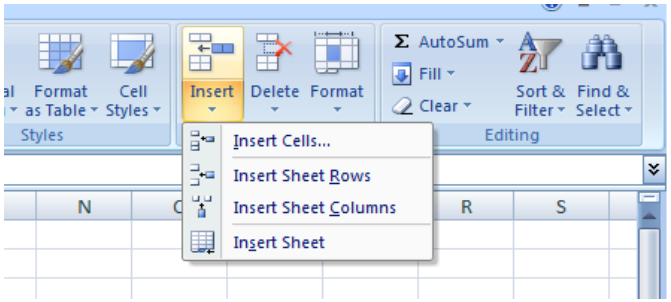
1.3 Editing data

1.3.1 Editing cell contents

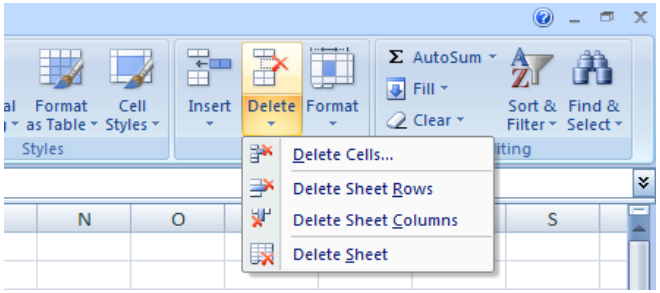
- To delete characters, use the [BACKSPACE] or [DEL] key.
- To insert characters, click where you want toinset them and then type. You can toggle between insert and overtype mode by pressing the [INSERT] key.

1.3.2 Inserting or deleting cells

- To insert a new cell :
    1. Select the cell next to which you want to insert a new cell.
    2. On the Home ribbon, find the Cells group and click ***Insert*** followed by ***Insert Cells***.



- To delete a cell :
    1. Select the cell that you want to delete.
    2. On the Home ribbon, find the Cells group and click ***Delete*** followed by ***Delete Cells***.



1.3.3 Inserting or deleting rows

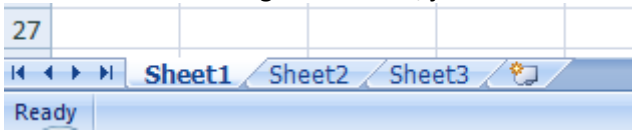
Refer 3.2.

1.3.4 Inserting or deleting columns

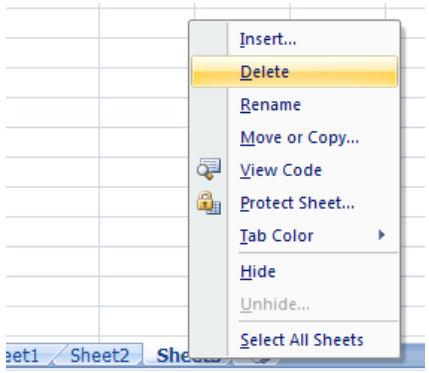
Refer 3.2.

1.3.5 Inserting or deleting a worksheet

- To insert a new worsheet at the end of the existing worksheets, just click the Insert Worksheet tab at the bottom of the screen.



- To delete a worsheet, select the worksheet that you want to delete, right click and select ***Delete***.



1.3.6 Moving or copying a worksheet

- To move or copy worksheet, right-click on the worksheet tab, and select ***Move or Copy***.

1.3.7 Renaming a worksheet

- To rename a worksheet, right-click on the worksheet tab, and select ***Rename*** or just double click at the worksheet tab and rename it.

1.4 Formatting data

1.4.1 Cell formatting

- The icons on the Home ribbon provide you with variety of formatting options. To apply any of these, just select the cell or cells that you want to format, and then click the desired icon.

1.4.2 Formatting rows and columns

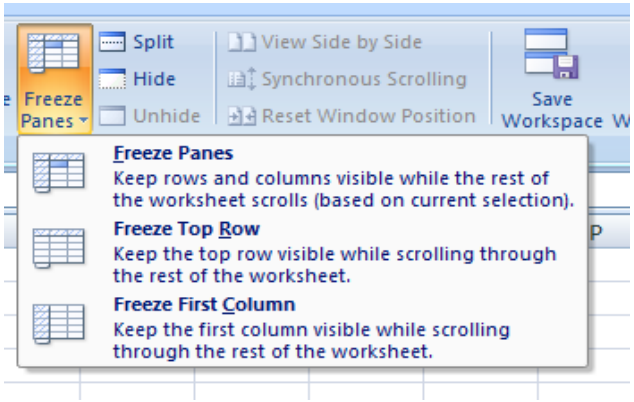
- To manually adjust the width of a column, click and drag the boundary between two column headings.
- To automatically adjust the width, select the required columns and then in the Cell group on the Home ribbon, ***select Format, Cell Size, Autofit Column Width***.
- To manually adjust the height of a row, click and drag the boundary between two row labels.
- To automatically adjust the height, select the required rows and then in the Cell group on the Home ribbon***, select Format, Cell Size, Autofit Row Height***

1.4.3 Hiding rows and columns

- If your spreadsheet contains sensitive data that you don't want displayed on the screen or included in printouts, then you can hide the corresponding rows or columns. The cell values can still be used for calculations, but will be hidden from view.
- Select the row or column heading, right-click to view the pop-up menu, and then select ***Hide*** or ***Unhide***.

1.4.4 Keeping row and column headings in view

- If you scroll through a lot of data in worksheet, you'll probably lose sight of the column headings as they dissappear off the top of your "page". This can make life really difficult. You can use Freeze Panes feature to specify particular rows and columns that will always remain visible as you scroll through the worksheet.
- Select a cell immediately below the row that you want to remain visible, and immediately to the right of the columns that you want to remain visible. On the View tab, click Freezes Panes, and select the first option.



1.5 Formulas

1.5.1 Creating a formula

- A formula always starts with an equal sign ("="). This lets Excel know it's going to have to work something out.
- In the body of the formula, you're going to tell Excell what you want it to calculate. You can use all standard math operations, like addition and multiplication, and you can include numbers, cell references, or built in functions.

1.5.2 How formula are evaluated

- The operators that you need to know are
  - \+ addition
  - \- subtraction
  - * multiplication
  - / division

- The operations are evaluated in a particular order of precedence by Excel :
  - Operations inside brackets are calculated first.
  - Exponentiation is calculated second.
  - Multiplication and division are calculated third.
  - Addition and subtraction are calculated fourth.
  - When you have several items at the same level of precedence, they are calculated from left to right.

- Let's look at some examples :

=10 + 5 * 3 – 7 (result : 10 + 15 - 7 = 18)
=(10 + 5) * 3 – 7 (result : 15*3 - 7 = 38)
=(10 + 5) * (3 – 7) (result : 15*-4 = -60)

If you're not sure how a formula will be evaluated – sed brackets !

1.5.3 Relative cell referencing

- Relative addressing is what Excel uses by default. This means that when you copy or move formula to a new location in a worksheet (or even to another workbook), Excel automatically adjusts the cell references in the copied formula to be consistent with the original formula. If the original formula referenced a value five columns to the left and two rows down, then the copied formula will do the same.

	A	B	C	D
1				
2				
3	Name	Test 1	Test 2	Total
4	Hakimi	25	50	=B4+C4
5	Danish	45	35	=B5+C5
6	Airin	10	45	=B6+C6

1.5.4 Absolute cell referencing

- Relative addressing makes with formulas really easy, but sometimes you don't want the referncing in the formula to change as the formula is copied. Then you need to use absolute addressing. Absolute addressing fixes a cell reference so that regardless of where the formula is copied to, it will always reference the same original cell.

	A	B	C	D
1	Bonus marks :	10		
2				
3	Name	Test 1	Test 2	Total
4	Hakimi	25	50	=B4+C4+$B$1
5	Danish	45	35	=B5+C5+$B$1
6	Airin	10	45	=B6+C6+$B$1

1.6 Functions

1.6.1 Using AutoSum

- Because addition is the most frequently used Excel function, a shortcut has been provided to quickly add a set of numbers :
  1. Select the cell where you want thetotal to appear.
  2. Click on the Sum button on the Home ribbon.
  3. Check that the correct set of numbers has been selected (indicated by a dotted line). If not, then drag to slect a different set of numbers.
  4. Press [ENTER] and the total will be calculated.

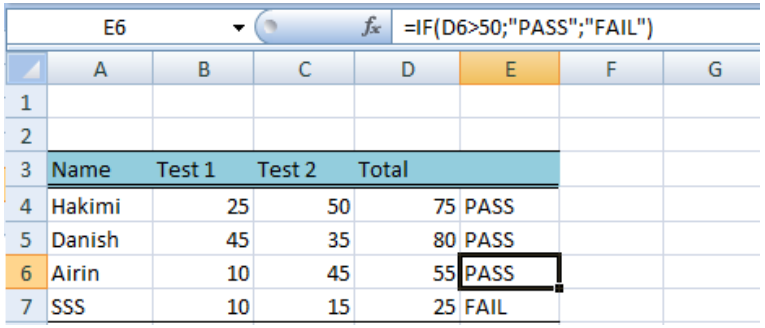
1.6.2 Basic functions

- Some of the most commonly used functions include :

SUM()	to calculate the total set of a set of numbers
AVERAGE()	to calculate the average of a set of numbers
MAX()	to calculate the maximum value within a set of numbers
MIN()	to calculate the minimum value within a set of numbers
ROUND()	to round a set a values to specified number of decimal places
TODAY()	to show the current date
IF()	to calculate a result depending on one or more conditions

1.6.3 The IF() function

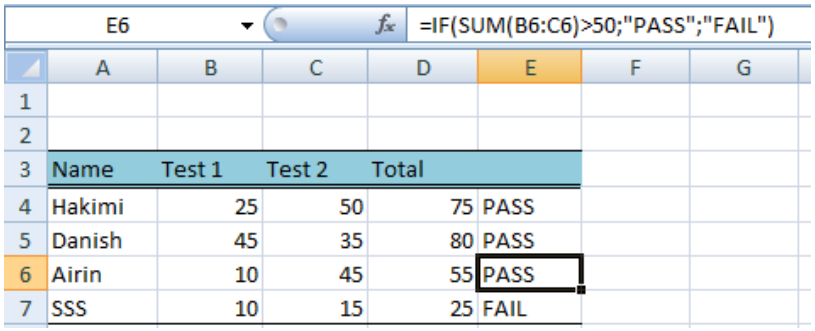
- The IF() function checks for a specific condition. If the condition is met, then one action is taken; if the condition is not met, then a different action is taken.



	A	B	C	D	E	F	G
1							
2							
3	Name	Test 1	Test 2	Total			
4	Hakimi	25	50	75	PASS		
5	Danish	45	35	80	PASS		
6	Airin	10	45	55	PASS		
7	SSS	10	15	25	FAIL		

1.6.4 Nested functions

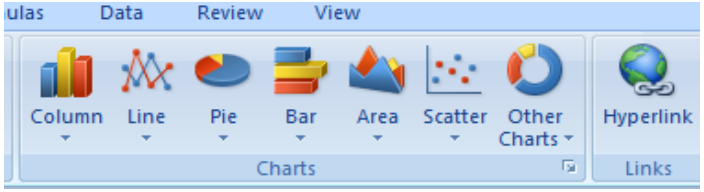
- In this IF statement, we've nested one function inside another. The reference to cell D6 has been replaced with a function that calculates the total test mark, and then check it against the same condition as in 6.3 (">50"), with the same possible outcomes. Doing it this way, you wouldn't ned column D in the worksheet at all.



	A	B	C	D	E	F	G
1							
2							
3	Name	Test 1	Test 2	Total			
4	Hakimi	25	50	75	PASS		
5	Danish	45	35	80	PASS		
6	Airin	10	45	55	PASS		
7	SSS	10	15	25	FAIL		

1.7 Charts

Charts allow you to present information contained in the worksheet in a graphic format. Excel offers many types of charts including: Column, Line, Pie, Bar, Area, Scatter and more. To view the charts available click the Insert Tab on the Ribbon.



1.7.1 Create a chart

- To create a chart :
  1. Select the ***cells*** that contain the data you want to use in the chart.
  2. Click the ***Insert*** tab on the Ribbon.
  3. Click the type of ***Chart*** you want to create.

1.7.2 Modify a chart

- To move the chart:
  1. Click the ***Chart*** and ***Drag*** it another location on the same worksheet, or

2. Click the ***Move Chart*** button on the ***Design*** tab
3. Choose the desired location (either a new sheet or a current sheet in the workbook)

- To move the chart:To change the data included in the chart:
  1. Click the ***Chart*.**
  2. Click the ***Select Data*** button on the ***Design*** tab.
- To modify the labels and titles
  1. Click the ***Chart*.**
  2. On the ***Layout*** tab, click the ***Chart Title*** or the ***Data Labels*** button
  3. Change the ***Title*** and click ***Enter***

1.7.3 Copy a chart to Word

1. Select the ***chart*.**
2. Click ***Copy*** on the ***Home*** tab
3. Go to the ***Word*** document where you want the chart located
4. Click ***Paste*** on the ***Home*** tab

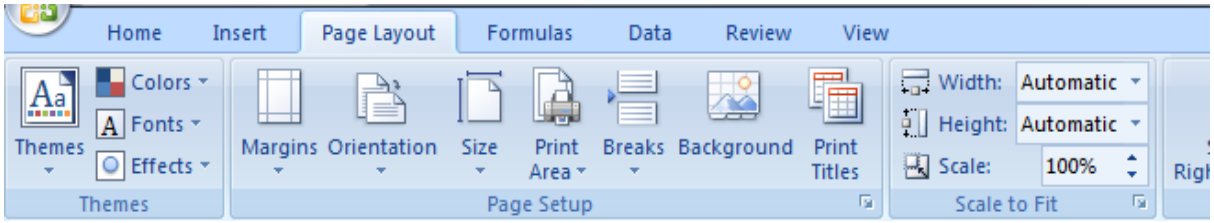
1.8 Printing

1.8.1 Print preview

- Start by using Print Preview to see what your data will look like when it's printed. Click the Office button, select ***Print*** and then ***Print Preview***.

1.8.2 Preparing to print

- Your best option is to use the Page Layout ribbon for this.



1.8.3 Printing a worksheet

- Click the Office button and select the ***Print*** command.

CHAPTER 2 : MATRIX OPERATIONS

2.1 Introduction

2.1.1 Array formulas

- Array formulas refer to evaluations whose results are placed in a range of cells (instead of a single cell) and are invoked by **[CTRL Shift ENTER]**. A group of brackets will automatically enclose the formula to remind the user that it is an array formula.

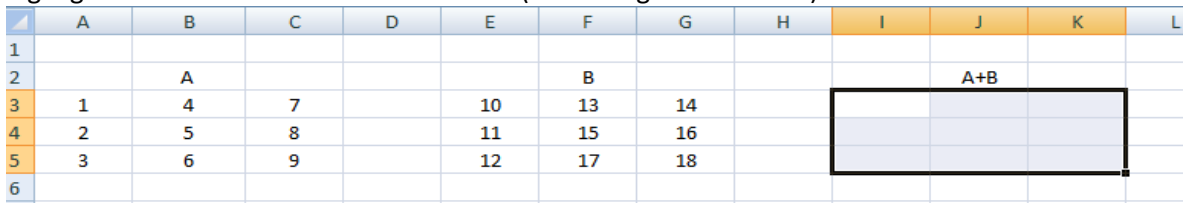
2.1.2 Basic operations

- Some Excel functions perform matrix operations such as :
  1. Addition
  2. Subtraction
  3. Multiplication
  4. Transpose
  5. Determinant
  6. Inverse

2.2 Addition

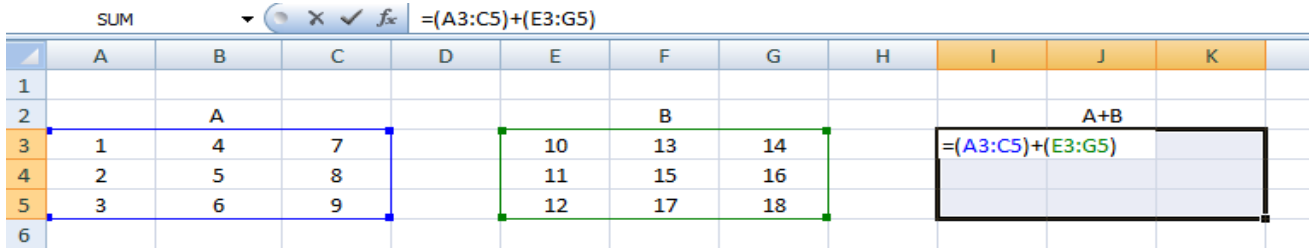
To add matrices :

1. Enter the data of each matrix.
2. Highlight another section of the worksheet (near the given matrices) that has the same dimensions as the answer matrix.



	A	B	C	D	E	F	G	H	I	J	K	L
1												
2		A				B				A+B		
3	1	4	7		10	13	14					
4	2	5	8		11	15	16					
5	3	6	9		12	17	18					
6												

3. Type: **= (A2:C4)+(E2:G4)**



SUM =(A3:C5)+(E3:G5)

	A	B	C	D	E	F	G	H	I	J	K
1											
2		A				B				A+B	
3	1	4	7		10	13	14		=(A3:C5)+(E3:G5)		
4	2	5	8		11	15	16				
5	3	6	9		12	17	18				
6											

4. Since this answer will result in an array (matrix), you will need to: ***CTRL+SHIFT+ENTER.***

2.3 Subtraction

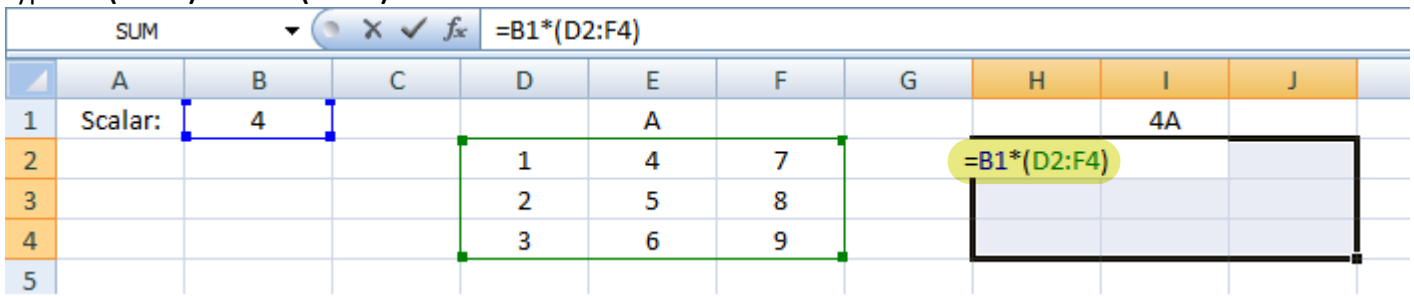
To subtract matrices :

Use the same method in addition, just change + into -.

2.4 Multiplication

To multiply a matrix by a scalar :

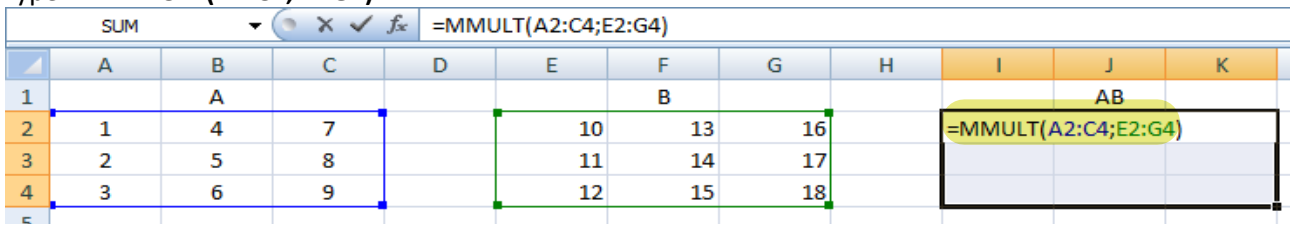
1. Enter the elements of the given matrix.
2. Highlight another section of the worksheet (near the given matrix) that has the same dimensions as the answer matrix.
3. Type: **= (A3:C5)*F2 or (A3:C5)*4**



4. Since this answer will result in an array (matrix), you will need to: ***CTRL+SHIFT+ENTER.***

To multiply two matrices :

1. Enter the elements of the given matrix.
2. Highlight another section of the worksheet (near the given matrix) that has the same dimensions as the answer matrix.
3. Type: **= MMULT(A2:C4;E2:G4)**

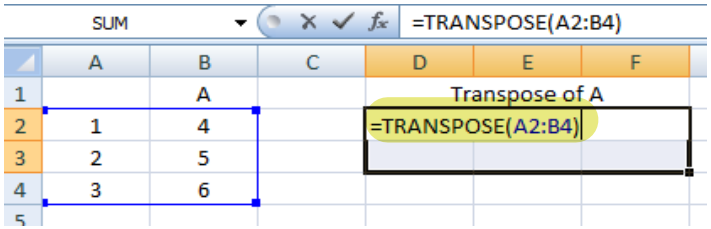


4. Since this answer will result in an array (matrix), you will need to: ***CTRL+SHIFT+ENTER.***

2.5 Transpose

To find the transpose of a matrix :

1. Enter the elements of the given matrix.
2. Highlight another section of the worksheet (near the given matrix) that has the same dimensions as the answer matrix.
3. For example, if matrix A is a 2 x 3 matrix, the transpose of matrix A will be a 3 x 2 matrix.

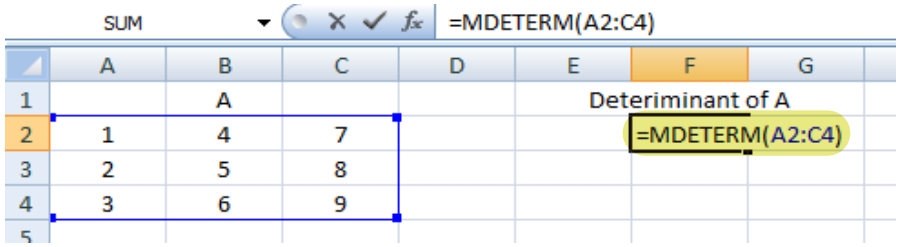


4. Type: **= TRANSPOSE(A2:B4).**
5. Since this answer will result in an array (matrix), you will need to: ***CTRL+SHIFT+ENTER.***

2.6 Determinant

To find the determinant of a matrix :

1. Enter the given square matrix.
2. Highlight a cell of the worksheet (near the given matrix) where you wish the answer to appear.

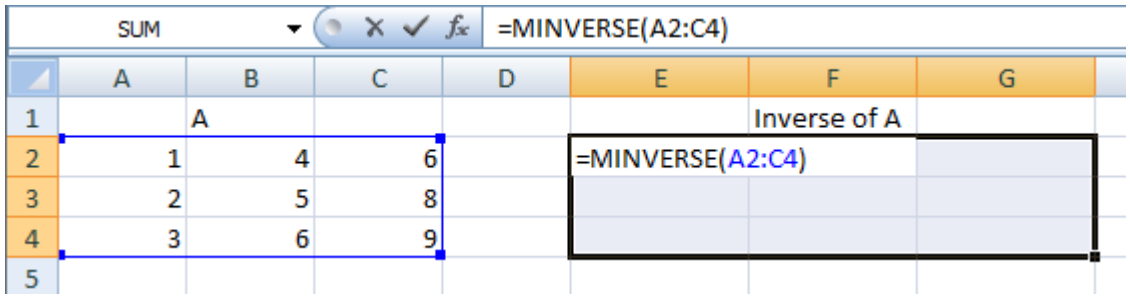


3. Type: = MDETERM(A2:C4)
4. Since this answer will NOT result in an array (matrix) of more than one answer, you may just press ENTER to get the result. (However, if you forget – and do ***CTRL+SHIFT+ENTER***, you will get the same result.)

2.7 Inverse

To find the inverse of a matrix :

1. Enter the square matrix that is to be inverted.
2. Highlight another section of the worksheet (near the given matrix) that has the same dimensions as the given square matrix.



3. Type: **= MINVERSE(A2:C4).**
4. Since this answer will result in an array (matrix), you will need to: ***CTRL+SHIFT+ENTER***.

CHAPTER 3 : SYSTEM OF LINEAR EQUATIONS

3.1 Inverse method

Suppose you have the following equations

$$\begin{aligned} 3x_1 + 5x_2 + 7x_3 &= 10 \\ x_1 + x_2 + 4x_3 &= 8 \\ -6x_1 + x_3 &= 3 \end{aligned}$$

- check determinant $\neq 0$ (no solution, infinite solution)
- extract A.

First, we need to formulate into a matrix equation

$$AX = B$$

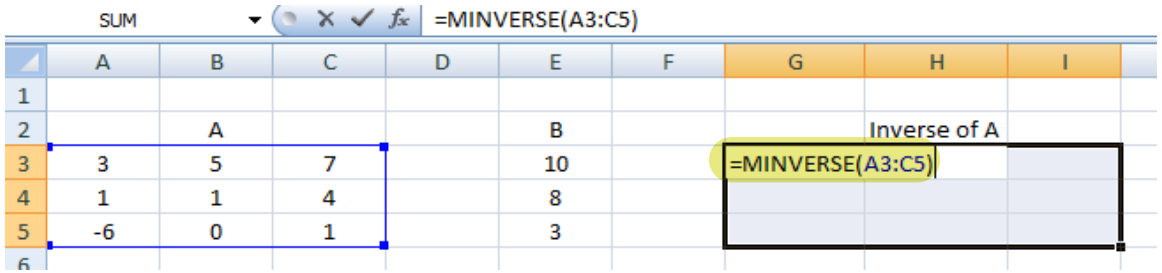
where,

$$A = \begin{pmatrix} 3 & 5 & 7 \\ 1 & 1 & 4 \\ -6 & 0 & 1 \end{pmatrix}; \quad X = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}; \quad B = \begin{pmatrix} 10 \\ 8 \\ 3 \end{pmatrix}$$

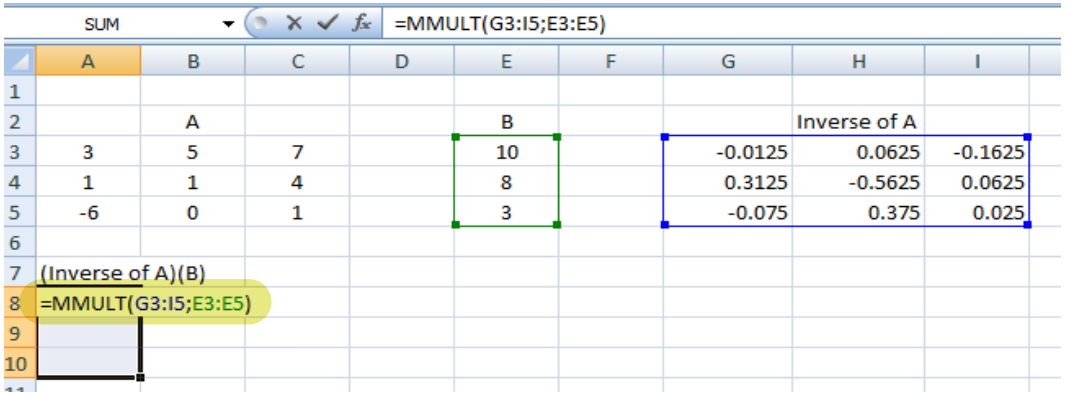
and the solution is then given by

$$X = A^{-1}B$$

- To solve the system of linear equations using Excel :
  1. Enter the element of matrix A and B.
  2. Find the inverse of matrix A.

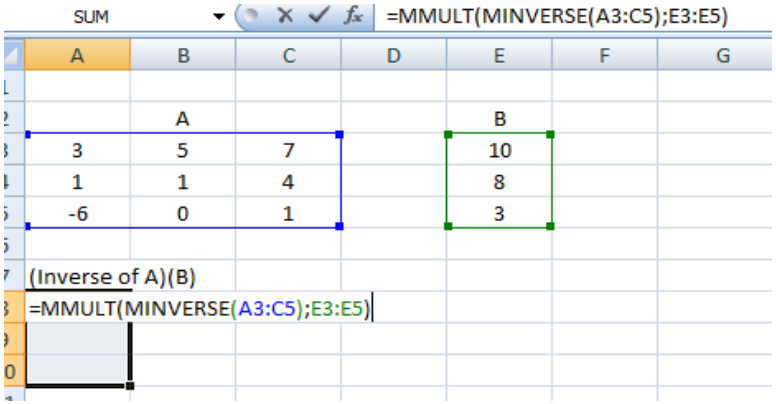


  3. Type : =MINVERSE(A3:C5) and press ***CTRL+SHIFT+ENTER***.
  4. Find the multiplication between inverse of A and B:
     i. Highlight another section of the worksheet (near the given matrix) that has the same dimensions as the answer matrix. (3 x 1 matrix)



ii. Type: **= MMULT(G3;I5;E3:5)** and press ***CTRL+SHIFT+ENTER.***

5. Or, you can just type: **= MMULT(MINVERSE(A3:C5);E3:E5)** and press ***CTRL+SHIFT+ENTER.***



3.2 Cramer's Rules

- Cramer's rule is a another method for solving system of linear equations. It makes use of determinants.
- If we are given a pair of simultaneous equations

$$a_1x + b_1y = d_1$$
$$a_2x + b_2y = d_2$$

then x and y can be found from

$$x = \frac{\begin{vmatrix} \boldsymbol{d_1} & b_1 \\ \boldsymbol{d_2} & b_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}}, \qquad y = \frac{\begin{vmatrix} a_1 & \boldsymbol{d_1} \\ a_2 & \boldsymbol{d_2} \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}}.$$

it can be represented by

$$x = \frac{|A_1|}{|A|}, \qquad y = \frac{|A_2|}{|A|}.$$

- Suppose you have the following equations

$$3x + 4y = -14 \quad (1)$$
$$-2x - 3y = 11 \quad (2)$$

To solve the system of linear equations using Excel :

1. Enter matrix A, A_1 and A_2 in array **B1:C2, B4:C5** and **B7:C8.**
2. To find $|A|$, $|A_1|$ and $|A_2|$: type **=MDETERM(B1:C2), =MDETERM(B4:C5)** and **=MDETERM(B7:C8)** in cell **B10**, **B12** and **B14** respectively.
3. To find the solution of x and y : type **=B12/B10** and **=B14/B10** in cell **B16** and **B17** respectively.

	A	B	C
1	A=	3	4
2		-2	-3
3			
4	A_1=	14	4
5		11	3
6			
7	A_2=	3	-14
8		-2	11
9			
10	|A|=	=MDETERM(B1:C2)	
11			
12	|A_1|=	=MDETERM(B4:C5)	
13			
14	|A_2|=	=MDETERM(B7:C8)	
15			
16	x=	=B12/B10	
17	y=	=B14/B10	

- For the case of three equations in three unknowns:

$$\begin{aligned} a_1x + b_1y + c_1z &= d_1 \\ a_2x + b_2y + c_2z &= d_2 \\ a_3x + b_3y + c_3z &= d_3 \end{aligned}$$

then x and y can be found from

$$x = \frac{\begin{vmatrix} \boldsymbol{d_1} & b_1 & c_1 \\ \boldsymbol{d_2} & b_2 & c_2 \\ \boldsymbol{d_3} & b_3 & c_3 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}}, \quad y = \frac{\begin{vmatrix} a_1 & \boldsymbol{d_1} & c_1 \\ a_2 & \boldsymbol{d_2} & c_2 \\ a_3 & \boldsymbol{d_3} & c_3 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}}, \quad z = \frac{\begin{vmatrix} a_1 & b_1 & \boldsymbol{d_1} \\ a_2 & b_2 & \boldsymbol{d_2} \\ a_3 & b_3 & \boldsymbol{d_3} \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}}.$$

it can be represented by

$$x = \frac{|A_1|}{|A|}, \quad y = \frac{|A_2|}{|A|}, \quad z = \frac{|A_3|}{|A|}.$$

- Let us use the same example as in section 3.1.

- You can follow the same step in 2x2 case:

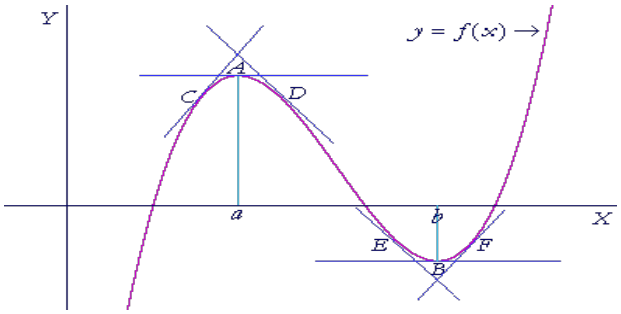
	A	B	C	D
1		3	5	7
2	A=	1	1	4
3		-6	0	1
4				
5		10	5	7
6	A_1=	8	1	4
7		3	0	1
8				
9		3	10	7
10	A_2=	1	8	4
11		-6	3	1
12				
13		3	5	10
14	A_3=	1	1	8
15		-6	0	3
16				
17	|A|=	=MDETERM(B1:D3)		
18				
19	|A_1|=	=MDETERM(B5:D7)		
20				
21	|A_2|=	=MDETERM(B9:D11)		
22				
23	|A_3|=	=MDETERM(B13:D15)		
24				
25	x=	=B19/B17		
26	y=	=B21/B17		
27	z=	=B23/B17		

Any questions?

CHAPTER 4 : EXCEL SOLVER

4.1 Maximum or minimum points

- The value of a function $f(x)$, at either a maximum or a minimum is called an extreme value.
- What characterizes the graph at an extreme value?
  → the tangent to the curve is horizontal (we see this at the points A and B). The slope of each tangent line is the derivative when evaluated at a or b is 0 $[f'(x) = 0]$.



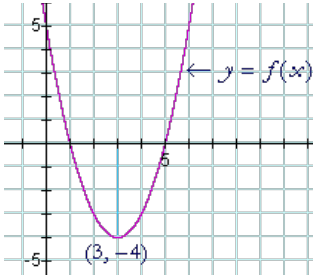
- Moreover, at points immediately to the left of a maximum (at point C), the slope of the tangent is positive, $[f'(x) > 0]$.
- While at points immediately to the right of a maximum (at point D), the slope of the tangent is negative, $[f'(x) < 0]$.
- In other words, at a maximum, $f'(x)$ changes sign from + to – (see points C and D) while at a minimum, $f'(x)$ changes sign from – to + (see points E and F).
- We can also observe that at a maximum, at A, the graph is concave downward, while at a minimum, at B, it is concave upward.
- A value of x at which the function has either a maximum or a minimum is called a critical value. In this figure, the critical values are $x = a$ and $x = b$.
- The critical values determine turning points, at which the tangent is parallel to the x-axis.
- The critical values, if any, will be the solution to the equation $f'(x) = 0$.
- For example :

  Let $f(x) = x^2 - 6x + 5$, *Are there any critical values? If so, do they determine a maximum or a minimum?.*

  Solution :

$$f'(x) = 2x - 6$$
$$=> x = 3$$
$$f(3) = 3^2 - 6(3) + 5 = -4$$

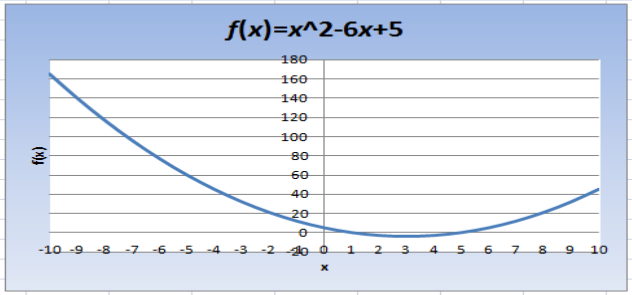
The extreme value is -4. To see whether it is a maximum or a minimum, in this case we can simply look at the graph.



$f(x)$ is a parabola, and we can see that the turning point is a minimum.
By finding the value of x, where the derivative is 0, then we have discovered that the vertex of the parabola is at (3,-4).

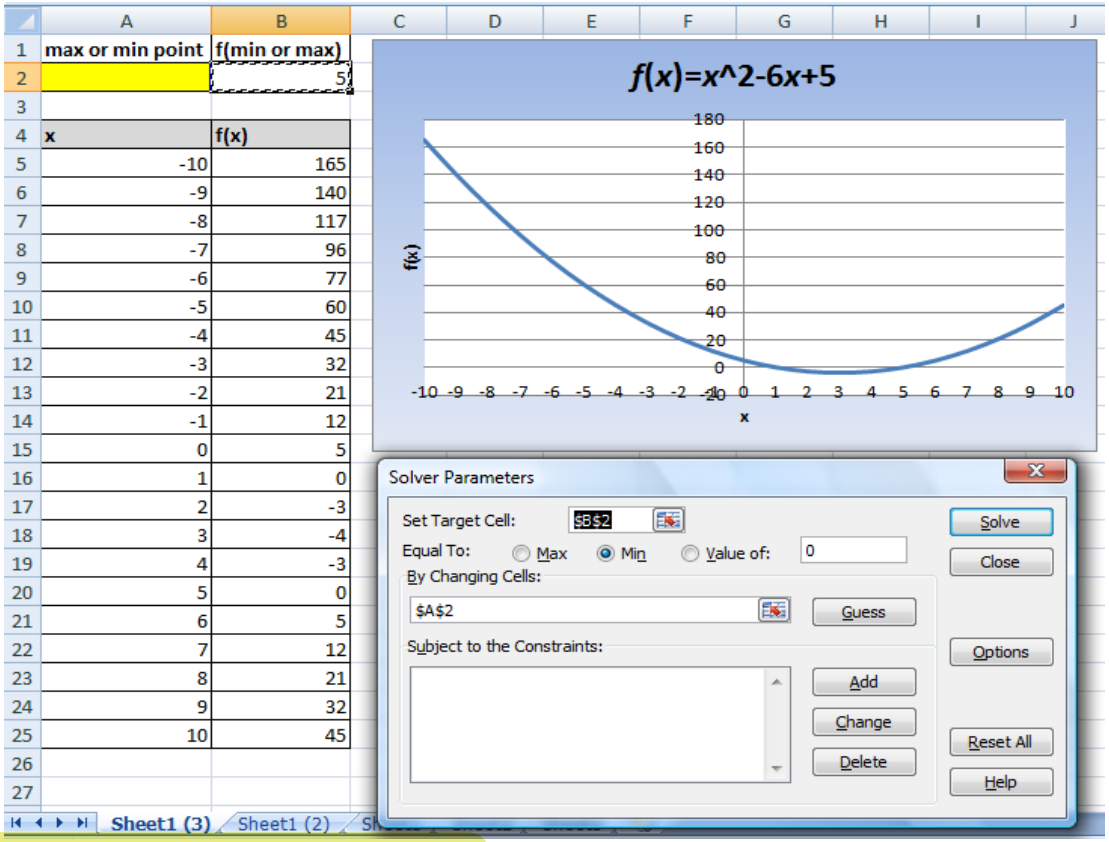
- To find the maximum and minimum point using Excel :
  1. You can start by plotting the graph so that you can see whether the turning point is minimum or maximum.

	A	B
1	max or min point	f(min or max)
2		
3		
4	x	f(x)
5	-10	165
6	-9	140
7	-8	117
8	-7	96
9	-6	77
10	-5	60
11	-4	45
12	-3	32
13	-2	21
14	-1	12
15	0	5
16	1	0
17	2	-3
18	3	-4
19	4	-3
20	5	0
21	6	5
22	7	12
23	8	21
24	9	32
25	10	45
26		

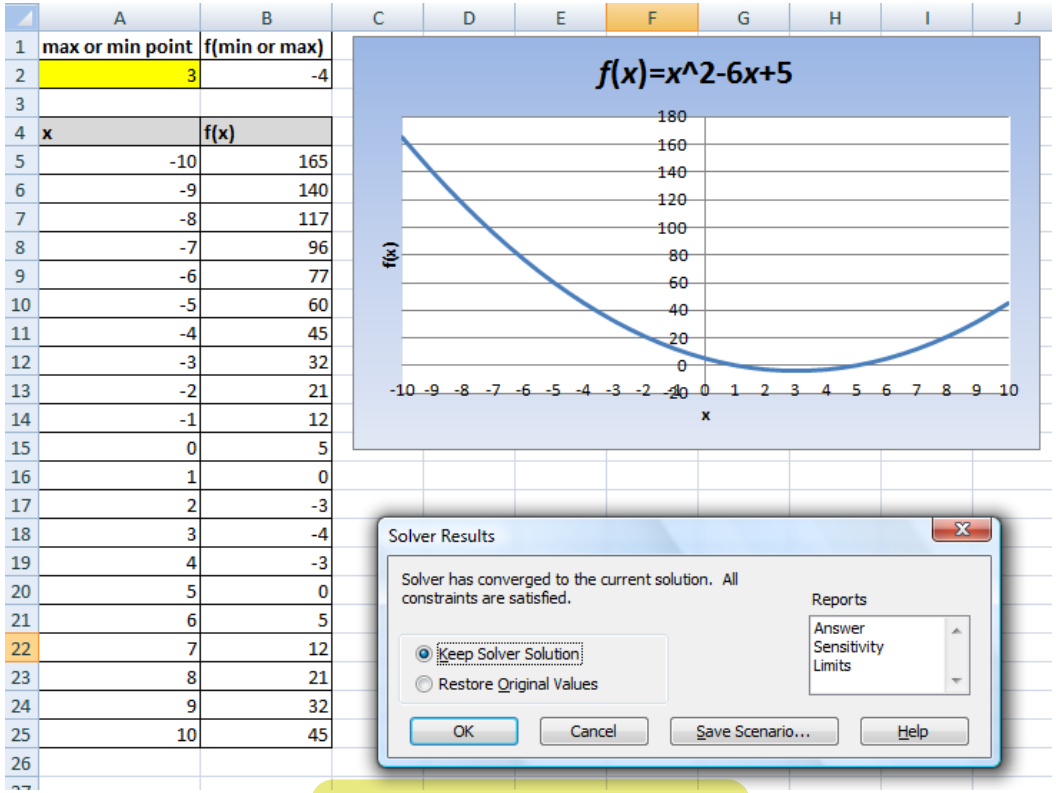


  2. Set cell A2 as the maximum or minimum point which will be given by Solver.
  3. Type the function in cell B2 : **A2^2 – 6*A2 + 5** .

4. Now, in the **Data** tab, click **Solver**.



5. Set the **Target Cell** for the $f\,(minimum)$ value i.e., **$B$2**.
6. Choose the **Equal To** : Min.
7. In the **By Changing Cell** option, select the solution values of the variables x and y i.e., **$A$2**.
8. Now, click the **Solve** button.



	A	B	C	D	E	F	G	H	I	J
1	max or min point	f(min or max)								
2	3	-4								
3										
4	x	f(x)								
5	-10	165								
6	-9	140								
7	-8	117								
8	-7	96								
9	-6	77								
10	-5	60								
11	-4	45								
12	-3	32								
13	-2	21								
14	-1	12								
15	0	5								
16	1	0								
17	2	-3								
18	3	-4								
19	4	-3								
20	5	0								
21	6	5								
22	7	12								
23	8	21								
24	9	32								
25	10	45								
26										

9. The default option has a **Keep Solver Solution** and click on the **Answer** in the **Reports** section on the right hand side. Finally, click the **OK** button to get the results.

	A	B
1	max or min point	f(min or max)
2	3	-4

10. Finally the Excel Solver gives the solution values for variables $minimum\ point$ and the value of the $f(minimum\ point)$.

Not in the final.

4.2 Linear programming

- Linear programming is the most popular of the approaches falling under the general heading of mathematical optimization techniques.
- For example :

An oil refinery produces two products: jet fuel and gasoline. The profit for the refinery is \$0.10 per barrel for jet fuel and \$0.20 per barrel for gasoline. The following conditions must be met.

1. *Only 10,000 barrels of crude oil are available for processing.*
2. *The refinery has a government contract to produce at least 1,000 barrels of jet fuel.*
3. *The refinery has a private contract to produce at least 2,000 barrels of gasoline.*
4. *Both products are shipped in trucks, the delivery capacity of the truck fleet is 180,000 barrel-miles.*
5. *The jet fuel is delivered to an airfield 10 miles from the refinery.*
6. *The gasoline is transported 30 miles to the distributor.*

How much of each product should be produced for maximum profit?

Let x represent the number of barrels of jet fuel,
y represent the number of barrels of gasoline.

Our objective function is : $Z(x, y) = 0.1x + 0.2y$

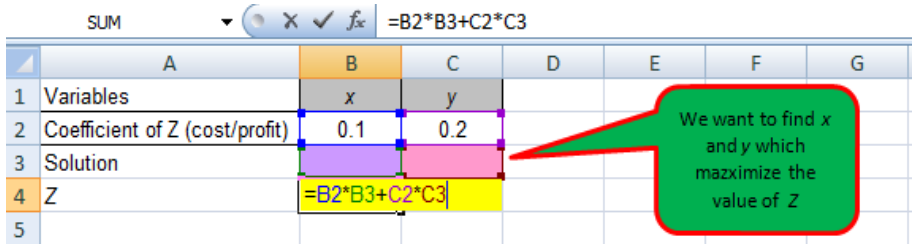
Subject to constraint :

1. $x + y \leq 10000$
2. $x \geq 1000$
3. $y \geq 2000$
3. $10x + 30y \leq 180000$

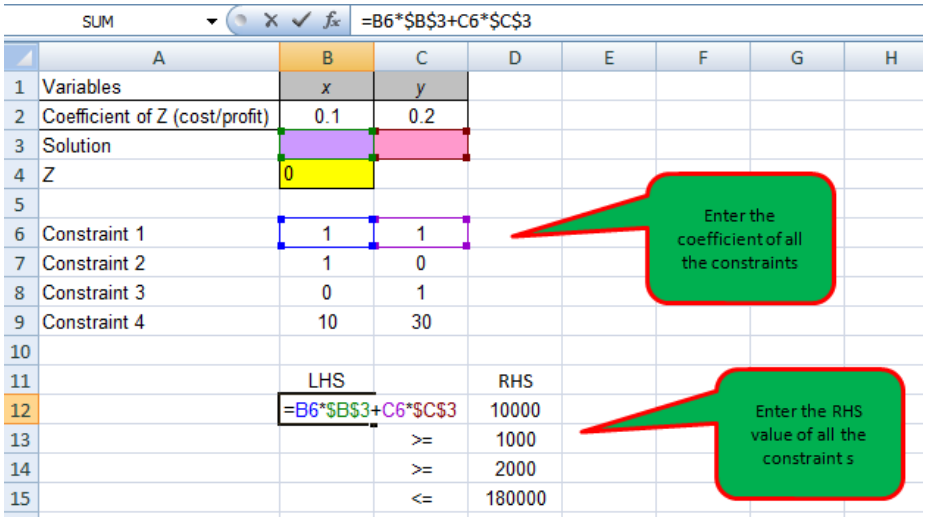
- Our focus here is on how we can quickly set up and solve these problems using the Solver that comes with Microsoft Excel.

1. Enter the coefficients of the objective function Z i.e., **(0.1, 0.2)** in cells B2 and C2.

   This formula uses the coefficient values and also the solution values for variable x and y, which aresupposed to be solved. At the moment, if you enter, the value of Z will be 0.

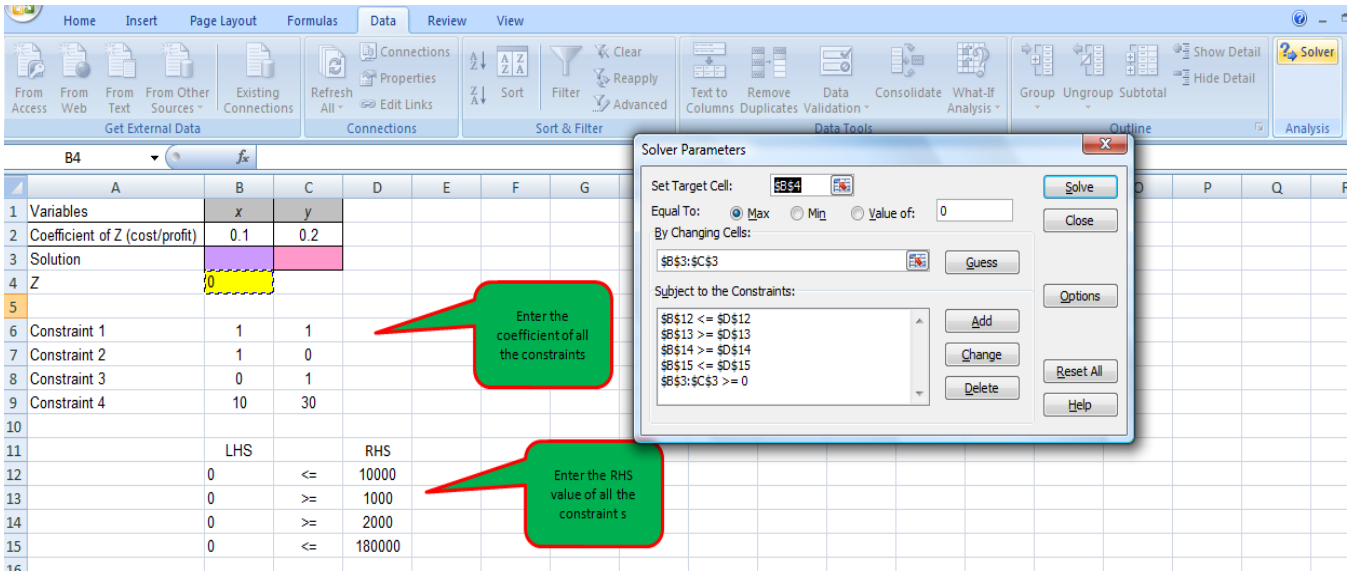


2. Enter the coefficients of the Constraint 1 i.e., **(1, 1)** in cells B6 and C6.
3. Enter the coefficients of the Constraint 2 i.e., **(1, 0)** in cells B7 and C7.
4. Enter the coefficients of the Constraint 3 i.e., **(0, 1)** in cells B8 and C8.
5. Enter the coefficients of the Constraint 4 i.e., **(10, 30)** in cells B9 and C9.
6. Enter the formula for LHS of the Constraint 1, 2, 3 & 4 i.e., type **=B6*$B$3 + C6*$C$3** in cell B12 and then copy this formula until B15.



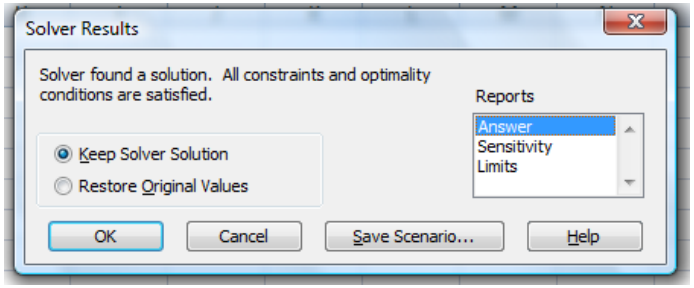
	A	B	C	D
1	Variables	*x*	*y*	
2	Coefficient of Z (cost/profit)	0.1	0.2	
3	Solution			
4	Z	0		
5				
6	Constraint 1	1	1	
7	Constraint 2	1	0	
8	Constraint 3	0	1	
9	Constraint 4	10	30	
10				
11		LHS		RHS
12		=B6*$B$3+C6*$C$3		10000
13			>=	1000
14			>=	2000
15			<=	180000

7. Now, in the **Data** tab, click **Solver**.



	A	B	C	D
1	Variables	*x*	*y*	
2	Coefficient of Z (cost/profit)	0.1	0.2	
3	Solution			
4	Z	0		
5				
6	Constraint 1	1	1	
7	Constraint 2	1	0	
8	Constraint 3	0	1	
9	Constraint 4	10	30	
10				
11		LHS		RHS
12		0	<=	10000
13		0	>=	1000
14		0	>=	2000
15		0	<=	180000

8. Set the **Target Cell** for the Objective Function Z value i.e., **$B$4**.
9. Choose the **Equal To** : Min.
10. For Changing Cell, select the solution values of the variables x and y i.e., **$B$3:$C$3**.
11. For **Subject to the Constraint**, click on the Add option and select **$B$12 <=$D$12**, **$B$13 <=$D$13**, **$B$14 <=$D$14** and **$B$15 <=$D$15**.
12. Also all the values need to be positive, select **SB$3:C$3 >=0.**
13. Now, click the **Solve** button.



14. The default option has a **Keep Solver Solution** and click on the **Answer** in the **Reports** section on the right hand side. Finally, click the **OK** button to get the results.

	A	B	C	D
1	Variables	*x*	*y*	
2	Coefficient of Z (cost/profit)	0.1	0.2	
3	Solution	6000	4000	
4	*Z*	1400		
5				
6	Constraint 1	1	1	
7	Constraint 2	1	0	
8	Constraint 3	0	1	
9	Constraint 4	10	30	
10				
11		LHS		RHS
12		10000	<=	10000
13		6000	>=	1000
14		4000	>=	2000
15		180000	<=	180000
16				

15. Finally the Excel Solver gives the solution values for variables x and y and the value of the objective function Z.

Any questions?

CHAPTER 5 : FINDING THE ROOTS OF EQUATIONS

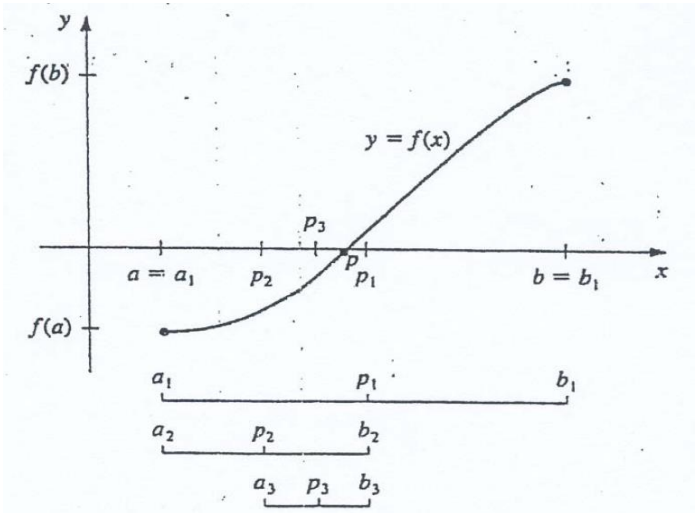
Root-finding problem

- Root-finding is one of the basic problems in numerical approximation.
- It involves finding a root x of an equation of the form $f(x) = 0$, for a given function f.
- The number x is also called a zero of f.
- We will consider two methods :
    1. Bisection method
    2. Newton method

5.1 Bisection method

- The first technique based on the Intermediate Value Theorem.

- Suppose f is a continuous function defined on the interval $[a, b]$, with $f(a)$ and $f(b)$of opposite sign. By the Intermediate Value Theorem, there exists a number p in (a, b) with $f(p) = 0$. Although the procedure will work for the case when $f(a)$ and $f(b)$ have opposite signs and there is more than one root in the interval (a, b), we assume for simplicity that the root in this interval is unique. The method calls for a repeated halving of subintervals of $[a, b]$ and, at each step, locating the half containing p.



- This is the steps in bisection method :
  1. Set $a_1 = a$ and $b_1 = b$, and let p_1 be the midpoint of $[a, b]$; that is
  $$p_1 = \frac{1}{2}(a_1 + b_1).$$
  2. If $f(p_1) = 0$, then $p = p_1$. If not, then $f(p_1)$ has the same sign as either $f(a_1)$ or $f(b_1)$.
  3. If $f(p_1)$ and $f(a_1)$ have the same signs, then $p \in (p_1, b_1)$, and we set $a_2 = p_1$ and $b_2 = b_1$. If $f(p_1)$ and $f(a_1)$ have opposite signs, then $p \in (a_1, p_1)$, and we set $a_2 = a_1$ and $b_2 = p_1$.
  4. We then reapply the process to the interval $[a_2, b_2]$.

$f(p_1)$ & $f(a_1)$ same sign
$\downarrow$ $p \in (p_1, b_1)$

- Our focus here is to find the roots using Microsoft Excel.

- For example, we want to find the roots for $f(x) = x^2 - 2$ in $[-2, -1]$.
  1. Type the heading for the value of $a, p_1, b, f(a), f(p_1)$ and $f(b)$in cells **A1**, **B1**, **C1**, **D1**, **E1**and **F1**.

	A	B	C	D	E	F
1	a	midpoint, p1	b	f(a)	f(midpoint)=f(p1)	f(b)
2						
3						
4						

  2. Enter the endpoints of the interval into cells **A2** and **C2**, i.e. (-2,-1).

  3. Type the formula for midpoint, **=(A2+C2)/2** in cell **B2**. Copy this formula to cell **B3**.

	A	B	C	D	E	F
1	a	midpoint, p1	b	f(a)	f(midpoint)=f(p1)	f(b)
2	-2	=(A2+C2)/2	-1			
3						

  4. Type the formula of the function $f(x) = x^2 - 2$, i.e**.,** **= A2^2 -2** in cell D2.

	A	B	C	D	E	F
1	**a**	**midpoint, p1**	**b**	**f(a)**	**f(midpoint)=f(p1)**	**f(b)**
2	-2	-1.5	-1	=A2^2-2		
3						

5. Copy the formula in **D2** across to cells **E2** and **F2**. Then, copy this three cells (**D2**, **E2** and **F2**) down to **D3** through **F3**. The relative references in cell **D2** cause cells **E2** and **F2** to contain the values of the function at **B2** and **C2** respectively.

6. If the function does not change sign on the left half interval, the new left endpoint is the midpoints, **B2**, otherwise it is the previous left endpoint **A2**. Then, type

   =IF(SIGN(D2)=SIGN(E2),B2, A2)

   into cell **A3** and copy this cell down to cell **A4**.

	A	B	C	D	E	F	G
1	**a**	**midpoint, p1**	**b**	**f(a)**	**f(midpoint)=f(p1)**	**f(b)**	
2	-2	-1.5	-1	2	0.25	-1	
3	=IF(SIGN(D2)=SIGN(E2);B2;A2)			0.25	-1.4375	-2	
4							

7. Similarly, if the function does not change sign on the right half interval, the new right endpoint is the midpoints, **B2**, otherwise it is the previous right endpoint **C2**. Then, type

   =IF(SIGN(E2)=SIGN(F2),B2, C2)

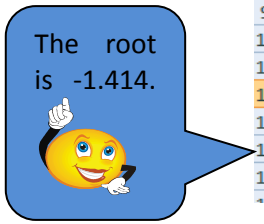
	A	B	C	D	E	F
1	**a**	**midpoint, p1**	**b**	**f(a)**	**f(midpoint)=f(p1)**	**f(b)**
2	-2	-1.5	-1	2	0.25	-1
3	-1.5	-1.5	=IF(SIGN(E2)=SIGN(F2);B2;C2)			0.25
4						

8. If the value of $f(a) = f(p_1 = midpoint) = f(b) = 0$ and the value of $a = p_1(midpoint) = b$ then the root value is $x = a = p_1(midpoint) = b$.

SUGGESTIONS :
Format all the numbers in 3 decimal places,

	A	B	C	D	E	F
1	**a**	**midpoint, p1**	**b**	**f(a)**	**f(midpoint)=f(p1)**	**f(b)**
2	-2.000	-1.500	-1.000	2.000	0.250	-1.000
3	-1.500	-1.250	-1.000	0.250	-0.438	-1.000
4						

and then copy **A3:F3** down until you get the root which satisfy condition in step 8.



	A	B	C	D	E	F
1	**a**	**midpoint, p1**	**b**	**f(a)**	**f(midpoint)=f(p1)**	**f(b)**
2	-2.000	-1.500	-1.000	2.000	0.250	-1.000
3	-1.500	-1.250	-1.000	0.250	-0.438	-1.000
4	-1.500	-1.375	-1.250	0.250	-0.109	-0.438
5	-1.500	-1.438	-1.375	0.250	0.066	-0.109
6	-1.438	-1.406	-1.375	0.066	-0.022	-0.109
7	-1.438	-1.422	-1.406	0.066	0.022	-0.022
8	-1.422	-1.414	-1.406	0.022	0.000	-0.022
9	-1.422	-1.418	-1.414	0.022	0.011	0.000
10	-1.418	-1.416	-1.414	0.011	0.005	0.000
11	-1.416	-1.415	-1.414	0.005	0.002	0.000
12	-1.415	-1.415	-1.414	0.002	0.001	0.000
13	-1.415	-1.414	-1.414	0.001	0.000	0.000
14	-1.414	-1.414	-1.414	0.000	0.000	0.000
15	-1.414	-1.414	-1.414	0.000	0.000	0.000

5.2 Newton's method

- We can also approximate roots of functions using Newton's method.
- We need to assume that f is a differentiable function in some interval $[a, b]$, i.e., we can find f' in the interval $[a, b]$.
- Figure 5.1.2.1 illustrates how the approximations are obtained using successive tangents.
    - Starting with the initial approximation p_0, the approximation $p_1$is the x-intercept of the tangent line to the graph of f at $\big(p_0, f(p_0)\big)$.
    - The approximation p_2 is the x-intercept of the tangent line to the graph of f at $\big(p_1, f(p_1)\big)$ and so on.

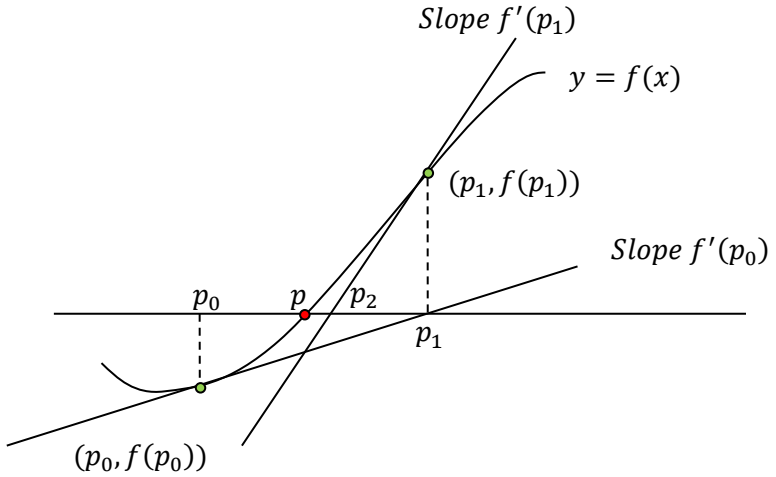


Figure 5.1.2.1

- Now, let us consider this method algebraically.
- Referring back to Figure 5.1.2.1, we see that the slope of the line through points $(p_1, 0)$ and $(p_0, f(p_0))$ is equal to

$$\frac{0 - f(p_0)}{p_1 - p_0}.$$

Moreover, the slope also equals $f'(p_0)$ since this line is tangent to the curve of f. Hence,

$$f'(p_0) = \frac{0 - f(p_0)}{p_1 - p_0}$$

which yields

$$p_1 - p_0 = \frac{0 - f(p_0)}{f'(p_0)}$$

or

$$p_1 = p_0 - \frac{f(p_0)}{f'(p_0)}.$$

- Similarly, the slope of the line through points $(p_2, 0)$ and $(p_1, f(p_1))$ is equal to

$$\frac{0 - f(p_1)}{p_2 - p_1}.$$

Moreover, the slope also equals $f'(p_1)$ since this line is tangent to the curve of f. Hence,

$$f'(p_1) = \frac{0 - f(p_1)}{p_2 - p_1}$$

which yields

$$p_2 - p_1 = \frac{0 - f(p_1)}{f'(p_1)}$$

or

$$p_2 = p_1 - \frac{f(p_1)}{f'(p_1)}.$$

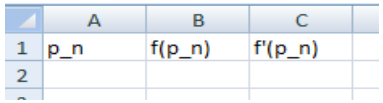
- Iterating this yields the general term

$$p_{n+1} = p_n - \frac{f(p_n)}{f'(p_n)} \; for \; n = 0,1,2, \ldots.$$

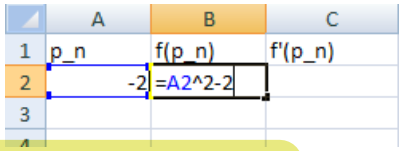
- Newton's method can be easily applied via Excel.
- Let us consider the same example in Bisection method, i.e., we want to find the root for $f(x) = x^2 - 2$ in $[-2, -1]$.

- These are the steps :

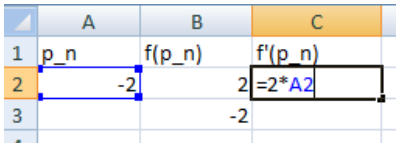
1. Type the heading for the value of $p_n, f(p_n)$ and $f'(p_n)$ in cells **A1**, **B1** and **C1**.



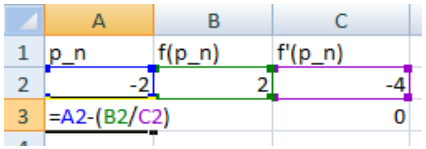
2. Enter the first initial guess, p_0 into cell **A2.**
3. Type the formula of the function $f(x) = x^2 - 2,$ i.e**., = A2^2 -2** in cell **B2**. Copy this formula down to cell **B3.**



4. Type the formula of the function $f'(x) = 2x,$ i.e**., = 2*A2** in cell **C2**. Copy this formula down to cell **C3.**



5. Type the Newton's formula, i.e., =**A2-(B2/C2)** in cell **A3.**

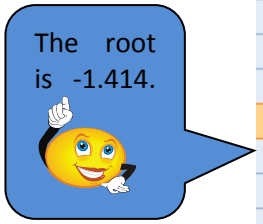


SUGGESTIONS :

Format all the numbers in 3 decimal places,

	A	B	C
1	p_n	f(p_n)	f'(p_n)
2	-2.000	2.000	-4.000
3	-1.500	0.250	-3.000
4			

and then copy **A3:C3** down until you get the root where the value of $f(p_n)$ in column B equal to zero.



	A	B	C
1	p_n	f(p_n)	f'(p_n)
2	-2.000	2.000	-4.000
3	-1.500	0.250	-3.000
4	-1.417	0.007	-2.833
5	-1.414	0.000	-2.828
6	-1.414	0.000	-2.828
7	-1.414	0.000	-2.828

Any questions?

CHAPTER 6 : NUMERICAL INTEGRATION

6.1 Introduction

- An ***Integral*** *is a* mathematical object that can be interpreted as an area or a generalization of area. It can be written as

$$\int_a^b f(x)\,dx$$

where a and b are given and f is a function given analytically by a formula. Geometrically, integral is the area under the curve of f between a and b as shown in Figure 6.1.

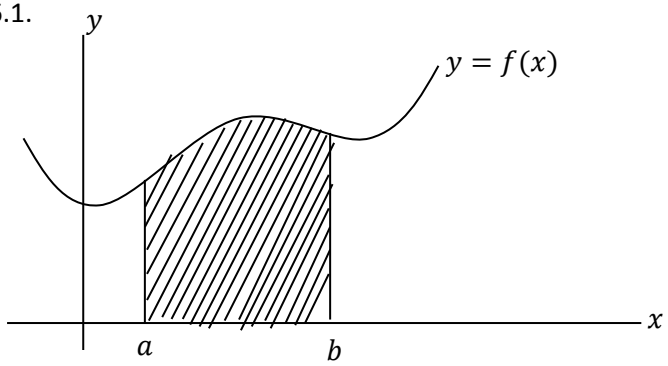


Figure 6.1 : Geometrical interpretation of a definite integral

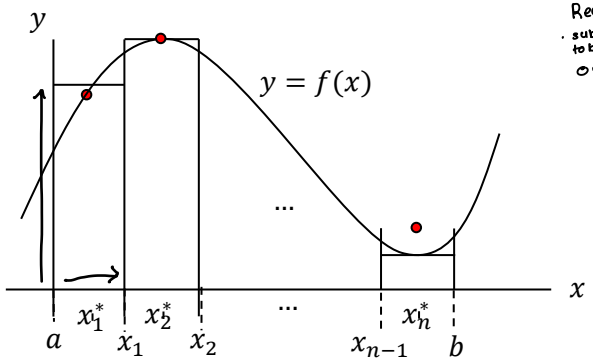
- The process of computing an integral is called ***integration.***
- ***Numerical integration*** is the approximate computation of integral using numerical methods.
- In this course, we will consider three numerical integration methods :
    1. Rectangular rule
    2. Trapezaidal rule
    3. Simpson's rule

6.2 Rectangular Rule

- Numerical integration method are obtained by approximating the integrand f by functions that can easily be integrated.
- The simplest formula, the Rectangular rule, is ontained if we subdivide the interval of integration $a \le x \le b$ into n subintervals of equal length $h = (b-a)/n$ and in each subinterval approximate f by the constant $f(x_j^*)$, the value of f at the midpoint x_j^* of the jth subinterval (Figure 6.2). Then f is approximated by a step **function** (piecewise constant function), the n rectangles in Figure 6.2 have the areas $f(x_1^*)h$, $f(x_2^*)h$, ..., $f(x_n^*)h$, and the Rectangular Rule is

area. $$J = \int_a^b f(x)\,dx \approx f(x_1^*)h + f(x_2^*)h + \cdots + f(x_n^*)h$$
$$\approx h[f(x_1^*) + f(x_2^*) + \cdots + f(x_n^*)]$$

where $h = (b-a)/n$.



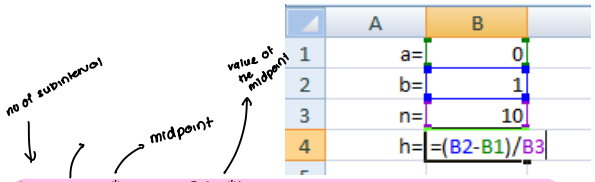
Rectangular Rule:
- subdivide the integral a to b into n subinterval of equal length
$h = \frac{(b-a)}{n}$

Figure 6.2 : Rectangular rule

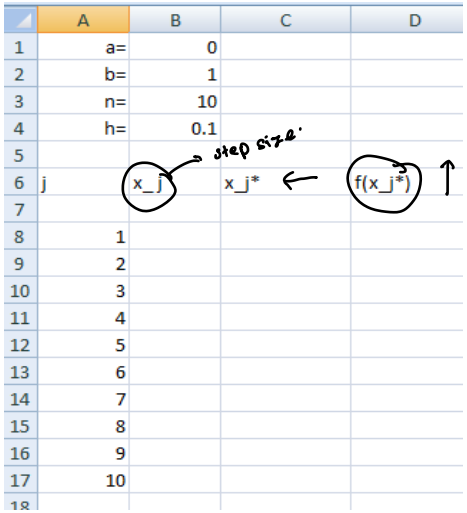
- Example :
  Evaluate $J = \int_0^1 e^{-x^2}\,dx$ *with* $n = 10$.

Bilangan interval

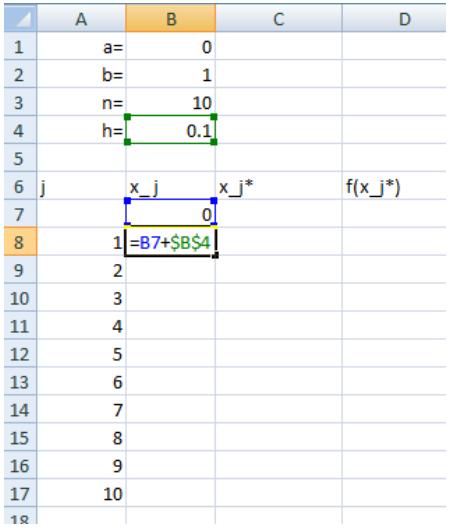
- We will use Excel to calculate the approximation of J using Rectangular rule.
  1. Enter the value of a, b and n in cell **B1**, **B2** and **B3** respectively.
  2. Calculate the value stepsize, h in cell **B4.**



3. Type the heading of j, x_j, x_j^* and $f(x_j^*)$ in cell **A6**, **B6**, **C6** and **D6** respectively.
4. We have 10 subintervals, then type 1 until 10 in cell **A8:A17.**

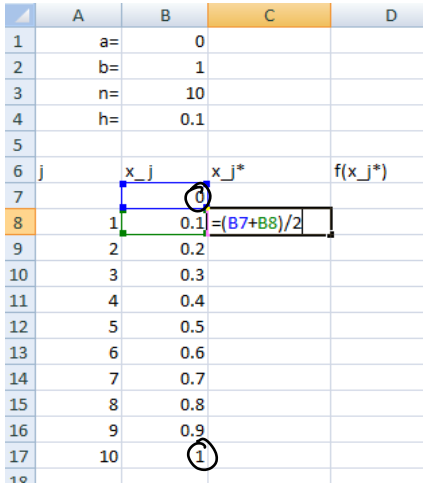


5. Enter the value of $a = 0$ in cell **B7**.
6. Type : **=B7+$B$4** in cell **B8** and copy down this formula until **B17.**



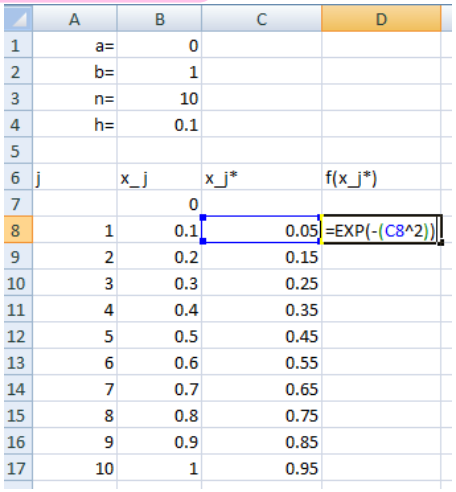
	A	B	C	D
1	a=	0		
2	b=	1		
3	n=	10		
4	h=	0.1		
5				
6	j	x_j	x_j*	f(x_j*)
7		0		
8	1	=B7+$B$4		
9	2			
10	3			
11	4			
12	5			
13	6			
14	7			
15	8			
16	9			
17	10			

7. Type the formula of midpoint : **= (B7+B8)/2** in cell **C8** and copy down until **C17.**



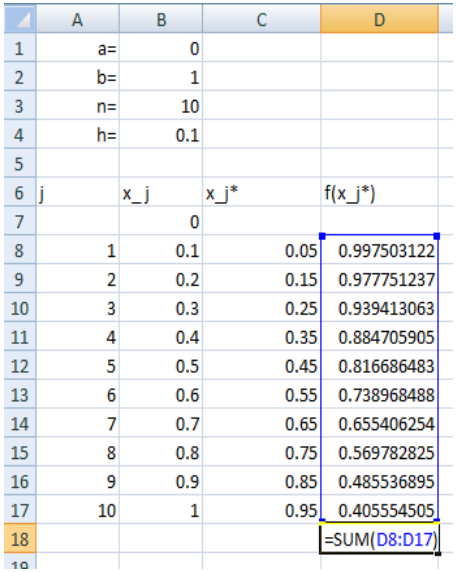
	A	B	C	D
1	a=	0		
2	b=	1		
3	n=	10		
4	h=	0.1		
5				
6	j	x_j	x_j*	f(x_j*)
7		0		
8	1	0.1	=(B7+B8)/2	
9	2	0.2		
10	3	0.3		
11	4	0.4		
12	5	0.5		
13	6	0.6		
14	7	0.7		
15	8	0.8		
16	9	0.9		
17	10	1		

8. Type the formula of $f(x_j^*)$: **=EXP(-(C8^2))** in cell **D8** and copy down until cell **D17.**



	A	B	C	D
1	a=	0		
2	b=	1		
3	n=	10		
4	h=	0.1		
5				
6	j	x_j	x_j*	f(x_j*)
7		0		
8	1	0.1	0.05	=EXP(-(C8^2))
9	2	0.2	0.15	
10	3	0.3	0.25	
11	4	0.4	0.35	
12	5	0.5	0.45	
13	6	0.6	0.55	
14	7	0.7	0.65	
15	8	0.8	0.75	
16	9	0.9	0.85	
17	10	1	0.95	

9. Type : **SUM(D8:D17)** in cell **D18**.



	A	B	C	D
1	a=	0		
2	b=	1		
3	n=	10		
4	h=	0.1		
5				
6	j	x_j	x_j*	f(x_j*)
7		0		
8	1	0.1	0.05	0.997503122
9	2	0.2	0.15	0.977751237
10	3	0.3	0.25	0.939413063
11	4	0.4	0.35	0.884705905
12	5	0.5	0.45	0.816686483
13	6	0.6	0.55	0.738968488
14	7	0.7	0.65	0.655406254
15	8	0.8	0.75	0.569782825
16	9	0.9	0.85	0.485536895
17	10	1	0.95	0.405554505
18				=SUM(D8:D17)

10. Finally, type J in cell **A19** and the formula of Rectangular rule in **B19** which gives the approximation of $J = \int_0^1 e^{-x^2}\, dx$.

	A	B	C	D
1	a=	0		
2	b=	1		
3	n=	10		
4	h=	0.1		
5				
6	j	x_j	x_j*	f(x_j*)
7		0		
8	1	0.1	0.05	0.997503122
9	2	0.2	0.15	0.977751237
10	3	0.3	0.25	0.939413063
11	4	0.4	0.35	0.884705905
12	5	0.5	0.45	0.816686483
13	6	0.6	0.55	0.738968488
14	7	0.7	0.65	0.655406254
15	8	0.8	0.75	0.569782825
16	9	0.9	0.85	0.485536895
17	10	1	0.95	0.405554505
18				7.471308777
19	J=	=B4*D18		

	A	B	C	D
1	a=	0		
2	b=	1		
3	n=	10		
4	h=	0.1		
5				
6	j	x_j	x_j*	f(x_j*)
7		0		
8	1	0.1	0.05	0.997503122
9	2	0.2	0.15	0.977751237
10	3	0.3	0.25	0.939413063
11	4	0.4	0.35	0.884705905
12	5	0.5	0.45	0.816686483
13	6	0.6	0.55	0.738968488
14	7	0.7	0.65	0.655406254
15	8	0.8	0.75	0.569782825
16	9	0.9	0.85	0.485536895
17	10	1	0.95	0.405554505
18				7.471308777
19				
20	J=	0.747131		

6.3 Trapezoidal Rule

- The trapezoidal rule is generally more accurate.
- We obtain it if we take the same subdivision as before and approximate f by broken line of segments (chords) with endpoints $[a, f(a)]$, $[x_1, f(x_1)), \dots, [b, f(b)]$ on the curve of f (Figure 6.3).

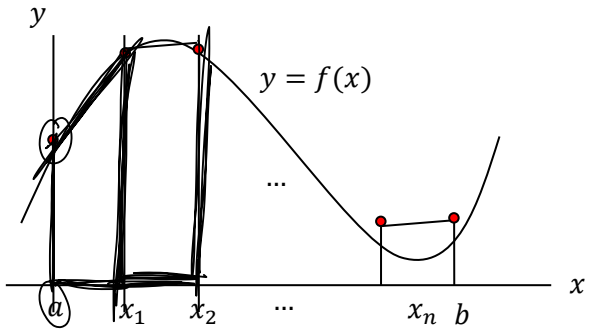


Figure 6.3 : Trapezoidal rule

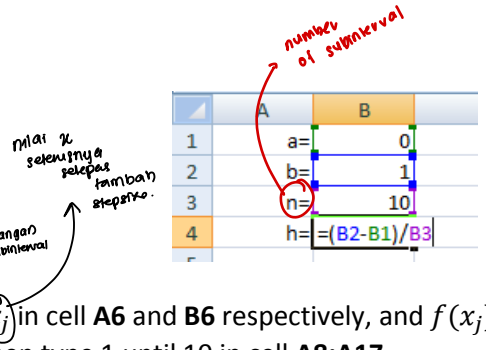
- Then, the area under the curve of a and b is approximated by n trapezoids of areas

$$J = \int_a^b f(x)\,dx \approx \frac{1}{2}[f(a) + f(x_1)]h + \frac{1}{2}[f(x_1) + f(x_2)]h \dots + \frac{1}{2}[f(x_{n-1}) + f(b)]h$$

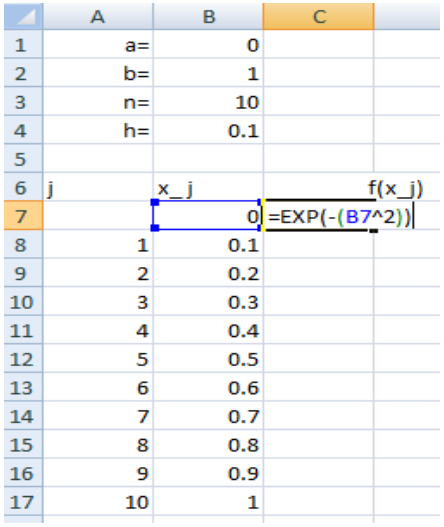
$$\approx h\left[\frac{1}{2}f(a) + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + \frac{1}{2}f(b)\right]$$

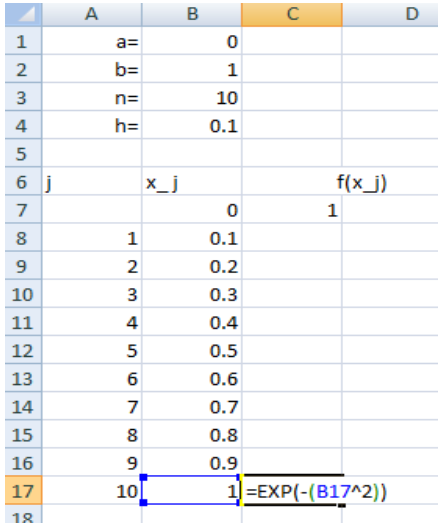
where $h = (b-a)/n$

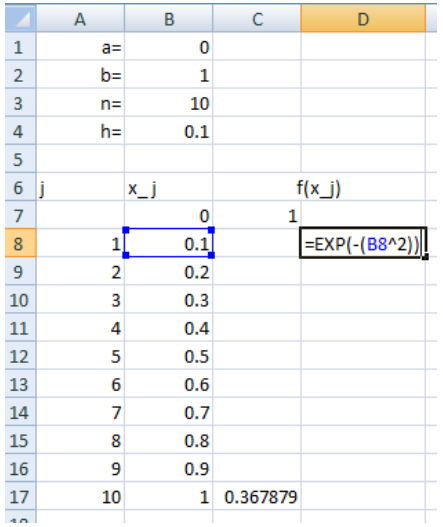
- Let's consider the similar example in 6.2. Now, we will use Excel to calculate the approximation of J using Trapezoidal rule.
  1. Enter the value of a, b and n in cell **B1**, **B2** and **B3** respectively.
  2. Calculate the value stepsize, h in cell **B4.**



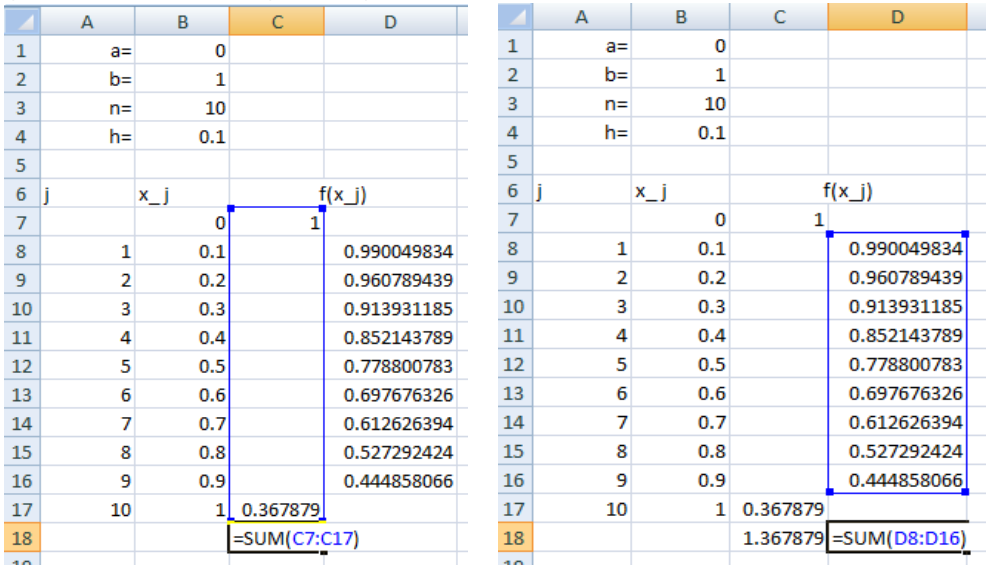
3. Type the heading of j and x_j in cell **A6** and **B6** respectively, and $f(x_j)$ in cells **C6** and **D6**.
4. We have 10 subintervals, then type 1 until 10 in cell **A8:A17.**
5. Enter the value of $a = 0$ in cell **B7**.
6. Type : **=B7+$B$4** in cell **B8** and copy down this formula until **B17.**
7. Type the formula of $f(x_j)$: **=EXP(-(B7^2))** in cell **C7** , **=EXP(-(B17^2))** in cell **C17** and **=EXP(-(B8^2))** in cell **D8:D16.**







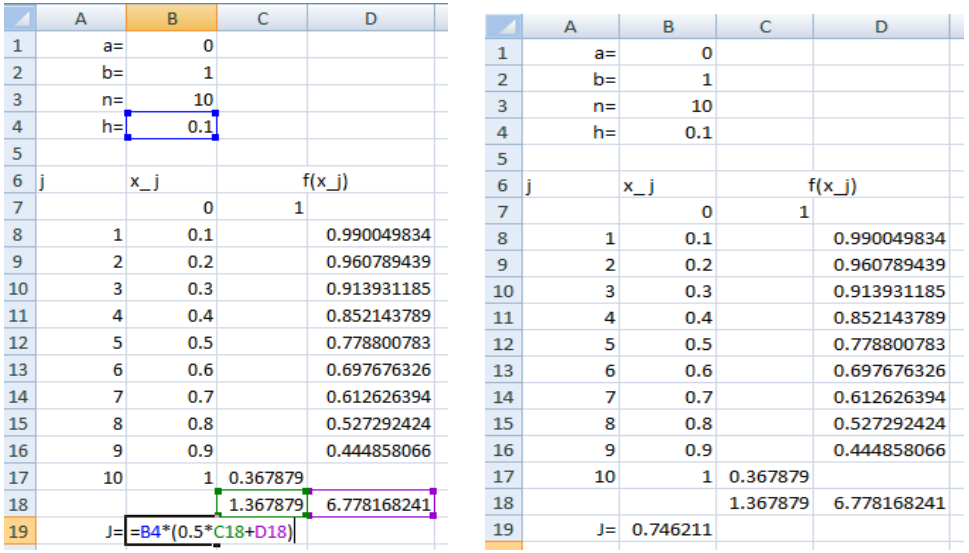
7. Type : **SUM(C7:C17)** in cell **C18** and **SUM(D8:D16)** in cell **D18.**



	A	B	C	D
1	a=	0		
2	b=	1		
3	n=	10		
4	h=	0.1		
5				
6	j	x_j		f(x_j)
7		0	1	
8	1	0.1		0.990049834
9	2	0.2		0.960789439
10	3	0.3		0.913931185
11	4	0.4		0.852143789
12	5	0.5		0.778800783
13	6	0.6		0.697676326
14	7	0.7		0.612626394
15	8	0.8		0.527292424
16	9	0.9		0.444858066
17	10	1	0.367879	
18			=SUM(C7:C17)	

	A	B	C	D
1	a=	0		
2	b=	1		
3	n=	10		
4	h=	0.1		
5				
6	j	x_j		f(x_j)
7		0	1	
8	1	0.1		0.990049834
9	2	0.2		0.960789439
10	3	0.3		0.913931185
11	4	0.4		0.852143789
12	5	0.5		0.778800783
13	6	0.6		0.697676326
14	7	0.7		0.612626394
15	8	0.8		0.527292424
16	9	0.9		0.444858066
17	10	1	0.367879	
18			1.367879	=SUM(D8:D16)

9. Finally, type J in cell **A19** and the formula of Trapezoidal rule in **B19** which gives the approximation of $J = \int_0^1 e^{-x^2}\, dx$.



	A	B	C	D
1	a=	0		
2	b=	1		
3	n=	10		
4	h=	0.1		
5				
6	j	x_j		f(x_j)
7		0	1	
8	1	0.1		0.990049834
9	2	0.2		0.960789439
10	3	0.3		0.913931185
11	4	0.4		0.852143789
12	5	0.5		0.778800783
13	6	0.6		0.697676326
14	7	0.7		0.612626394
15	8	0.8		0.527292424
16	9	0.9		0.444858066
17	10	1	0.367879	
18			1.367879	6.778168241
19	J=	=B4*(0.5*C18+D18)		

	A	B	C	D
1	a=	0		
2	b=	1		
3	n=	10		
4	h=	0.1		
5				
6	j	x_j		f(x_j)
7		0	1	
8	1	0.1		0.990049834
9	2	0.2		0.960789439
10	3	0.3		0.913931185
11	4	0.4		0.852143789
12	5	0.5		0.778800783
13	6	0.6		0.697676326
14	7	0.7		0.612626394
15	8	0.8		0.527292424
16	9	0.9		0.444858066
17	10	1	0.367879	
18			1.367879	6.778168241
19	J=	0.746211		

6.4 Simpson's Rule

- Piecewise constant approximation of f led to the Rectangular rule, piecewise line approximation to the Trapezoidal rule, and piecewise quadratic approximation will give Simpso's rule, which is of great practical importance because it is sufficiently accurate for most problems, but still sufficiently simple.
- To derive Simpson's rule, we divide the interval of integration $a \le x \le b$ into an even numberof equal subintervals, say, into $n = 2m$ subintervals of length $h = (b-a)/2m$ with endpoints $x_0 (= 0), x_1, \ldots, x_{2m-1}, x_{2m} (= b)$; see Figure 6.4.

Simpson's rule
- Quadratic approximation
- divide the interval into even number of equal subinterval.
- subinterval length → $h = \frac{(b-a)}{2m}$

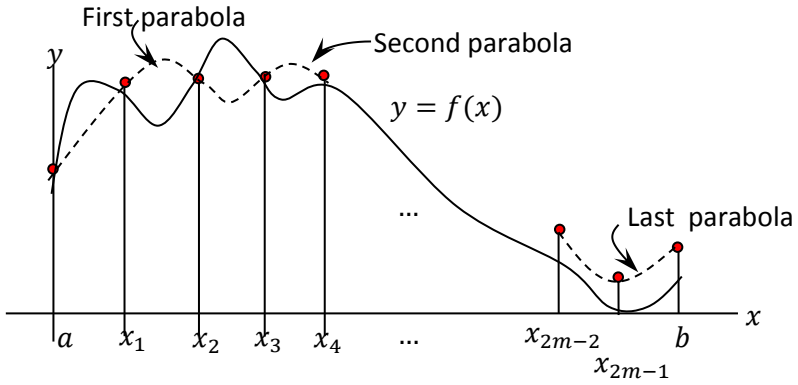


Figure 6.3 : Simpson's rule

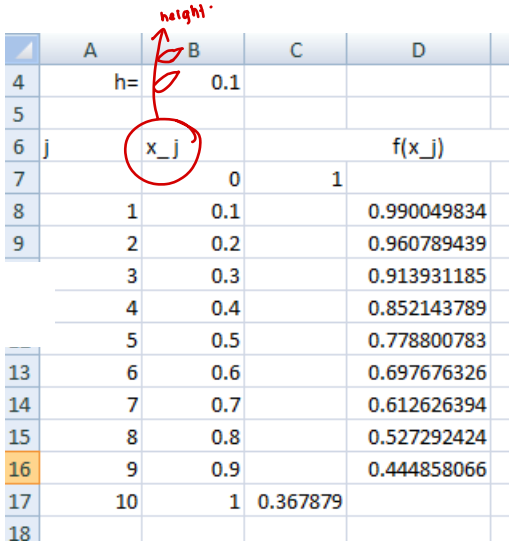
- The area under the curve of a and b is approximated by the Simpson's rule:

$$J = \int_a^b f(x)\,dx \approx \frac{h}{3}[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \cdots + 2f(x_{2m-2}) + 4f(x_{2m-1}) + f(x_{2m})]$$

- Let's consider the similar example in 6.3. Now, we will use Excel to calculate the approximation of J using Simpson's rule.
  1. Enter the value of a, b and n in cell **B1**, **B2** and **B3** respectively.
  2. Calculate the value stepsize, h in cell **B4.**
  3. Type the heading of j and x_j in cell **A6** and **B6** respectively, and $f(x_j)$ in cells **C6, D6 and E6**.

  3. Type the formula of $f(x_j)$: **=EXP(-(B7^2))** in cell **C7** , **=EXP(-(B17^2))** in cell **C17.**
  4. Type the formula of $f(x_j)$ **=EXP(-(B8^2))** in cell **D8** and copy down until cell **D16.**

$$J = \int_a^b f(x)\,dx \approx \frac{h}{3}[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \cdots + 2f(x_{2m-2}) + 4f(x_{2m-1}) + f(x_{2m})]$$

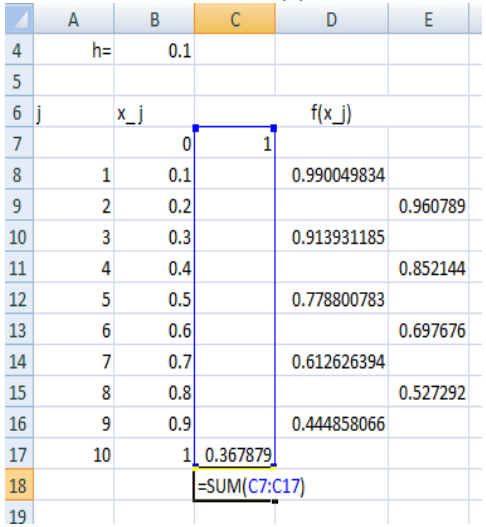


	A	B	C	D
4	h=	0.1		
5				
6	j	x_j		f(x_j)
7		0	1	
8	1	0.1		0.990049834
9	2	0.2		0.960789439
	3	0.3		0.913931185
	4	0.4		0.852143789
	5	0.5		0.778800783
13	6	0.6		0.697676326
14	7	0.7		0.612626394
15	8	0.8		0.527292424
16	9	0.9		0.444858066
17	10	1	0.367879	
18				

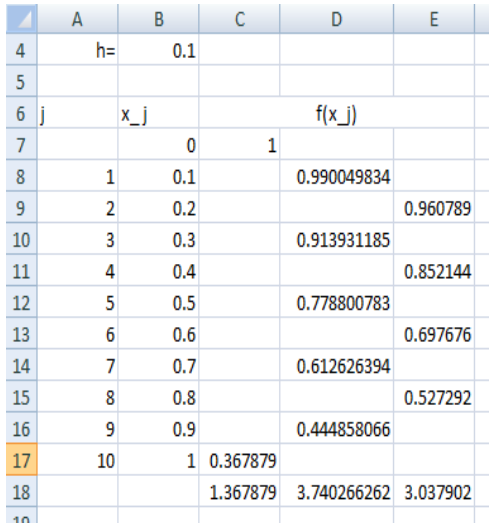
5. Move the formula for $f(x_2)$, $f(x_4)$, $f(x_6)$ and $f(x_8)$ in cell D9, D11, D13 and D15 to cell E9, E11, E13 and E15 respectively.

	A	B	C	D	E
4	h=	0.1			
5					
6	j	x_j		f(x_j)	
7		0	1		
8	1	0.1		0.990049834	
9	2	0.2			0.960789
10	3	0.3		0.913931185	
11	4	0.4			0.852144
12	5	0.5		0.778800783	
13	6	0.6			0.697676
14	7	0.7		0.612626394	
15	8	0.8			0.527292
16	9	0.9		0.444858066	
17	10	1	0.367879		
18					

6. Type : **=SUM(C7:C17)** in cell **C18** and copy this formula across to cell **E18**.

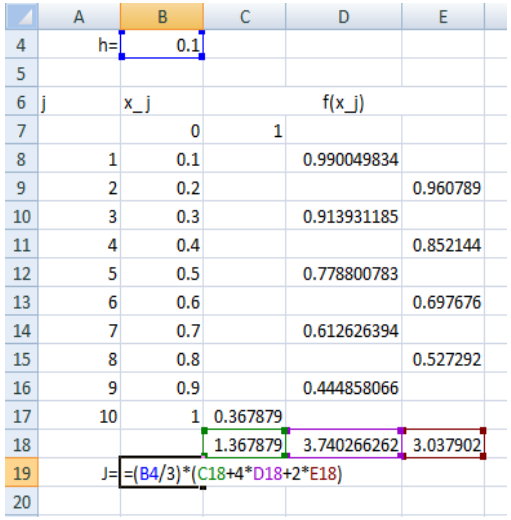


	A	B	C	D	E
4	h=	0.1			
5					
6	j	x_j		f(x_j)	
7		0	1		
8	1	0.1		0.990049834	
9	2	0.2			0.960789
10	3	0.3		0.913931185	
11	4	0.4			0.852144
12	5	0.5		0.778800783	
13	6	0.6			0.697676
14	7	0.7		0.612626394	
15	8	0.8			0.527292
16	9	0.9		0.444858066	
17	10	1	0.367879		
18			=SUM(C7:C17)		
19					

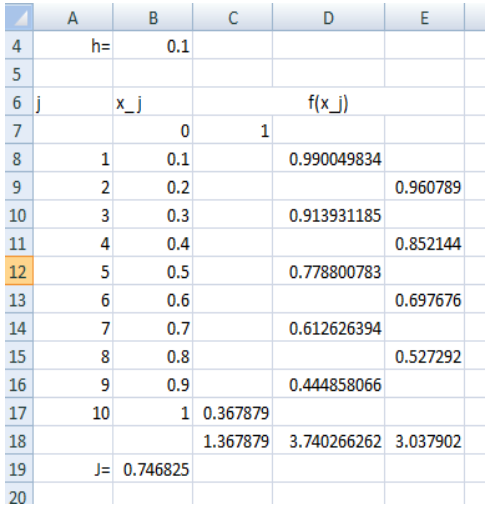


	A	B	C	D	E
4	h=	0.1			
5					
6	j	x_j		f(x_j)	
7		0	1		
8	1	0.1		0.990049834	
9	2	0.2			0.960789
10	3	0.3		0.913931185	
11	4	0.4			0.852144
12	5	0.5		0.778800783	
13	6	0.6			0.697676
14	7	0.7		0.612626394	
15	8	0.8			0.527292
16	9	0.9		0.444858066	
17	10	1	0.367879		
18			1.367879	3.740266262	3.037902

7. Finally, type J in cell **A19** and the formula of Simpson's rule in **B19** which gives the approximation of $J = \int_0^1 e^{-x^2}\, dx$.

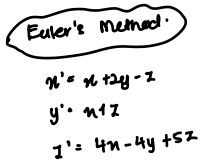


	A	B	C	D	E
4	h=	0.1			
5					
6	j	x_j		f(x_j)	
7		0	1		
8	1	0.1		0.990049834	
9	2	0.2			0.960789
10	3	0.3		0.913931185	
11	4	0.4			0.852144
12	5	0.5		0.778800783	
13	6	0.6			0.697676
14	7	0.7		0.612626394	
15	8	0.8			0.527292
16	9	0.9		0.444858066	
17	10	1	0.367879		
18			1.367879	3.740266262	3.037902
19	J=	=(B4/3)*(C18+4*D18+2*E18)			
20					



	A	B	C	D	E
4	h=	0.1			
5					
6	j	x_j		f(x_j)	
7		0	1		
8	1	0.1		0.990049834	
9	2	0.2			0.960789
10	3	0.3		0.913931185	
11	4	0.4			0.852144
12	5	0.5		0.778800783	
13	6	0.6			0.697676
14	7	0.7		0.612626394	
15	8	0.8			0.527292
16	9	0.9		0.444858066	
17	10	1	0.367879		
18			1.367879	3.740266262	3.037902
19	J=	0.746825			
20					

Any questions?



CHAPTER 7 : ORDINARY DIFFERENTIAL EQUATIONS

7.1 Introduction

- There are many situations in science and engineering in which one encounters ordinary differential equations, ODE.

- The general form of ODE :

$$\frac{dy}{dx} = f(x, y)$$

- Most ODE models are complicated enough to preclude analytical methods of solution; instead, numerical methods must be used.

- In this course, we will consider two numerical methods to approximate the solution to the initial value problem in ODE :
  4. Euler's method
  5. Taylor's method

7.2 Euler's method

- Euler's method (or the tangent line method) is a procedure for constructing approximate solutions to an initial value problem for a first order differential equation

$$\frac{dy}{dx} = f(x, y), \qquad a \le x \le b, \quad y(a) = \propto$$

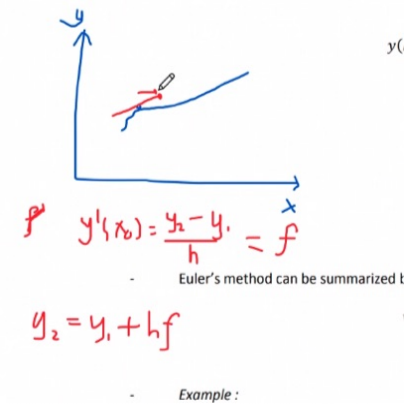
initial value problem

at $N + 1$ equally spaced numbers in the interval $[a, b]$.

- In actuality, a continuous approximation to the solution $y(x)$ will not be obtained; instead approximations to $y(x)$ will be generated at various values, called mesh points, in the interval $[a, b]$. Once the approximate solution is obtained at the points, the approximate solution at other points in the interval can be obtained by interpolation.

- To interpret Euler's method geometrically, note that when w_i is close approximation to $y(x_i)$, the assumption that the problem is well-posed implies that

$$f(x_i, w_i) \approx y'(x_i) = f\big(x_i,\ y(x_i)\big).$$



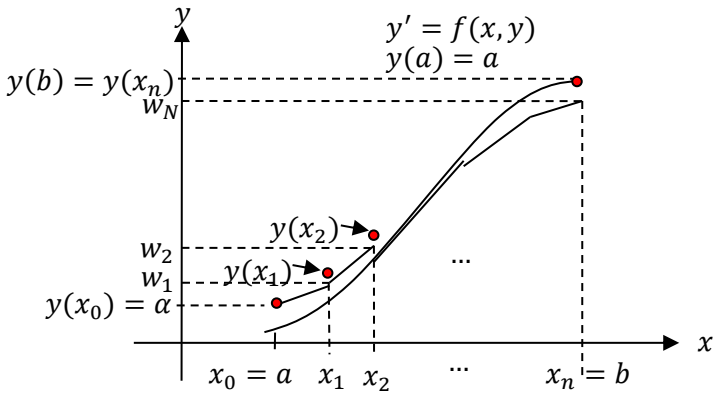


Figure 7.1 : Euler's method

- Euler's method can be summarized by the recursive formulas

$$w_{i+1} = w_i + hf(x_i, w_i), \quad w_0 = \alpha, \quad i = 0, 1, 2, \ldots .$$

y-axis

- *Example :*
  Use Euler's method to find approximates to the solution of the initial value problem

$$\frac{dy}{dx} = y - x^2 + 1, \quad 0 \le x \le 2, \quad y(0) = 0.5, \text{ with } 10 \text{ steps}$$

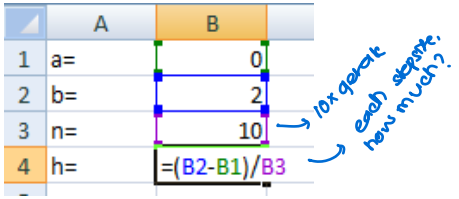
w

where the exact solution is $y(x) = (x+1)^2 - 0.5e^x$.

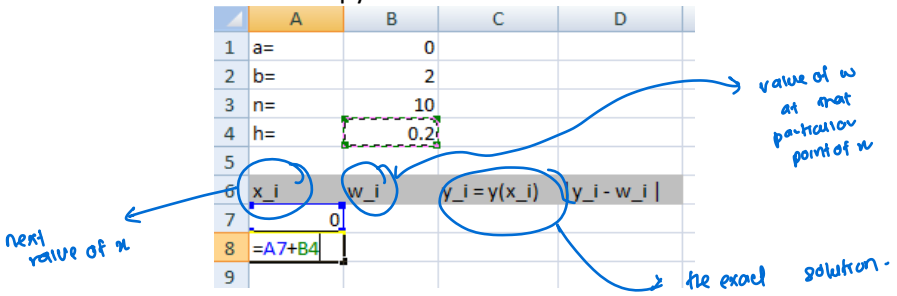
(x) n = 10 steps
(x) h = value of h will measure step size how small the approximating segments are.

- We will use Excel to calculate the approximation the solution of initial value problem using Euler's method.
  2. Enter the value of a, b and n in cell **B1**, **B2** and **B3** respectively.
  3. Calculate the value stepsize, h in cell **B4.**

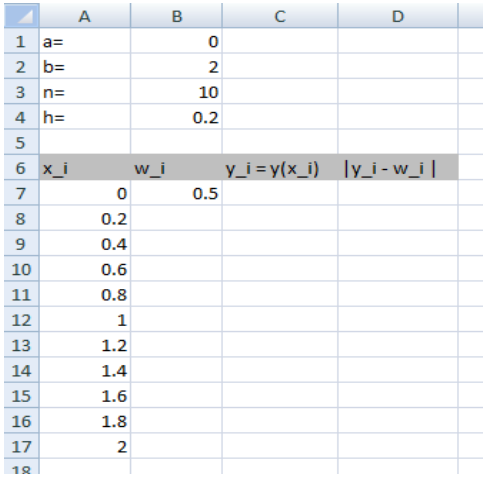
(x+h)



4. Type the heading of x_i, w_i $y_i = y(x_i)$ and $|y_i - w_i|$ in cell **A6**, **B6**, **C6** and **D6** respectively.
5. Enter the value of $x_0 = a$ in cell **A7**.
5. Type the formula : = A7+B4 in cell **A8** and copy down this formula until the cell **A17**.



6. Enter the value of $w_0 = y(x_0 = 0) = 0.5$ in cell **B7.**



7. Use the Euler's formula : $w_{i+1} = w_i + h(w_i - x_i^2 + 1)$, type : **=B7+$B$4*(B7-A7^2+1)** in cell **B8** and copy down this formula until cell **B17**.

	A	B	C	D
1	a=	0		
2	b=	2		
3	n=	10		
4	h=	0.2		
5				
6	x_i	w_i	y_i = y(x_i)	|y_i - w_i|
7	0	0.5		
8	0.2	=B7+$B$4*(B7-A7^2+1)		
9	0.4			
10	0.6			
11	0.8			
12	1			
13	1.2			
14	1.4			
15	1.6			
16	1.8			
17	2			

	A	B	C	D
1	a=	0		
2	b=	2		
3	n=	10		
4	h=	0.2		
5				
6	x_i	w_i	y_i = y(x_i)	|y_i - w_i|
7	0	0.5		
8	0.2	0.8		
9	0.4	1.152		
10	0.6	1.5504		
11	0.8	1.98848		
12	1	2.458176		
13	1.2	2.949811		
14	1.4	3.451773		
15	1.6	3.950128		
16	1.8	4.428154		
17	2	4.865785		

8. Type the formula for the exact solution : **=(A7+1)^2 - 0.5*exp(A7)** in cell **C7** and copy down this formula until cell **B17**.

	A	B	C	D
1	a=	0		
2	b=	2		
3	n=	10		
4	h=	0.2		
5				
6	x_i	w_i	y_i = y(x_i)	|y_i - w_i|
7	0	0.5	=(A7+1)^2 - 0.5*exp(A7)	
8	0.2	0.8		
9	0.4	1.152		
10	0.6	1.5504		
11	0.8	1.98848		
12	1	2.458176		
13	1.2	2.949811		
14	1.4	3.451773		
15	1.6	3.950128		
16	1.8	4.428154		
17	2	4.865785		

	A	B	C	D
1	a=	0		
2	b=	2		
3	n=	10		
4	h=	0.2		
5				
6	x_i	w_i	y_i = y(x_i)	|y_i - w_i|
7	0	0.5	0.5	
8	0.2	0.8	0.82929862	
9	0.4	1.152	1.21408765	
10	0.6	1.5504	1.6489406	
11	0.8	1.98848	2.12722954	
12	1	2.458176	2.64085909	
13	1.2	2.949811	3.17994154	
14	1.4	3.451773	3.73240002	
15	1.6	3.950128	4.28348379	
16	1.8	4.428154	4.81517627	
17	2	4.865785	5.30547195	

9. Type the formula for the error of using Euler's method : **=abs(C7-B7)** and copy down this formula until cell **D17**.

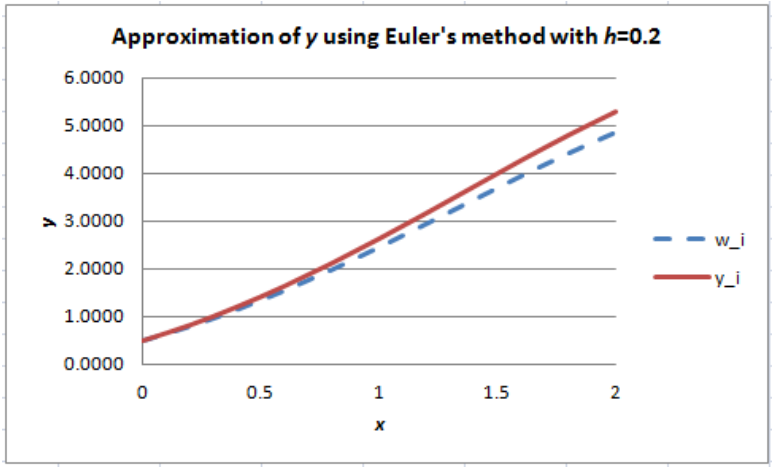
	A	B	C	D
1	a=	0		
2	b=	2		
3	n=	10		
4	h=	0.2		
5				
6	x_i	w_i	y_i = y(x_i)	|y_i - w_i|
7	0	0.5	0.5	=abs(C7-B7)
8	0.2	0.8	0.82929862	
9	0.4	1.152	1.21408765	
10	0.6	1.5504	1.6489406	
11	0.8	1.98848	2.12722954	
12	1	2.458176	2.64085909	
13	1.2	2.949811	3.17994154	
14	1.4	3.451773	3.73240002	
15	1.6	3.950128	4.28348379	
16	1.8	4.428154	4.81517627	
17	2	4.865785	5.30547195	

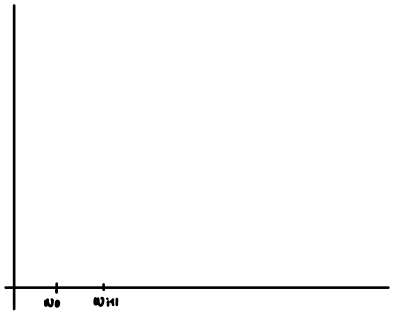
	A	B	C	D
1	a=	0		
2	b=	2		
3	n=	10		
4	h=	0.2		
5				
6	x_i	w_i	y_i = y(x_i)	|y_i - w_i|
7	0	0.5	0.5	0
8	0.2	0.8	0.82929862	0.029298621
9	0.4	1.152	1.21408765	0.062087651
10	0.6	1.5504	1.6489406	0.0985406
11	0.8	1.98848	2.12722954	0.138749536
12	1	2.458176	2.64085909	0.182683086
13	1.2	2.949811	3.17994154	0.230130339
14	1.4	3.451773	3.73240002	0.280626577
15	1.6	3.950128	4.28348379	0.33335566
16	1.8	4.428154	4.81517627	0.387022514
17	2	4.865785	5.30547195	0.439687446

10. You may format the value for w_i, y_i and $|y_i - w_i|$ in same number of decimal places.

	A	B	C	D
1	a=	0		
2	b=	2		
3	n=	10		
4	h=	0.2		
5				
6	x_i	w_i	y_i = y(x_i)	|y_i - w_i|
7	0	0.5000	0.5000	0.0000
8	0.2	0.8000	0.8293	0.0293
9	0.4	1.1520	1.2141	0.0621
10	0.6	1.5504	1.6489	0.0985
11	0.8	1.9885	2.1272	0.1387
12	1	2.4582	2.6409	0.1827
13	1.2	2.9498	3.1799	0.2301
14	1.4	3.4518	3.7324	0.2806
15	1.6	3.9501	4.2835	0.3334
16	1.8	4.4282	4.8152	0.3870
17	2	4.8658	5.3055	0.4397

11. Try to plot the graph includes the actual and the approximate results.





7.3 Taylor methods

- Taylor methods is a numerical method that have faster rates of convergence which are natural extensions of the Euler procedure.
- Taylor method of order n :

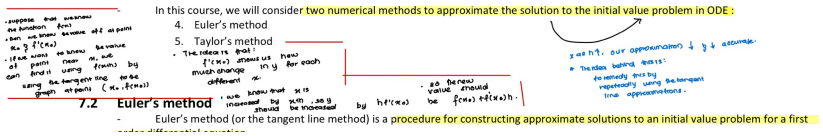
$$w_0 = \alpha$$

$$w_{i+1} = w_i + hT^{(n)}(x_i, w_i) \text{ for each } i = 0, 1, \ldots, N-1$$

where

$$T^{(n)}(x_i, w_i) = f(x_i, w_i) + \frac{h}{2}f'(x_i, w_i) + \cdots + \frac{h}{n!}f^{n-1}(x_i, w_i).$$

- *Let's consider the similar example in 7.2 :*
  Use Taylor's method of order two and four to the initial value problem

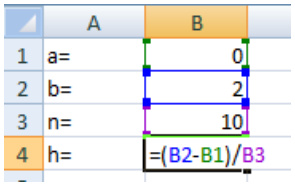


$$\frac{dy}{dx} = y - x^2 + 1, \quad 0 \le x \le 2, \quad y(0) = 0.5, \textit{ with } 10 \textit{ steps}$$

where the exact solution is $y(x) = (x+1)^2 - 0.5e^x$.

- We will use Excel to calculate the approximation the solution of initial value problem using Taylor's method.

1. Enter the value of a, b and n in cell **B1**, **B2** and **B3** respectively.
2. Calculate the value stepsize, h in cell **B4.**



	A	B
1	a=	0
2	b=	2
3	n=	10
4	h=	=(B2-B1)/B3

3. Type the heading of $x_i, w_i(TO2), y_i = y(x_i), |y_i - w_i|, w_i(TO4), y_i = y(x_i)$ and $|y_i - w_i|$ in cell **A6**, **B6**, **C6, D6, E6, F6** and **G6** respectively.

	A	B	C	D	E	F	G
1	a=	0					
2	b=	2					
3	n=	10					
4	h=	0.2					
5							
6	x_i	w_i (TO2)	y_i = y(x_i)	|y_i - w_i|	w_i (TO4)	y_i = y(x_i)	|y_i - w_i|
7							

4. Enter the value of $x_0 = a$ in cell **A7**.
5. Type the formula : = A7+B4 in cell **A8** and copy down this formula until the cell **A17**.

	A	B	C	D	E	F	G
1	a=	0					
2	b=	2					
3	n=	10					
4	h=	0.2					
5							
6	x_i	w_i (TO2)	y_i = y(x_i)	|y_i - w_i|	w_i (TO4)	y_i = y(x_i)	|y_i - w_i|
7	0						
8	=A7+B4						

6. Enter the value of $w_0 = y(x_0 = 0) = 0.5$ in cell **B7.**

	A	B	C	D	E	F	G
1	a=	0					
2	b=	2					
3	n=	10					
4	h=	0.2					
5							
6	x_i	w_i (TO2)	y_i = y(x_i)	|y_i - w_i|	w_i (TO4)	y_i = y(x_i)	|y_i - w_i|
7	0	0.5000					
8	0.2						
9	0.4						
10	0.6						
11	0.8						
12	1						
13	1.2						
14	1.4						
15	1.6						
16	1.8						
17	2						

$f(x,y) = dy/dx$

7. Use the Taylor's formula : $w_{i+1} = w_i + hT^{(2)}(x_i, w_i)$ where $T^{(2)}(x_i, w_i) = f(x_i, w_i) + \frac{h}{2}f'(x_i, w_i)$

$$f'(x_i, y(x_i)) = \frac{d}{dx}(y - x^2 + 1) = y' - 2x$$
$$= y - x^2 + 1 - 2x = y - x^2 - 2x + 1$$

So,

$$T^{(2)}(x_i, w_i) = (w_i - {x_i}^2 + 1) + \frac{h}{2}(w_i - {x_i}^2 - 2x_i + 1).$$

$T(2)$

It follows : $w_{i+1} = w_i + h\left[(w_i - {x_i}^2 + 1) + \frac{h}{2}(w_i - {x_i}^2 - 2x_i + 1)\right]$.

Now, type : **=B7+$B$4*((B7-(A7^2)+1)+($B$4/2)*(B7-(A7^2)-2*A7+1))** in cell **B8** and copy down this formula until cell **B17**.

$\frac{dy}{dx} = y - x^2 + 1 = y'$

$\frac{d^2y}{dx^2} = y' - 2x$

$= (y - x^2 + 1) - 2x$

$= y - x^2 + 1 - 2x$

	A	B	C	D	E	F	G
1	a=	0					
2	b=	2					
3	n=	10					
4	h=	0.2					
5							
6	x_i	w_i (TO2)	y_i = y(x_i)	|y_i - w_i|	w_i (TO4)	y_i = y(x_i)	|y_i - w_i|
7	0	0.5000					
8	0.2	=B7+$B$4*((B7-(A7^2)+1)+($B$4/2)*(B7-(A7^2)-2*A7+1))					
9	0.4						
10	0.6						
11	0.8						
12	1						
13	1.2						
14	1.4						
15	1.6						
16	1.8						
17	2						

	A	B	C	D	E	F	G
1	a=	0					
2	b=	2					
3	n=	10					
4	h=	0.2					
5							
6	x_i	w_i (TO2)	y_i = y(x_i)	|y_i - w_i|	w_i (TO4)	y_i = y(x_i)	|y_i - w_i|
7	0	0.5000					
8	0.2	0.8300					
9	0.4	1.2158					
10	0.6	1.6521					
11	0.8	2.1323					
12	1	2.6486					
13	1.2	3.1913					
14	1.4	3.7486					
15	1.6	4.3061					
16	1.8	4.8463					
17	2	5.3477					

8. Type the formula for the exact solution : **=(A7+1)^2 - 0.5*exp(A7)** in cell **C7** and copy down this formula until cell **B17**.

	A	B	C	D	E	F	G
1	a=	0					
2	b=	2					
3	n=	10					
4	h=	0.2					
5							
6	x_i	w_i (TO2)	y_i = y(x_i)	|y_i - w_i|	w_i (TO4)	y_i = y(x_i)	|y_i - w_i|
7	0	0.5000	=(A7+1)^2 - 0.5*exp(A7)				
8	0.2	0.8300					
9	0.4	1.2158					
10	0.6	1.6521					
11	0.8	2.1323					
12	1	2.6486					
13	1.2	3.1913					
14	1.4	3.7486					
15	1.6	4.3061					
16	1.8	4.8463					
17	2	5.3477					

	A	B	C	D	E	F	G
1	a=	0					
2	b=	2					
3	n=	10					
4	h=	0.2					
5							
6	x_i	w_i (TO2)	y_i = y(x_i)	|y_i - w_i|	w_i (TO4)	y_i = y(x_i)	|y_i - w_i|
7	0	0.5000	0.5000				
8	0.2	0.8300	0.8293				
9	0.4	1.2158	1.2141				
10	0.6	1.6521	1.6489				
11	0.8	2.1323	2.1272				
12	1	2.6486	2.6409				
13	1.2	3.1913	3.1799				
14	1.4	3.7486	3.7324				
15	1.6	4.3061	4.2835				
16	1.8	4.8463	4.8152				
17	2	5.3477	5.3055				

9. Type the formula for the error of using Taylor's method of order 2 : **=abs(C7-B7)** in cell **D7** and copy down this formula until cell **D17**.

	A	B	C	D	E	F	G
1	a=	0					
2	b=	2					
3	n=	10					
4	h=	0.2					
5							
6	x_i	w_i (TO2)	y_i = y(x_i)	|y_i - w_i |	w_i (TO4)	y_i = y(x_i)	|y_i - w_i |
7	0	0.5000	0.5000	=abs(C7-B7)			
8	0.2	0.8300	0.8293				
9	0.4	1.2158	1.2141				
10	0.6	1.6521	1.6489				
11	0.8	2.1323	2.1272				
12	1	2.6486	2.6409				
13	1.2	3.1913	3.1799				
14	1.4	3.7486	3.7324				
15	1.6	4.3061	4.2835				
16	1.8	4.8463	4.8152				
17	2	5.3477	5.3055				

	A	B	C	D	E	F	G
1	a=	0					
2	b=	2					
3	n=	10					
4	h=	0.2					
5							
6	x_i	w_i (TO2)	y_i = y(x_i)	|y_i - w_i |	w_i (TO4)	y_i = y(x_i)	|y_i - w_i |
7	0	0.5000	0.5000	0.0000			
8	0.2	0.8300	0.8293	0.0007			
9	0.4	1.2158	1.2141	0.0017			
10	0.6	1.6521	1.6489	0.0031			
11	0.8	2.1323	2.1272	0.0051			
12	1	2.6486	2.6409	0.0078			
13	1.2	3.1913	3.1799	0.0114			
14	1.4	3.7486	3.7324	0.0162			
15	1.6	4.3061	4.2835	0.0227			
16	1.8	4.8463	4.8152	0.0311			
17	2	5.3477	5.3055	0.0422			

10. Enter the value of $w_0 = y(x_0 = 0) = 0.5$ in cell **E7.**

	A	B	C	D	E	F	G
1	a=	0					
2	b=	2					
3	n=	10					
4	h=	0.2					
5							
6	x_i	w_i (TO2)	y_i = y(x_i)	|y_i - w_i |	w_i (TO4)	y_i = y(x_i)	|y_i - w_i |
7	0	0.5000	0.5000	0.0000	0.5000		
8	0.2	0.8300	0.8293	0.0007			
9	0.4	1.2158	1.2141	0.0017			
10	0.6	1.6521	1.6489	0.0031			
11	0.8	2.1323	2.1272	0.0051			
12	1	2.6486	2.6409	0.0078			
13	1.2	3.1913	3.1799	0.0114			
14	1.4	3.7486	3.7324	0.0162			
15	1.6	4.3061	4.2835	0.0227			
16	1.8	4.8463	4.8152	0.0311			
17	2	5.3477	5.3055	0.0422			

2nd order 4th order

11. Use the Taylor's formula : $w_{i+1} = w_i + hT^{(4)}(x_i, w_i)$ where $T^{(4)}(x_i, w_i) = f(x_i, w_i) + \frac{h}{2}f'(x_i, w_i) + \frac{h^2}{6}f''(x_i, w_i) + \frac{h^3}{24}f'''(x_i, w_i)$

$$f'(x_i, y(x_i)) = \frac{d}{dx}(y - x^2 + 1) = y' - 2x$$
$$= y - x^2 + 1 - 2x = y - x^2 - 2x + 1$$

$$f''(x_i, y(x_i)) = \frac{d}{dx}(y - x^2 - 2x + 1) = y' - 2x - 2$$
$$= y - x^2 + 1 - 2x - 2 = y - x^2 - 2x - 1$$

$$f'''(x_i, y(x_i)) = \frac{d}{dx}(y - x^2 - 2x - 1) = y' - 2x - 2$$
$$= y - x^2 + 1 - 2x - 2 = y - x^2 - 2x - 1$$

So,

$$T^{(4)}(x_i, w_i) = (w_i - {x_i}^2 + 1) + \tfrac{h}{2}(w_i - {x_i}^2 - 2x_i + 1) + \frac{h^2}{6}(w_i - {x_i}^2 - 2x_i - 1) + \frac{h^3}{24}(w_i - {x_i}^2 - 2x_i - 1)$$

.

It follows : $w_{i+1} = w_i + h[(w_i - {x_i}^2 + 1) + \tfrac{h}{2}(w_i - {x_i}^2 - 2x_i + 1) + \frac{h^2}{6}(w_i - {x_i}^2 - 2x_i - 1) + \frac{h^3}{24}(w_i - {x_i}^2 - 2x_i - 1)]$.

Now, type : **=E7+$B$4*((E7-(A7^2)+1)+($B$4/2)*(E7-(A7^2)-2*A7+1) + ($B$4^2/6)* (E7-(A7^2)-2*A7 -1) +($B$4^3/24)*(E7-(A7^2) - 2*A7-1))** in cell **E8** and copy down this formula until cell **E17**.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	a=	0													
2	b=	2													
3	n=	10													
4	h=	0.2													
5															
6	x_i	w_i (TO2)	y_i = y(x_i)	|y_i - w_i |	w_i (TO4)	y_i = y(x_i)	|y_i - w_i |								
7	0	0.5000	0.5000	0.0000	0.5000										
8	0.2	0.8300	0.8293	0.0007	=E7+$B$4*((E7-(A7^2)+1)+($B$4/2)*(E7-(A7^2)-2*A7+1) + ($B$4^2/6)* (E7-(A7^2)-2*A7 -1) +($B$4^3/24)*(E7-(A7^2) - 2*A7-1))										
9	0.4	1.2158	1.2141	0.0017											
10	0.6	1.6521	1.6489	0.0031											
11	0.8	2.1323	2.1272	0.0051											
12	1	2.6486	2.6409	0.0078											
13	1.2	3.1913	3.1799	0.0114											
14	1.4	3.7486	3.7324	0.0162											
15	1.6	4.3061	4.2835	0.0227											
16	1.8	4.8463	4.8152	0.0311											
17	2	5.3477	5.3055	0.0422											

	A	B	C	D	E	F	G
1	a=	0					
2	b=	2					
3	n=	10					
4	h=	0.2					
5							
6	x_i	w_i (TO2)	y_i = y(x_i)	|y_i - w_i |	w_i (TO4)	y_i = y(x_i)	|y_i - w_i |
7	0	0.5000	0.5000	0.0000	0.5000		
8	0.2	0.8300	0.8293	0.0007	0.8293		
9	0.4	1.2158	1.2141	0.0017	1.2141		
10	0.6	1.6521	1.6489	0.0031	1.6489		
11	0.8	2.1323	2.1272	0.0051	2.1272		
12	1	2.6486	2.6409	0.0078	2.6409		
13	1.2	3.1913	3.1799	0.0114	3.1800		
14	1.4	3.7486	3.7324	0.0162	3.7324		
15	1.6	4.3061	4.2835	0.0227	4.2835		
16	1.8	4.8463	4.8152	0.0311	4.8152		
17	2	5.3477	5.3055	0.0422	5.3056		

12. Type the formula for the exact solution : **=(A7+1)^2 - 0.5*exp(A7)** in cell **F7** and copy down this formula until cell **F17**.

	A	B	C	D	E	F	G
1	a=	0					
2	b=	2					
3	n=	10					
4	h=	0.2					
5							
6	x_i	w_i (TO2)	y_i = y(x_i)	|y_i - w_i |	w_i (TO4)	y_i = y(x_i)	|y_i - w_i |
7	0	0.5000	0.5000	0.0000	0.5000	=(A7+1)^2 - 0.5*EXP(A7)	
8	0.2	0.8300	0.8293	0.0007	0.8293		
9	0.4	1.2158	1.2141	0.0017	1.2141		
10	0.6	1.6521	1.6489	0.0031	1.6489		
11	0.8	2.1323	2.1272	0.0051	2.1272		
12	1	2.6486	2.6409	0.0078	2.6409		
13	1.2	3.1913	3.1799	0.0114	3.1800		
14	1.4	3.7486	3.7324	0.0162	3.7324		
15	1.6	4.3061	4.2835	0.0227	4.2835		
16	1.8	4.8463	4.8152	0.0311	4.8152		
17	2	5.3477	5.3055	0.0422	5.3056		

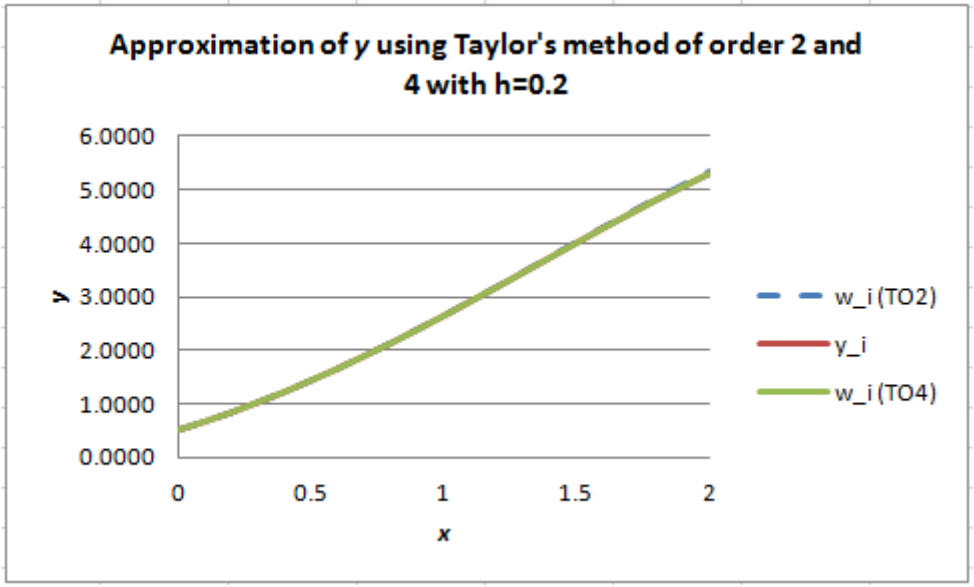
	A	B	C	D	E	F	G
1	a=	0					
2	b=	2					
3	n=	10					
4	h=	0.2					
5							
6	x_i	w_i (TO2)	y_i = y(x_i)	|y_i - w_i |	w_i (TO4)	y_i = y(x_i)	|y_i - w_i |
7	0	0.5000	0.5000	0.0000	0.5000	0.5000	
8	0.2	0.8300	0.8293	0.0007	0.8293	0.8293	
9	0.4	1.2158	1.2141	0.0017	1.2141	1.2141	
10	0.6	1.6521	1.6489	0.0031	1.6489	1.6489	
11	0.8	2.1323	2.1272	0.0051	2.1272	2.1272	
12	1	2.6486	2.6409	0.0078	2.6409	2.6409	
13	1.2	3.1913	3.1799	0.0114	3.1800	3.1799	
14	1.4	3.7486	3.7324	0.0162	3.7324	3.7324	
15	1.6	4.3061	4.2835	0.0227	4.2835	4.2835	
16	1.8	4.8463	4.8152	0.0311	4.8152	4.8152	
17	2	5.3477	5.3055	0.0422	5.3056	5.3055	

13. Type the formula for the error of using Taylor's method of order 4: **=abs(F7-E7)** in cell **G7** and copy down this formula until cell **G17**.

	A	B	C	D	E	F	G
1	a=	0					
2	b=	2					
3	n=	10					
4	h=	0.2					
5							
6	x_i	w_i (TO2)	y_i = y(x_i)	|y_i - w_i |	w_i (TO4)	y_i = y(x_i)	|y_i - w_i |
7	0	0.5000	0.5000	0.0000	0.5000	0.5000	=ABS(F7-E7)
8	0.2	0.8300	0.8293	0.0007	0.8293	0.8293	
9	0.4	1.2158	1.2141	0.0017	1.2141	1.2141	
10	0.6	1.6521	1.6489	0.0031	1.6489	1.6489	
11	0.8	2.1323	2.1272	0.0051	2.1272	2.1272	
12	1	2.6486	2.6409	0.0078	2.6409	2.6409	
13	1.2	3.1913	3.1799	0.0114	3.1800	3.1799	
14	1.4	3.7486	3.7324	0.0162	3.7324	3.7324	
15	1.6	4.3061	4.2835	0.0227	4.2835	4.2835	
16	1.8	4.8463	4.8152	0.0311	4.8152	4.8152	
17	2	5.3477	5.3055	0.0422	5.3056	5.3055	

	A	B	C	D	E	F	G
1	a=	0					
2	b=	2					
3	n=	10					
4	h=	0.2					
5							
6	x_i	w_i (TO2)	y_i = y(x_i)	|y_i - w_i |	w_i (TO4)	y_i = y(x_i)	|y_i - w_i |
7	0	0.5000	0.5000	0.0000	0.5000	0.5000	0.0000
8	0.2	0.8300	0.8293	0.0007	0.8293	0.8293	0.0000
9	0.4	1.2158	1.2141	0.0017	1.2141	1.2141	0.0000
10	0.6	1.6521	1.6489	0.0031	1.6489	1.6489	0.0000
11	0.8	2.1323	2.1272	0.0051	2.1272	2.1272	0.0000
12	1	2.6486	2.6409	0.0078	2.6409	2.6409	0.0000
13	1.2	3.1913	3.1799	0.0114	3.1800	3.1799	0.0000
14	1.4	3.7486	3.7324	0.0162	3.7324	3.7324	0.0000
15	1.6	4.3061	4.2835	0.0227	4.2835	4.2835	0.0000
16	1.8	4.8463	4.8152	0.0311	4.8152	4.8152	0.0001
17	2	5.3477	5.3055	0.0422	5.3056	5.3055	0.0001

14. Try to plot the graph includes the actual and the approximate results.



Any questions?

CHAPTER 8 : EXCEL VISUAL BASIC OF APPLICATION

8.1 INTRODUCTIONS

Hello and welcome to course for Excel VBA. This is a free beginners programming course, and assumes that you have no experience with coding. In this course you'll learn how to create your own macros, so that you can bring your Excel skills to the next level, and put yourself ahead of the competition.

8.1.1 What is Excel VBA?

VBA stands for Visual Basic for Applications. It's a programming language that enables you to control just about everything in Excel. You'll learn how to create Macros that can be run from things like a button on a spreadsheet, the Excel Ribbon - in fact, lots of places. Learning Excel VBA will enable you to do a lot more with the software than you can via the normal spreadsheet view.

8.1.2 Course Requisites

You don't need to be an Excel Guru to take this course, just average Excel skills will do. If you're new to Excel then this is not the place to start.

8.1.3 What software do I need?

You need Excel 2007 or greater to do this course. You can probably follow along if you have a version of Excel earlier than this, but the screenshots are all from later versions.

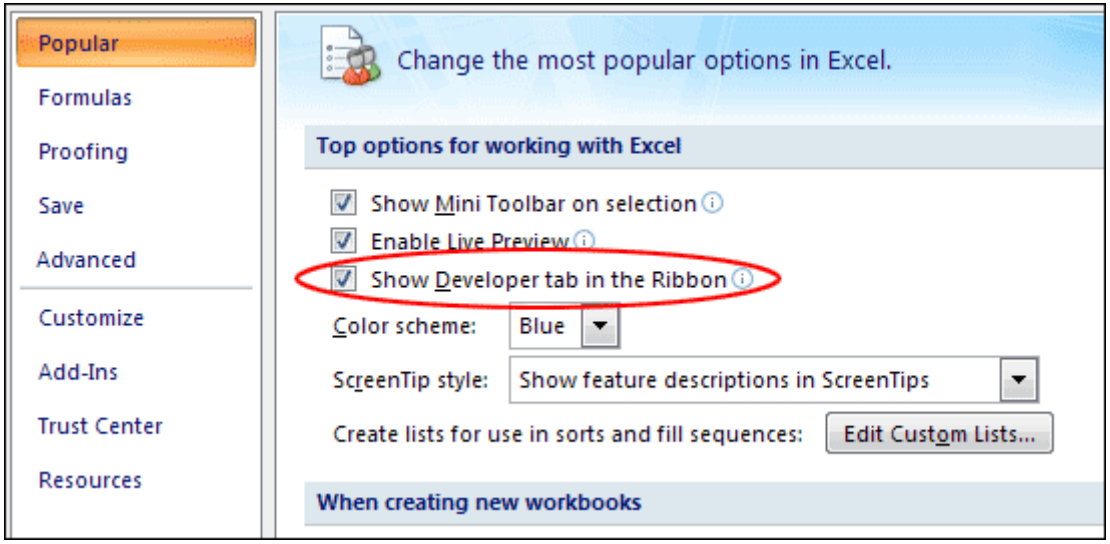
8.1.4 Do I need any programming skills?

No. We take you through the very basics of programming, so don't worry if you have no skills at all in this area.

8.2 How to Add the Developer Toolbar to Excel

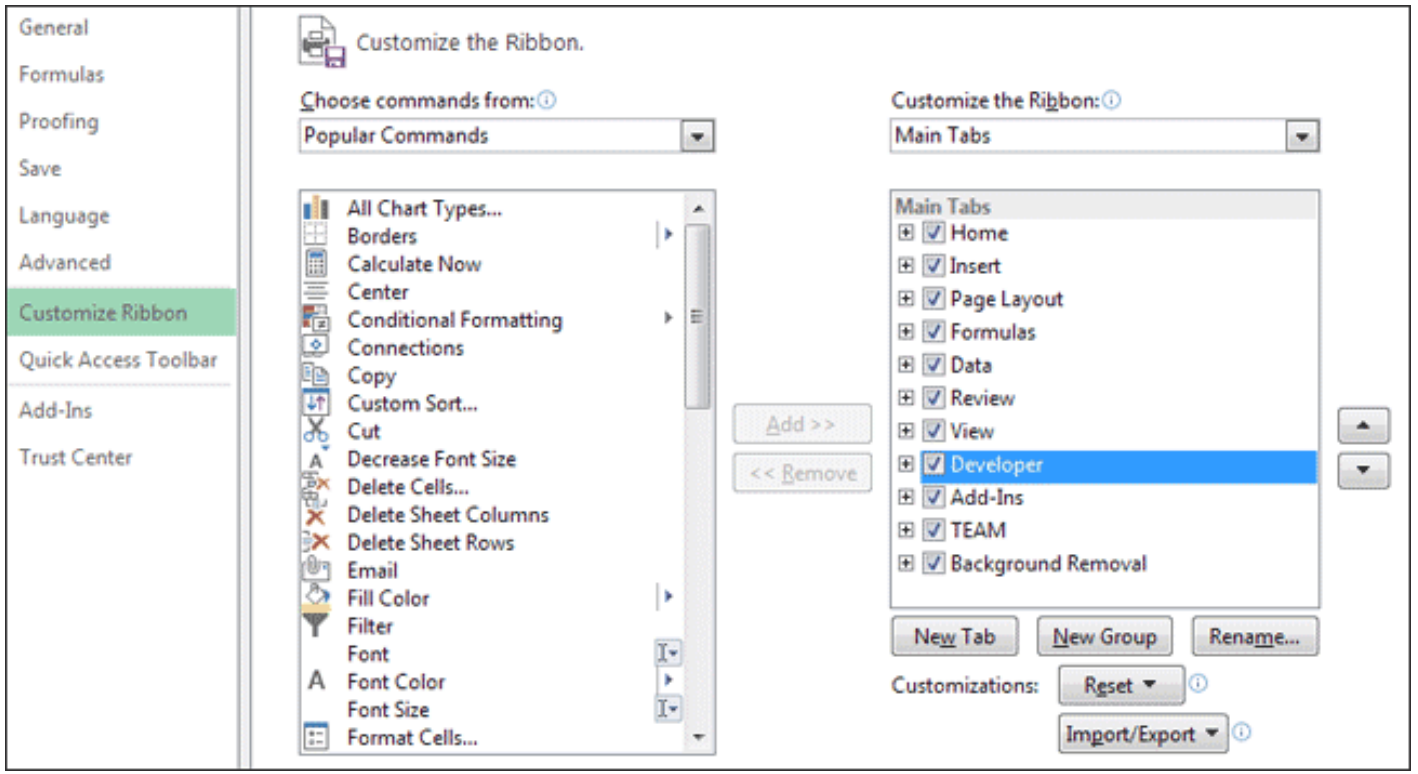
Before you can make a start, you need to add the Developer ribbon to the top of Excel.

If you have Excel 2007, click the round Office button, then click Excel Options at the bottom. When you get the Options dialogue box up, click on Popular from the left in 2007. In the section labelled "Top options for working with Excel" check the box for "Show Developer tab in the Ribbon":

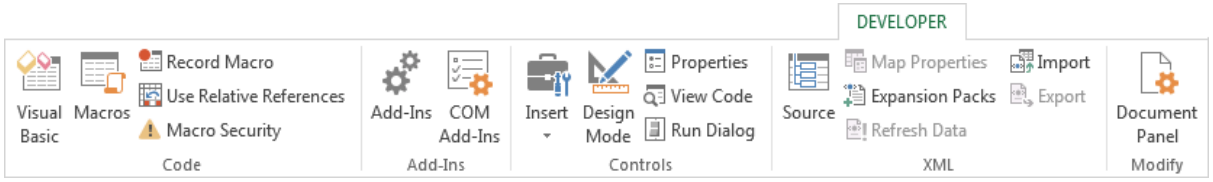


In Excel 2010 and 2013 click the **File** menu then select **Options**. From the dialogue box, click on **Customize Ribbon** on the left side. From the right hand side you'll then see an area called "Customize the Ribbon". Under "Main Tabs" check the box for **Developer**:

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.



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):



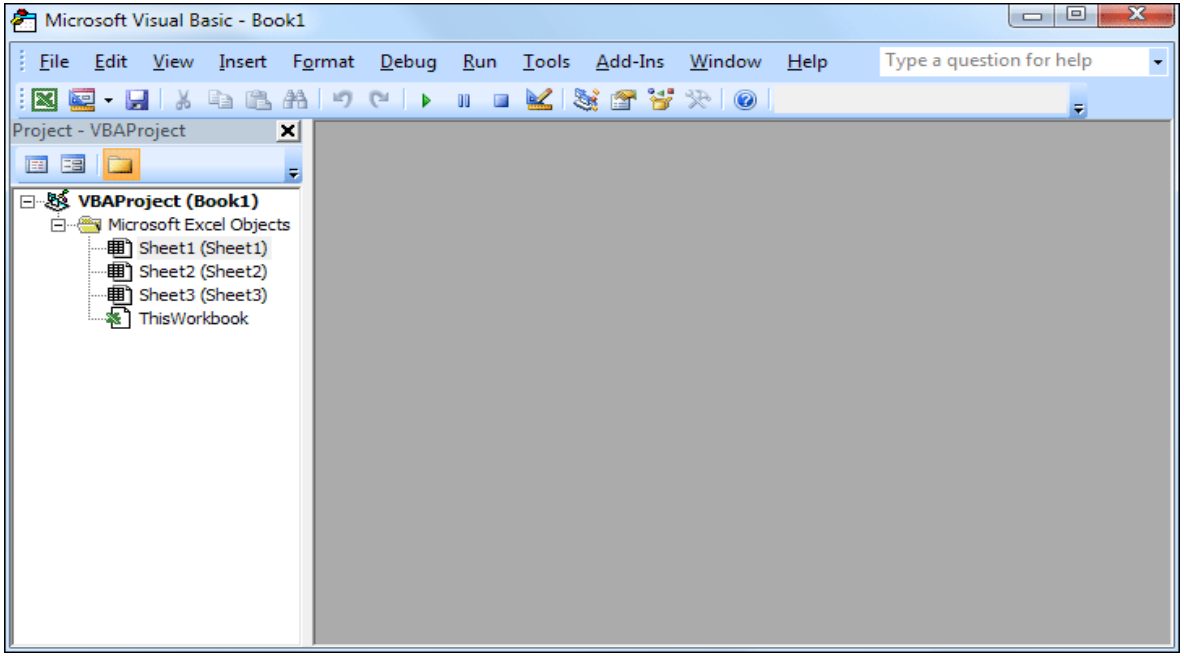
In order to run macros without any annoying security warnings, click on **Macro Security**, on the **Code**panel. Select the option for **Enable all macros**. Then make sure that "Trust access to the VBA object model" is checked:

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.

8.3 The Excel Visual Basic for Applications Development Environment

There are a few ways to open up the VBA Editor in Excel. From the Developer tab, on the Code panel, you can click the **Visual Basic** button. On the **Controls** panel of the Developer tab, you can click **View Code**. A keyboard shortcut is to hold down the left ALT key on your keyboard. Keep it held down and press the F11 key.

Whichever method you choose you should see a screen like this one:

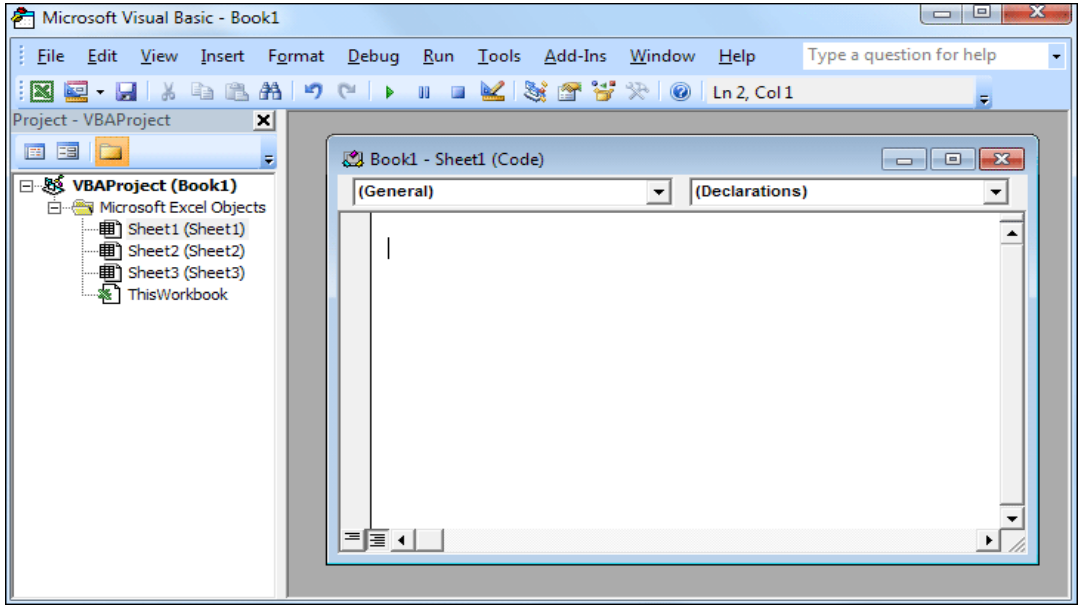


It's a little daunting at first glance. But the more your work with the editor the less daunting it will become.

There are a few things to notice about the editor. Firstly, there's an area on the left called **Project - VBA Project**. In the white area are all the object that your project currently has (we'll get into what object are a little later). By default, there are three Sheet objects: Sheet1, Sheet2, and Sheet3 (Excel 2013 only has 1 worksheet by default, so you only see Sheet1 in this version). These obviously represent the Excel worksheets. The fourth object is called ThisWorkbook and refers to the workbook where all your current macros are. There are also individual items for each worksheet.

One other object not visible above is called the Personal Macro Workbook. This workbook will appear as soon as you record a macro. The Personal Workbook can be used to store macros that you use quite frequently. They will be available whenever you close down Excel and open up a new workbook. The other sheet objects are specific to a particular workbook.

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 double click an object on your left. Double click the Sheet1 object and you'll see this:



All the macros you'll write need to be typed here, on this white screen. Because we double-clicked Sheet1 the code written here will only affect the worksheet called Sheet1. Likewise, if you add code to Sheet2 it will be run in the worksheet called Sheet2.

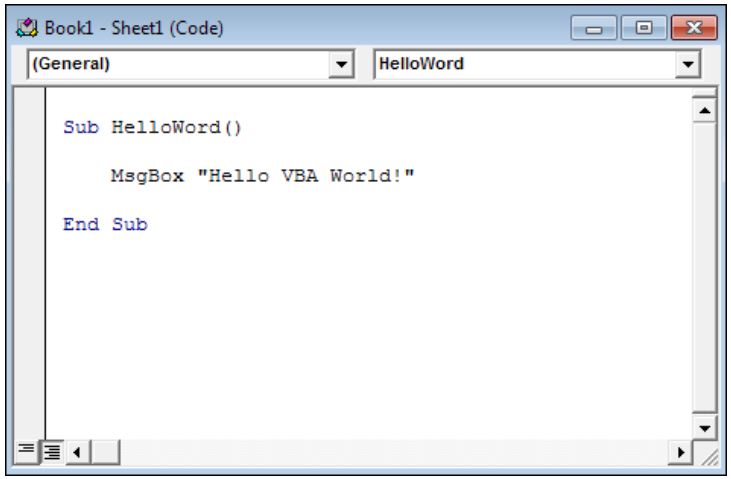
To give you a quick idea of what VBA code looks like, add the following in the white area for Sheet1:

Sub HelloWord()

MsgBox "Hello VBA World!"

End Sub

Your coding window will then look like this:



The **Sub** at the start of our code is short for Subroutine. A Subroutine is just a chunk of code that does a particular job. It has a corresponding **End Sub** to show where the code chunk ends. (What you should have noticed, though, is that as soon as you typed the first line and hit the Enter key VBA adds the End Sub for itself.)

A Sub needs a name followed by a pair of round brackets. There is a space between Sub and the name. In between Sub and End Sub is where you add the code for your macro. You can have practically anything you like as a name for your Subroutines. But try to keep them related to what the code will be doing. If we'd give our Sub the name Colin, for example, it would be a bit odd, to say the least. The Name HelloWorld describes what the code will do, however.

Subroutines can't contain spaces, though. But you can type an underscore. So this is OK:

Sub Hello_World()

But this is not:

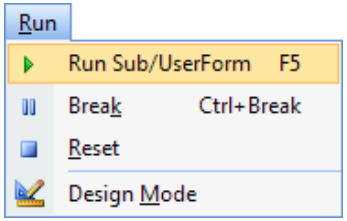
Sub Hello World()

Take note of the following when coming up with a name for your Subroutines:

- They can't start with a number, only alphabetical characters (you can have numbers elsewhere in your names, though)
- You can't have full stops/periods in them
- You can't use any of the following characters anywhere in your names: #, $, %, &, !

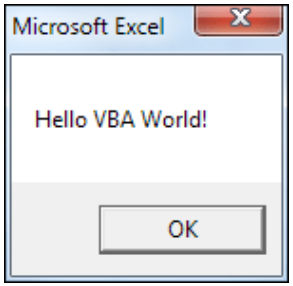
Once you've added the code, it's time to run it. To run your code, have a look at the toolbar at the top of the editor. Locate and click the green triangle:

Another way to run your code is to click the **Run** menu. From the Run menu, select **Run Sub/User Form**.



A shortcut to run your code is to press the F5 key on your keyboard.
What you should see when the code is run is this:

:



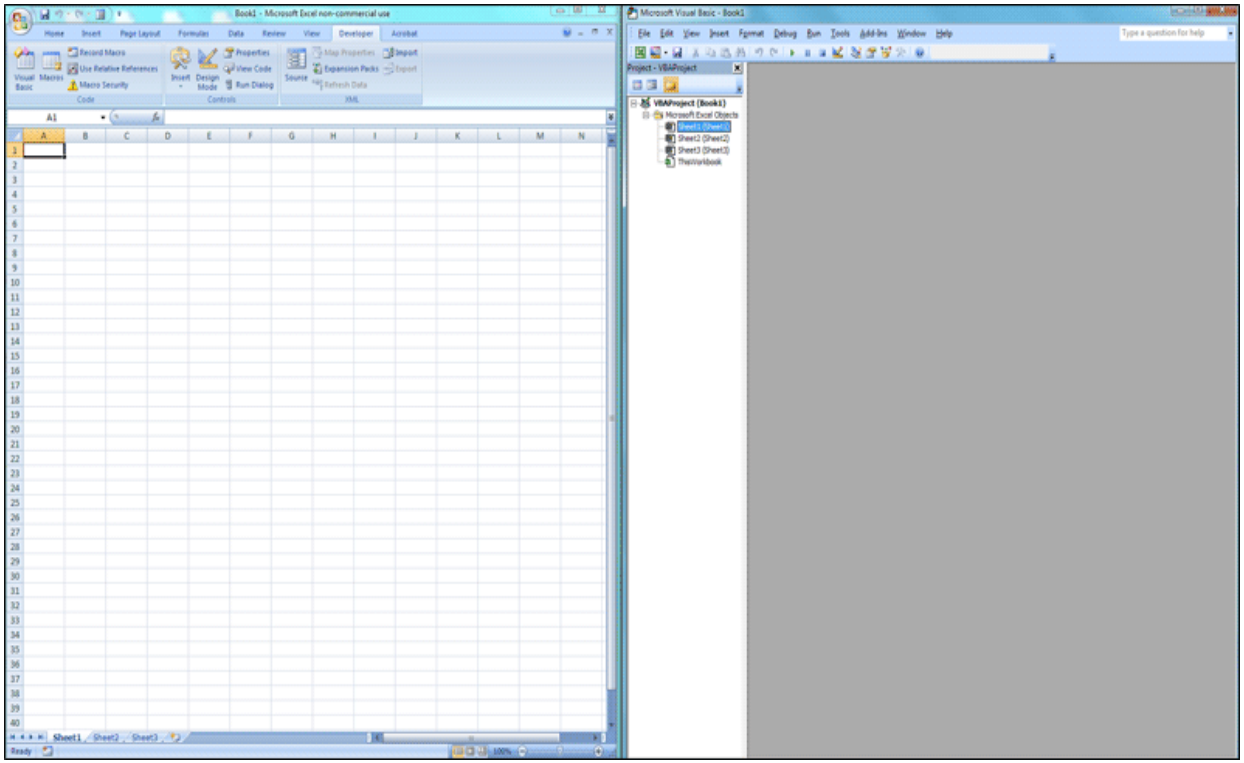
The **MsgBox** stands for Message Box. In between double quotes, we type what we wanted the message box to display.

In the next part, you'll use see how to record a Macro and watch what happens when Excel adds the code for you.

8.4 Watch as Excel records a Macro

You can watch as Excel records a macro. That way you'll get a feel for how things work. However, Excel tends to overdo it when it adds code, so don't think this is the type of code you'll have to write. Your own code will be much shorter!

First, make sure you have Excel and the coding window open side by side. Like this:



If you have Windows 7 or greater you can easily have two programmes open side by side by doing the following:

- Click on Excel to activate it
- Hold down the Windows key on your keyboard (the Windows key is between the left CTRL and left ALT key)
- Keep it held down and press the left arrow key
- Click on the coding window to activate it
- Hold down the Windows key on your keyboard
- Keep it held down and press the right arrow key
- Both Excel and the Editor should now be side by side, as above

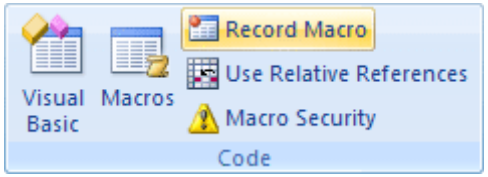
If the above doesn't work for you, then just position the two windows as best you can.

But we can now record a macro and watch as Excel does its thing.

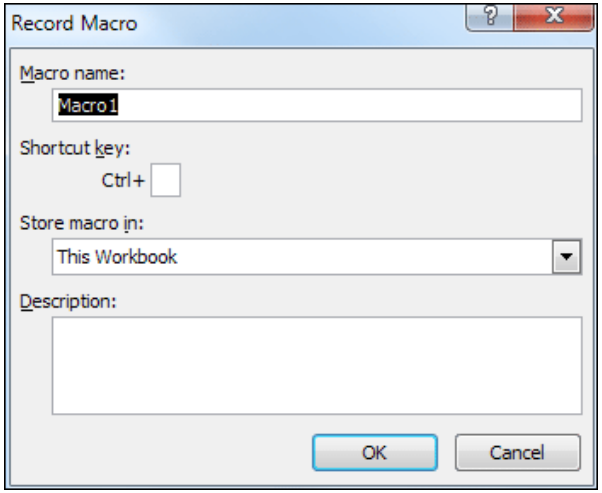
We'll record a simple macro that right aligns text in a cell. So enter the letters A to F in the A column of Excel:

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

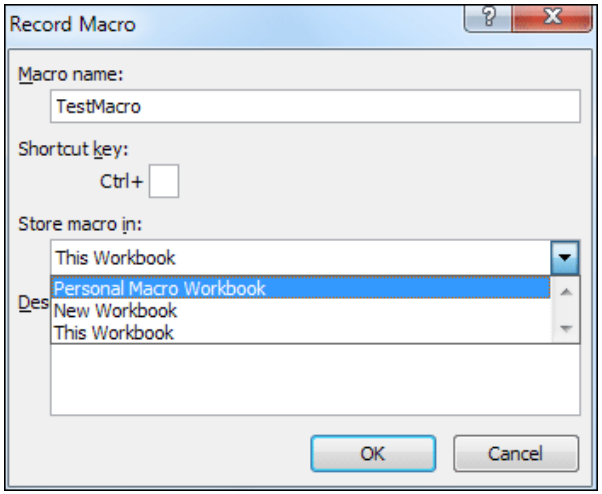
On the Developer tab, locate the **Code** panel and click **Record Macro**:



The Record Macro dialogue box should appear:



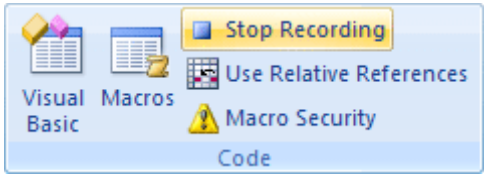
For the Macro Name, type **TestMacro**. Click the dropdown box for **Store macro in** and select**Personal Macro Workbook**:



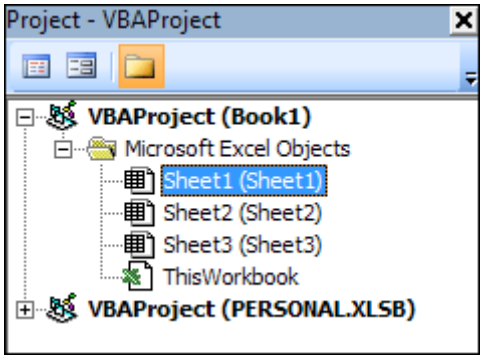
Click OK.

To record the Macro, select the cells A1 to A6. Click on the **Home** tab on the Excel ribbon. Locate the**Alignment** panel and click the right-align option:

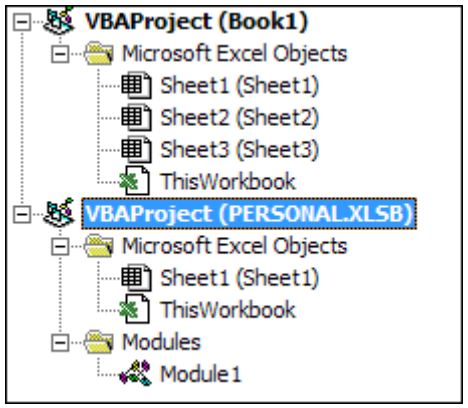
Switch back to the Developer tab. On the **Coding** panel, click **Stop Recording**:



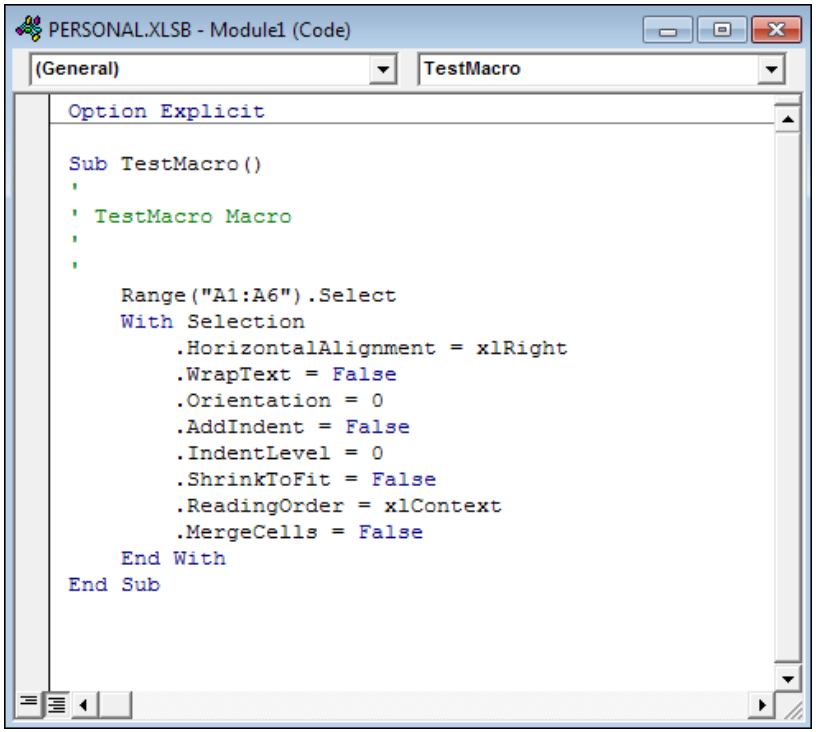
Now have a look at the coding window. If you haven't yet recorded a macro you should see a new item appear in the object window, just below your sheet objects:



The new object is called **Personal.XLSB**. Click the plus symbol to expand the entry and you'll see some yellow folders. Expand these as well:



Double click **Module1** and you'll see the code for the Macro you've just recorded:



(Don't worry if your coding window doesn't have the Option Explicit at the top. We'll explain about this in the variables chapter, later.)

Excel has created a Sub of its own. You don't need to understand all the code at this early stage. But you can probably pick out a few things that make sense. The cells A1 to A6 appear on the first line (the green lines are comments and will be ignored). This happened when you selected them. With the selection, Excel has added some formatting. The one that makes sense is **xlRight** for the horizontal alignment.

All those other lines highlight the fact that Excel tends to add lots of code unnecessarily. When you get some experience writing VBA code you'll spot lots of ways you can reduce the code Excel writes. For example, the whole of the macro above could have been replaced with just one line. This

Range("A1:A6").HorizontalAlignment = xlRight

We'll explore Ranges in the next section. But this single line of code is more readable - it reduces Excel's code to just the bare minimum.

Now that you've recorded a macro, click the **File** menu in the VB Editor. From the File menu, select **Save Personal.XLSB**.

Let's try another macro. With this one, we'll just select the text and make it bold. Keep Excel and the Coding Editor open side by side.

Return to Excel. Click on the Developer tab again, and click **Record Macro** on the **Code** panel. You should see the Record Macro dialogue box appear. Change the name from **Macro2** to **MakeBold**. Leave **Store macro in** on Personal Macro Workbook. Click OK.

Highlight cells A1 to A6 again. Click on the **Home** tab in the Ribbon. From the **Font** panel, click the**Bold** icon.
Keep an eye on the coding window when you do all this. You should see Excel automatically adding the code for you Sub. Something like this:

```
Range("A1:A6").Select
Selection.Font.Bold = True
```

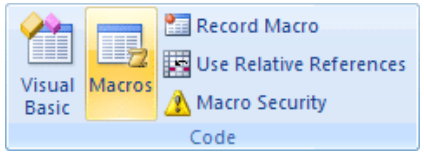
Stop the recording by clicking on the Developer tab, then the Code panel.

Exercise

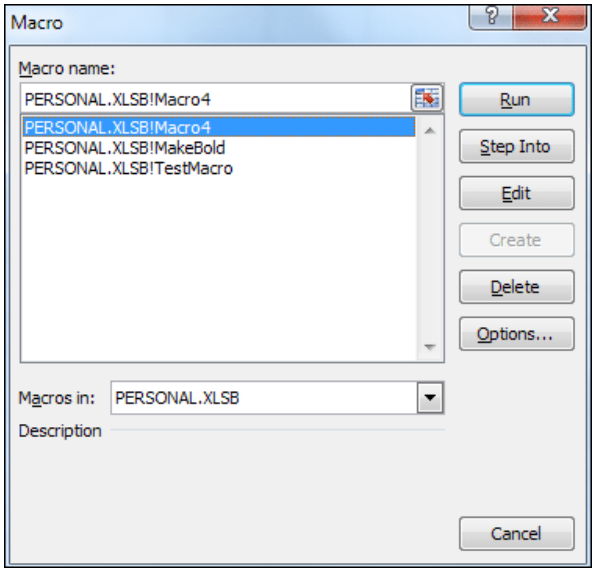
Record a new macro to make the text in cells A1 to A6 a different font, a different font size, and a different colour.

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 - your own VBA code will be a lot shorter!

Before we move on, it's best to delete all these macros. To do that, click on the Developer tab. From the Code panel, click on **Macros**:



You should see the following Macros dialogue box appear:



If you get an error message about Excel not being able to delete macros while the Personal work book is hidden, click **Cancel** on the dialogue box. From Excel, click on the **View** ribbon. Locate the**Window** panel and click **Unhide**:

From the Macro dialogue box, though, select a macro on the left and the click the **Delete** button. Do the same for all the macros on the list. Save your Personal workbox in the editor again (**File > Save**).

We're going to take a look at something called the Range object. Before we get onto Ranges, though, let's take a closer look at how the Excel dot notation works. We'll do that in the next section below.

8.5 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

Methods can come with extra settings, called parameters. A parameter of the buy method could be PaymentType. The PaymentType would then come with its own values (credit card, cash, cheques, etc). We could represent all this as follows:

tv.buy PaymentType:=Cash

We have a space between the method (buy) and the parameter (PaymentType). The value for the parameter comes after a colon and equal sign (:=), with no spaces in between.

We could add more parameters for buy. For example, we could have a Discount parameter, and a DeliveryCharge parameter:

tv.buy PaymentType:=Cash Discount:=No DeliveryCharge:=No

Notice that we've used a space to separate the three parameters and their values.

So, back to Excel. In the VBA programming language, you'll uses these object, methods, properties and parameters a lot. As an example, there's an object called ActiveCell. This is the cell where your cursor currently is. The ActiveCell object can have a font set for it:

ActiveCell.Font

Fonts have a name property all of their own. So after another dot, you type the Name property:

ActiveCell.Font.Name

Because it's a property, you need a value for it. Let's add the name of a font:

ActiveCell.Font.Name = "Times New Roman"

This sets "Times New Roman" to be the font of the ActiveCell.

We can also set a bold value for the Font:

ActiveCell.Font.Bold = True

Again, we have an object called ActiveCell. This is followed by the Font property. The Font property has a bold property of its own. This is set to True.

Methods

An example of an object method is **Quit**:

Application.Quit

To Quit is to do something, which is why it's a method rather than a property. If you're not sure whether something is a method or a property, try putting the word "to" before it. To Quit makes sense, but to Font doesn't. So Font would not be a method.

Another example of a method is **Add**. You can use this to add a new worksheet to your Excel workbook:

Worksheets.Add After:=Worksheets(1)

The object above is **Worksheets**. The **Add** method comes after a dot. Next, we have a space. One parameter of the Add method is called **After**. The value for **After** is the name of a worksheet. (Inside the round brackets you can have either the name of a worksheet in double quotes, or a number. The number 1 means worksheet 1 in the current workbook.)

8.5.1 The Range Property

Quite a lot of the macros you'll write will need to reference the **Range** property. But what is a Range?

A Range just means a range of cells on a worksheet. A Range can be an individual cell, or a group of cells. The Range you want to refer to goes between round brackets. Inside the round brackets you surround your Range of cells with double quotes. Here's a Range object that just refers to the cell A1:

Range("A1")

And here's a Range property that refers to a group of cells:

Range("A1:B7")

Notice the semicolon separating the two cell references, and no spaces between the two. The first cell reference is the top left cell, while the second cell reference is the bottom right cell in your selection - a square, in other words.

Another way to refer to more than one cell is to separate the two cell references with a comma. Each cell is then surrounded with double quotes. Like this:

```
Range("A1", "B7")
```

The above Range property refers to the cells A1 to B7.

Once you have a Range you can do something with it. One method you can use with Ranges is the **Select** method. As its name suggest, the method Selects a Range of cells:

```
Range("A1").Select

Range("A1:B7").Select
```

8.5.2 The Worksheets Object

We mentioned above that Range is a property. But what is it a property of? Well, Range is a property of the **Worksheets** object. If we wanted to be really specific we should have used the Worksheets object first, then the Range property. Like this:

```
Worksheets("Sheet1").Range("A1").Select
```

Here, the Worksheet comes first. In between round brackets and quote marks you type the name of a worksheet, "Sheet1" in this case. You can also type a number between the round brackets:

```
Worksheets(1).Range("A1").Select
```

The 1 above refers to the first worksheet in your open spreadsheet.

When you use Range by itself, Excel takes it mean the currently active worksheet, the one you currently have selected and displayed. If you want to refer to a different worksheet, say Sheet 2, you need to refer to it by name:

Worksheets(2).Range("A1").Select

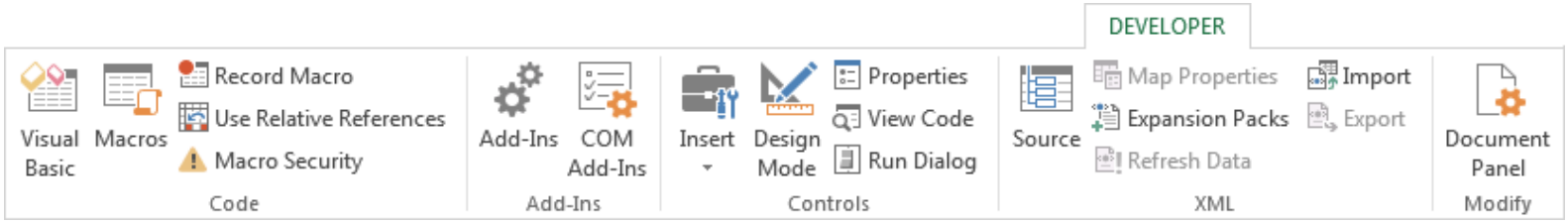
Or

Worksheets("Sheet2").Range("A1").Select

In the next part, we'll explore the VBA Developer toolbar in more depth.

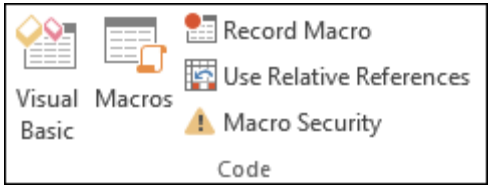
8.6 The VBA Developer Toolbar

Start a new spreadsheet for this. Now click on your Developer toolbar at the top of Excel. You should see this:



The above image is taken from Excel 2013. If you have an earlier version, you may not have the Add-Ins and Modify panels. But that's OK because we won't be using these.

The first panel to explore is the one on the left - **Code**:



Part 1: Getting Started with Excel VBA Programming

1.1 What I assume about you?

- ✓ You have access to a PC at work – and probably at home.
- ✓ You are running Excel 2010 or greater.
- ✓ You use Excel frequently in your work. If you are new to Excel then this is not the place to start.
- ✓ You have little or no programming experience.

1.2 What is VBA?

- ✓ VBA which stands for Visual Basic for Applications, is a programming language developed by Microsoft.
- ✓ VBA is the tool that people like you and me use to develop programs that control Excel.
- ✓ Learning Excel VBA will enable you to do a lot more with the software than you can via the normal spreadsheet view.

1.3 What can you do with VBA?

You are probably aware that people use Excel for thousands of different tasks. Here are just a few examples:

- ✓ Analyzing scientific data
- ✓ Budgeting and forecasting
- ✓ Creating invoices and other forms
- ✓ Developing charts from data
- ✓ and so on ……

The list could go on and on. The VBA can help you to automate some aspect of Excel. That is what VBA is all about.

For example, you might create a VBA program to import some numbers and then format and print your month-end sales report. After developing and testing the program, you can execute the macro with a single command, causing Excel to automatically perform many time-consuming procedures. Rather than struggle through a tedious sequence of commands, you can click a button and kill some time while your macro does the work.

Briefly, these are some common uses for VBA macros:

- Inserting a bunch a text - *For example, you might develop a macro that automatically types a list of all salespeople who work for your company.*
- Automating a task you perform frequently
- Automating repetitive operations - *If you need to perform the same action on, say, 12 different Excel workbooks, you can record a macro while you perform the task on the first workbook and then let the macro repeat your action on the other workbooks.*
- Creating a custom command
- Creating a custom button - *You can customize your Quick Access toolbar with your own buttons that execute the macros you write.*
- Developing new worksheet functions - *Although Excel includes hundreds of built-in functions (such as SUM and AVERAGE), you can create custom worksheet functions that can greatly simplify your formulas.*

- Creating custom add-ins for Excel - *You're probably familiar with some of the add-ins that ship with Excel. For example, the Analysis ToolPak is a popular add-in. You can use VBA to develop your own special-purpose add-ins.*
- Creating complete, macro-driven applications - *If you're willing to spend some time, you can use VBA to create large-scale applications complete with a custom Ribbon tab, dialog boxes, on-screen help, and lots of other accoutrements.*

1.4 Advantages of VBA

You can automate almost anything you do in Excel. To do so, you write instructions that Excel carries out. Automating a task by using VBA offers several advantages:

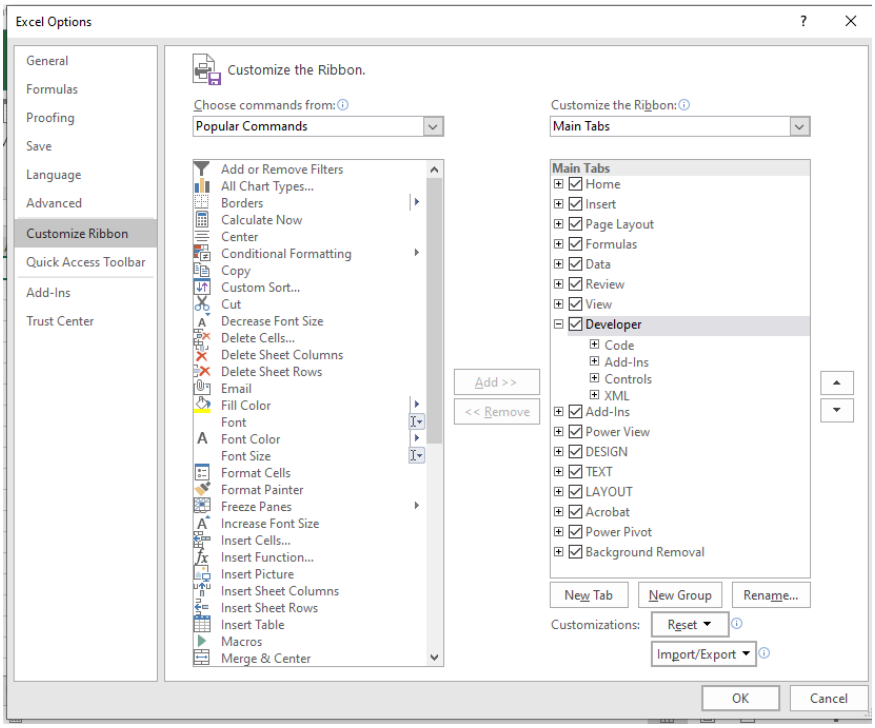
- ✓ Excel always executes the task in exactly the same way.
- ✓ Excel performs the task much faster than you can do it manually.
- ✓ If you're a good macro programmer, Excel always performs the task without errors.
- ✓ If you set things up properly, someone who doesn't know anything about Excel can perform the task.
- ✓ For long, time-consuming tasks, you don't have to sit in front of your computer and get bored. Excel will do the work for you.

1.5 Disadvantages of VBA

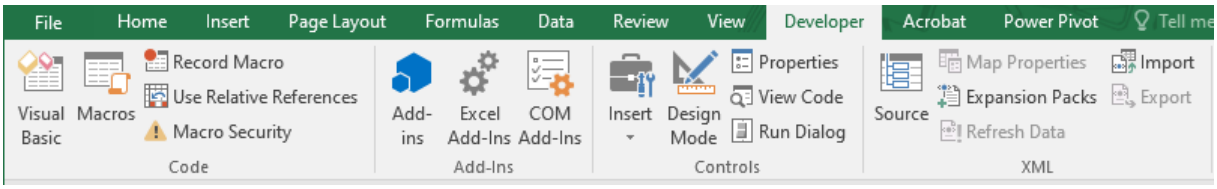
- ✓ You have to know how to write programs in VBA.
- ✓ Other people who need to use your VBA programs must have their own copies of Excel. It would be nice if you could press a button that transforms your Excel/VBA application into a stand-alone program, but that isn't possible.
- ✓ Sometimes, things go wrong. In other words, you can't blindly assume that your VBA program will always work correctly under all circumstances.
- ✓ You may discover that the VBA code you've written doesn't work properly with older versions or with a future version of Excel.

1.6 Adding the Developer toolbar to Excel

Before you can make a start, you need to add the Developer ribbon to the top of Excel. If you have Excel 2016, click the File, then click Excel Options at the bottom. When you get the Options dialogue box up, click on Customize Ribbon on the left side. From the right hand side you'll see an area called "Customize the Ribbon". Under "Main Tabs" check the box for Developer.



When you have the developer toolbar, you will see the following tab in the Ribbon.



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

Macro security is a key feature in Excel. The reason is that VBA is a powerful language — so powerful that it’s possible to create a macro that can do serious damage to your computer. A macro can delete files, send information to other computers, and even destroy Windows so that you can’t even start your system.

NOTE: If you are worried about macro security then you can always bring up this box 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.

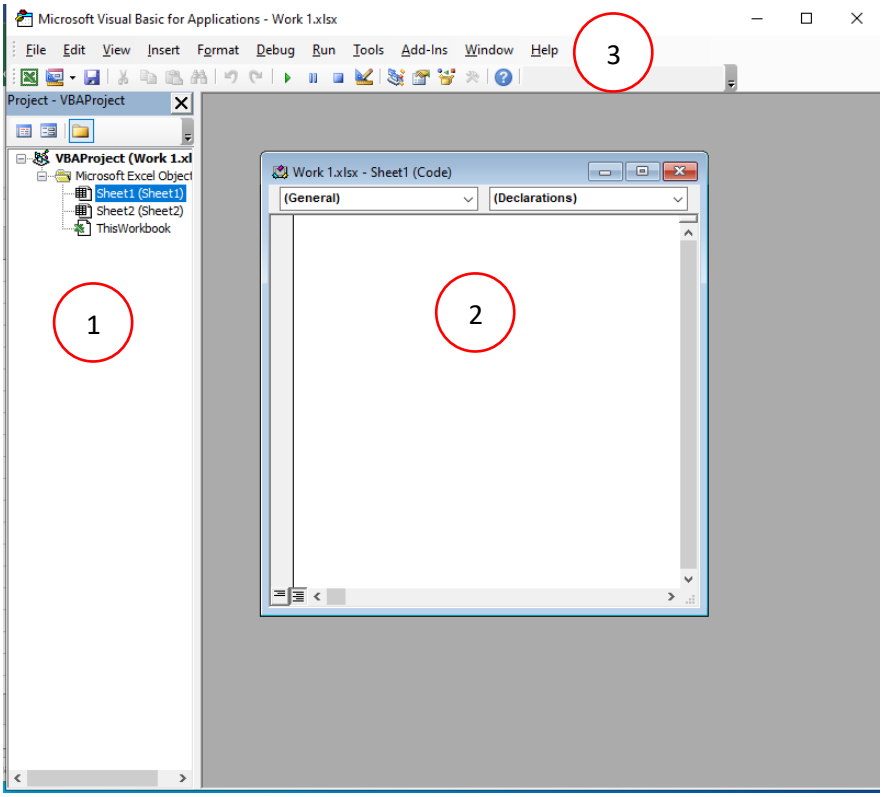
Excel can’t save macros in a file that ends in .xlxs. You need to change the file ending. To do that, click the Save As Type dropdown list. From the list, select **Excel Macro-Enabled Workbook(*.xlsm).**

1.7 The Excel VBA development environment

There are a few ways to open up the VBA Editor (VBE) in Excel.

1. From the Developer tab, on the Code panel, you can click the Visual Basic button.
2. On the Controls panel of the Developer tab, you can click View Code.
3. You can also you a keyboard shortcut. The keyboard shortcut is Alt + F11.

Whichever method you choose you should see a screen like this one:



(1) **Project window**; The Project window (also known as the Project Explorer window) contains a list of all workbooks and add-ins that are currently open. Each project is arranged as a *tree* and can be expanded (to show more information) or contracted (to show less information). For now, there are two Sheet objects: Sheet1 and Sheet2. These obviously represent the Excel worksheets. The fourth object is called ThisWorkbook and refers to the workbook where all your current macros are.

(2) **Code Window**; All the macros you'll write need to be typed here, on this white screen. Because we double-clicked Sheet1 the code written here will only affect the worksheet called Sheet1. Likewise, if you add code to Sheet2 it will be run in the worksheet called Sheet2.

(3) **Menu bar**; The VBE menu bar works just like every other menu bar you've encountered. It contains commands that you use to do things with the various components in the VBE. You also find that many of the menu commands have shortcut keys associated with them.

1.8 Sheet vs Code vs ThisWorkbook Modules

There are actually 5 different modules where we can store VBA code in a workbook.

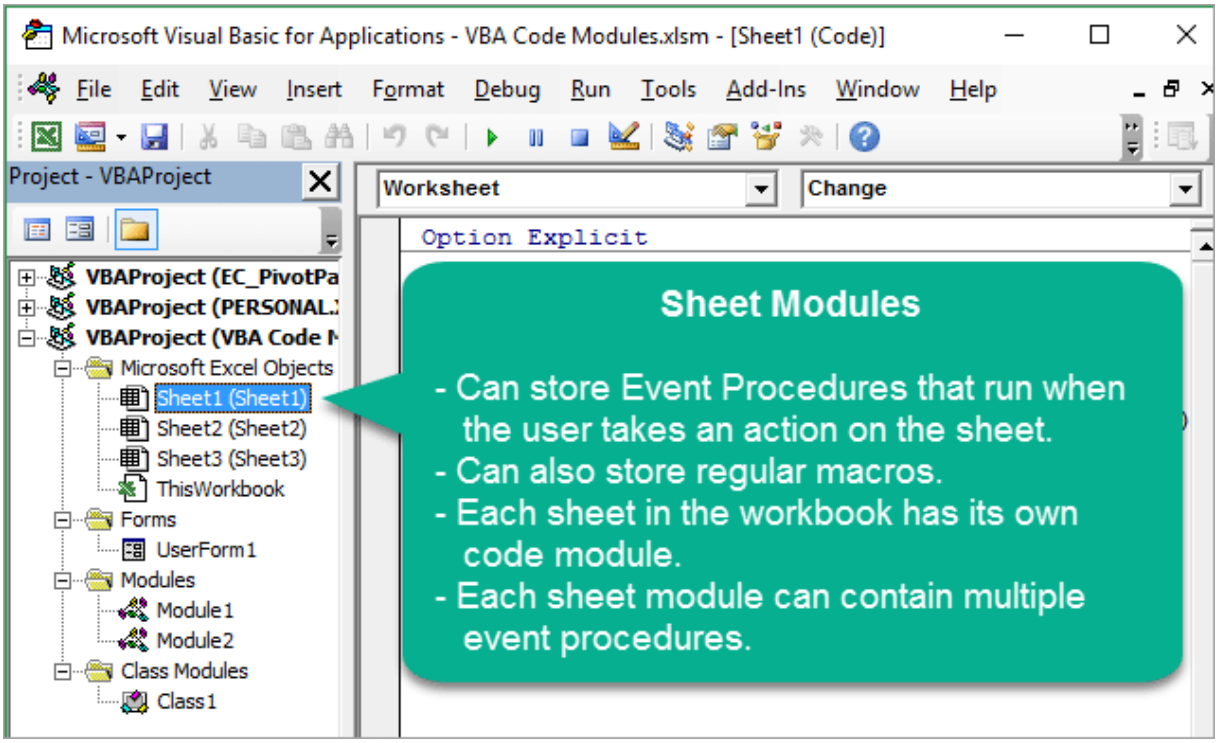
(1) Code modules - The code modules are the most common place we store macros. The modules are located in the Modules folder within the workbook.

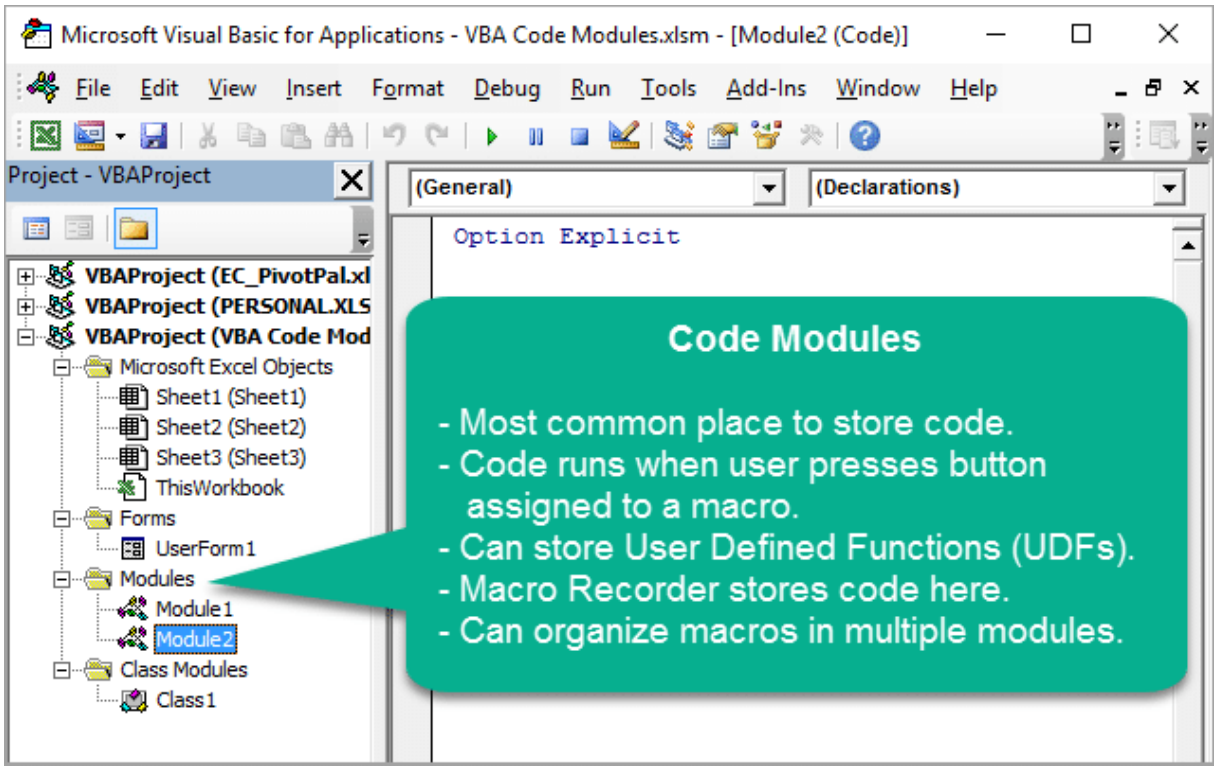
(2) Sheet modules - Each sheet in the workbook has a sheet object in the Microsoft Excel Objects folder. Double-clicking the sheet object opens its code module where we can add event procedures (macros). These macros run when the user takes a specific action in the sheet.

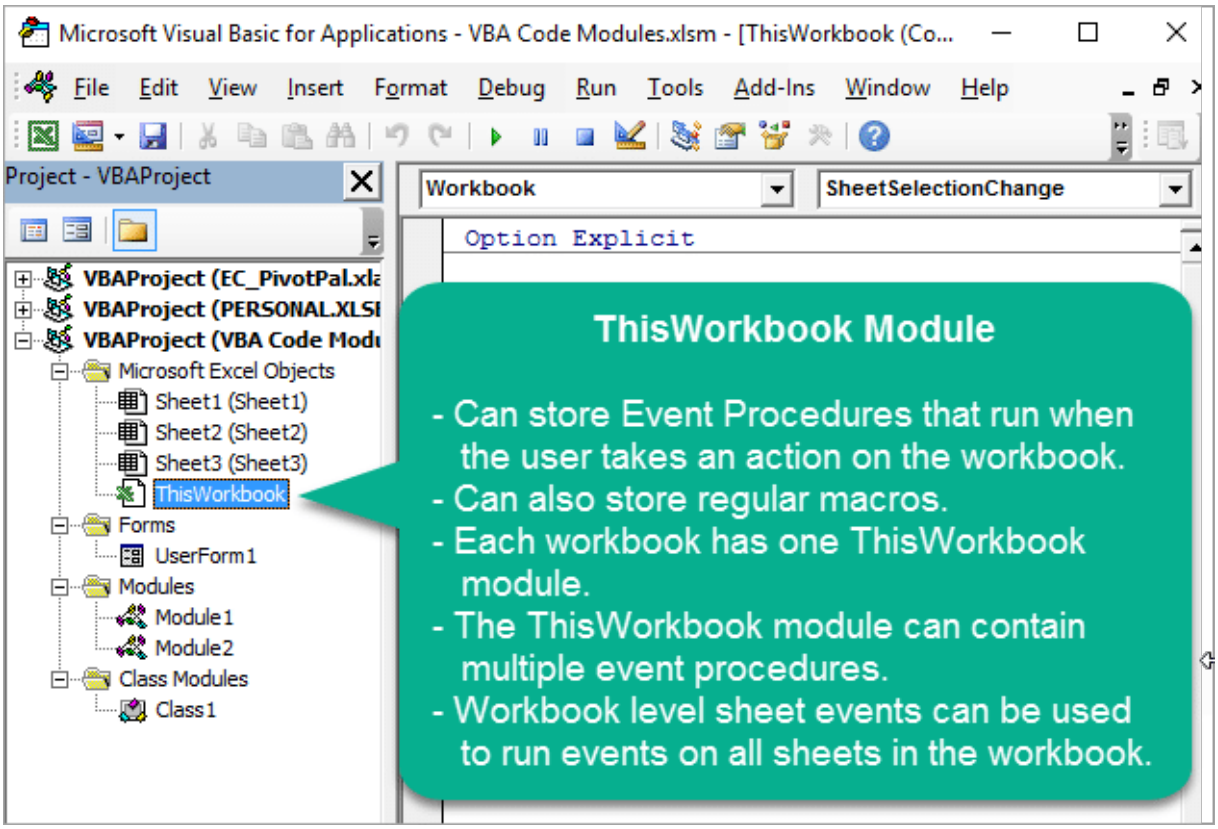
(3) ThisWorkbook module - Each workbook contains one ThisWorkbook object at the bottom of the Microsoft Excel Objects folder. We can event-based macros that run when the user takes actions in/on the workbook.

(4) Userforms - Userforms are interactive forms or windows where we can add controls like drop-down menus, list boxes, check boxes, buttons, etc. Each userform is stored in the Forms folder and has a code module where we can put macros that will run when the form is open, and/or the user interacts with the controls on the form.

(5) Class modules - Classes are stored in the Class Modules folder and allow us to write macros to create objects, properties, and methods. Classes can be used when we want to create custom objects or collections that don't exist in the Object Library.




Microsoft Visual Basic for Applications - VBA Code Modules.xlsm - [Module2 (Code)]
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
Project - VBAProject
(General)
(Declarations)
Option Explicit
VBAProject (EC_PivotPal.xl
VBAProject (PERSONAL.XLS
VBAProject (VBA Code Mod
Microsoft Excel Objects
Sheet1 (Sheet1)
Sheet2 (Sheet2)
Sheet3 (Sheet3)
ThisWorkbook
Forms
UserForm1
Modules
Module1
Module2
Class Modules
Class1
Code Modules
- Most common place to store code.
- Code runs when user presses button assigned to a macro.
- Can store User Defined Functions (UDFs).
- Macro Recorder stores code here.
- Can organize macros in multiple modules.


Microsoft Visual Basic for Applications - VBA Code Modules.xlsm - [ThisWorkbook (Co...
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
Project - VBAProject
Workbook
SheetSelectionChange
Option Explicit
VBAProject (EC_PivotPal.xla
VBAProject (PERSONAL.XLSI
VBAProject (VBA Code Modu
Microsoft Excel Objects
Sheet1 (Sheet1)
Sheet2 (Sheet2)
Sheet3 (Sheet3)
ThisWorkbook
Forms
UserForm1
Modules
Module1
Module2
Class Modules
Class1
ThisWorkbook Module
- Can store Event Procedures that run when the user takes an action on the workbook.
- Can also store regular macros.
- Each workbook has one ThisWorkbook module.
- The ThisWorkbook module can contain multiple event procedures.
- Workbook level sheet events can be used to run events on all sheets in the workbook.

In general, a VBA module can hold three types of code:

- **Declarations:** One or more information statements that you provide to VBA. For example, you can declare the data type for variables you plan to use, or set some other module-wide options. Declarations are basically housekeeping statements. They aren't actually executed.
- **Sub procedures:** A set of programming instructions that, when executed, performs some action.
- **Function procedures:** A set of programming instructions that returns a single value (similar in concept to a worksheet function, such as SUM).

A single VBA module can store any number of Sub procedures, Function procedures, and declarations.

How you organize a VBA module is completely up to you. Some people prefer to keep all their VBA code for an application in a single VBA module; others like to split up the code into several different modules. It's a personal choice, just like arranging furniture.

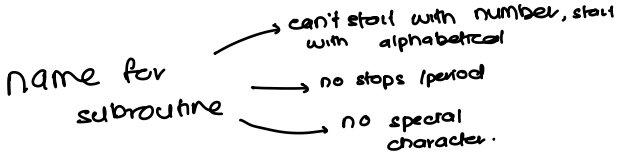
To give you a quick idea of what VBA code looks like, add the following in the Code Window:

```
Sub HelloWorld( )
    MsgBox "Hello VBA World!"
End Sub
```

- The **Sub** at the start of our code is short for Subroutine. A Subroutine is just a chunk of code that does a particular job. It has a corresponding **End Sub** to show where the code chunk ends. (What you should have noticed, though, is that as soon as you typed the first line and hit the Enter key VBA adds the End Sub for itself.)
- A Sub needs a name followed by a pair of round brackets. There is a space between Sub and the name.

```
Sub MyCode( )

    'This is where you write your code.

End Sub
```

- In between Sub and End Sub is where you add the code for your macro. You can have practically anything you like as a name for your Subroutines. But try to keep them related to what the code will be doing.
- Take note of the following when coming up with a name for your Subroutines:
    - They can't start with a number, only alphabetical characters (you can have numbers elsewhere in your names, though)
    - You can't have full stops/periods in them
    - You can't use any of the following characters anywhere in your names: #, $, %, &, !



Part 2: Working in the Visual Basic Editor

2.1 Creating a Macro

A macro (also called program/code) is an automated sequence which will apply every time you play it. To create a macro, you can use the Excel macro recorder or write your own macro.

2.2 Recording the Macro

- You can watch as Excel records a macro. That way you'll get a feel for how things work. However, Excel tends to overdo it when it adds code, so don't think this is the type of code you'll have to write. Your own code will be much shorter!
- You take the following basic steps when recording a macro.
  1) **Determine what you want the macro to do**
  2) **Get things set up properly;** This step determines how well your macro works
  3) **Determine whether you want cell references in your macro to be relative or absolute.**
  4) **Click the Record Macro button in the left side of the status bar** (or choose Developer ⇨ Code ⇨ Record Macro); Excel displays its Record Macro dialog box.
  5) **Enter a name, shortcut key, macro location, and description;** Each of these items — with the exception of the name — is optional.
  6) **Click OK in the Record Macro dialog box;** Excel automatically inserts a VBA module. From this point, Excel converts your actions into VBA code. It also displays a square Stop Recording button on your status bar.
  7) **Perform the actions you want recorded by using the mouse or the keyboard.**
  8) **After you're finished, click the Stop Recording button on the status bar** (or choose Developer ⇨ Code ⇨ Stop Recording); Excel stops recording your actions.
  9) **Test the macro to make sure it works correctly.**
  10) **As an option, you might want to clean up the code by removing extraneous statements.**

2.2.1 Relative or Absolute

- When recording your actions, Excel normally records absolute references to cells. (This is the default recording mode.) But quite often, this is the wrong recording mode. If you use absolute recording mode, Excel records actual cell references. If you use relative recording, Excel records relative references to cells.
- A macro recorded by using absolute references always produces the same results when it is executed.
- In some cases, you want your recorded macro to work with cell locations in a relative manner. You may want the macro to work in the active cell. In such a case, you need to use relative recording.
- You can change the manner in which Excel records your actions by clicking the Use 'Relative References' button in the Code group in the Developer tab. This button is a toggle button. When the button appears highlighted in a different color, the recording mode is relative. When the button appears normally, you are recording in absolute mode.
- You can change the recording method at any time, even in the middle of recording.

** Exercises

Record a new macro to organize any data i.e. change the font size, color, alignment etc. Record the macro in both absolute and relative mode and observe the differences.

- You should know that when you record a Macro it records even your mistakes, and it will repeat them when you run it. If you want to solve a problem like this you have two options:
  1. Record the Macro again.
  2. Edit the VBA code.

2.2.2 Is This Thing Efficient

- You might think that recording a macro would generate some award-winning VBA code — better than you could ever write manually. Think again. In many cases, the recorder spits out lots of extraneous garbage, and it often generates code that's less than efficient.
- To demonstrate just how inefficient the macro recorder's code can be, try this:
  1) Turn on the macro recorder.
  2) Change the page layout from Portrait to Landscape.
  3) Turn off the macro recorder.
  4) Observe the macro.

- You may be surprised by the amount of code generated by this single command. Although you changed only one print setting, Excel generated code that sets many other print related properties.
- This is a good example of macro-recording overkill. If you want a macro that just switches the page setup to landscape mode, you can simplify this macro considerably by deleting the extraneous code. This makes the macro a bit faster and a lot easier to read. Here's how the macro looks after we deleted the irrelevant lines:

```
Sub Macro1( )
    ActiveSheet.PageSetup.Orientation = xlLandscape
End Sub
```

In addition, the macro recorder cannot generate code for any of the following tasks,

- Performing any type of repetitive looping
- Performing any type of conditional actions (using an If-Then statement)
- Assigning values to variables
- Specifying data types
- Displaying pop-up messages
- Displaying custom dialog boxes

In the next part, we will learn how to write your own macro.

Part 3: Programming Concepts

3.1 Variables, Constants, and Data Types

3.1.1 Variables

- In computer programming, you need to store things in memory, things like numbers and strings of text. You store them so that you can retrieve them later and manipulate the numbers or strings of text in some way.
- To store things in memory you use a **variable**. Variables have a name (something that you yourself come up with) and a type (such as Integer or String).
- VBA enforces a few rules regarding variable names:
  - ✓ You can use letters, numbers, and some punctuation characters, but the first character must be a letter. You can combine your variable with underscore character.
  - ✓ VBA does not distinguish between uppercase and lowercase letters.
  - ✓ You cannot use any spaces, periods (full stop), or mathematical operators in a variable name.
  - ✓ You cannot use the following characters in a variable name: #, $, %, &, or !.
  - ✓ Variable names can be no longer than 255 characters. But nobody even gets close to that limit.
- Examples of valid variables:
  1) x
  2) Interest_Rate
  3) Data2Entered
  4) UserName1001
- Example of invalid variables:
  1) 2MyVariable
  2) My Variable
  3) $myvariable
  4) My#variable

- VBA has many reserved words that you can't use for variable names or procedure names. These include words such as **Sub, Dim, With, End, Next**, and **For**.
- If you attempt to use one of these words as a variable, you may get a compile error (which means your code won't run). So, if an assignment statement produces an error message, double-check and make sure that the variable name isn't a reserved word.

3.1.2 Constants

A variable's value may (and usually does) change while your procedure is executing. That's why they call it a *variable*. Sometimes you need to refer to a value or string that never changes. In such a case, you need a *constant* — a named element whose value doesn't change.

As shown in the following examples, you declare constants by using the **Const** statement.

- Const Rate = 0.0725
- Const Period = 12
- Const NumQuarters As Integer = 4
- Const ModName As String = "Budget Macros"

3.1.3 Data types

- Data type refers to the manner in which a program stores data in memory – for example, as integers, real numbers, or strings.
- Although VBA can take care of these details automatically, it does so at a cost. (There's no free lunch.) Letting VBA handle your data typing results in slower execution and inefficient memory use.
- VBA has a variety of built-in data types. Table 1 lists the most common types of data that VBA can handle.

Data Type	Byte Used	Range of Values
Integer	2	-32,768 to 32,767
Long	4	-2,147,483,648 to 2,147,483,647
Single	4	-3.40 to -1.40E-45 for negative values; 1.40E-45 to 3.40 for positive values
Double	8	-1.79E308 to -4.94E-324 for negative values; 4.94E-324 to 1.79E308 for positive values
String	1 per character	Varies
Variant	Varies	Varies
Boolean	2	True or False

- The difference between the above numerical data types is how many digits they can hold, and whether or not you want a decimal point.
- For example, if you want store a whole number bigger than 32,767, then clearly **Integer** may not be enough. This is where **Long** comes in.
- Variant is used when you're not sure what value will be returned. It can hold numbers, text, and objects. However, using it too much can slow down your programs, as it uses 16 bytes of data for numbers and 22 bytes for text.

3.2 Declaring Variables in Macro

- Before you use variables in a macro, it's an excellent practice to declare your variables — that is, tell VBA each variable's data type. Declaring your variables makes your macro run faster and use memory more efficiently.
- To force yourself to declare all the variables you use, include these two words as the first statement in your VBA module (optional):

```
Option Explicit
```

- When this statement is present, you won't be able to run your code if it contains any undeclared variables.
- To ensure that the Option Explicit statement is inserted automatically whenever you insert a new VBA module, turn on the Require Variable Definition option. You find it in the Editor tab of the Options dialog box (in the VBE, choose Tools ➪ Options). Highly recommend doing so.
- You now know the advantages of declaring variables, but how do you do it? The most common way is to use a **Dim** statement. Here are some examples of variables being declared:
    - Dim YourName As String
    - Dim January_Inventory As Double
    - Dim AmountDue As Double
    - Dim RowNumber As Long, ColumnNumber As Integer
    - Dim X

- The last variable, X is not declared as a specific data type, so it's treated as a Variant.
- Besides Dim, VBA has three other keywords that are used to declare variables:
  - Static
  - Public
  - Private
- A variable's scope determines which modules and procedures can use the variable

(i) Procedure-only variables

The lowest level of scope for a variable is at the procedure level. (A *procedure* is either a Sub or a Function procedure.) Variables declared with this scope can be used only in the procedure in which they are declared. When the procedure ends, the variable no longer exists, and Excel frees up its memory. If you execute the procedure again, the variable comes back to life, but its previous value is lost.

The most common way to declare a procedure-only variable is with a Dim statement. You usually place Dim statements immediately after the Sub or Function statement and before the procedure's code.

The following example shows some procedure-only variables declared by using Dim statements:

```
Sub MySub( )
    Dim x As Integer
    Dim First As Long
    Dim InterestRate As Single
    Dim TodaysDate As Date
    Dim UserName As String
    Dim MyValue
    '... [The procedure's code goes here] ...
End Sub
```

(ii) Module-only variables

Sometimes you want a variable to be available to all procedures in a module. If so, just declare the variable (using **Dim** or **Private**) *before* the module's first Sub or Function statement — outside any procedures. This is done in the Declarations section, at the beginning of your module. (This is also where the Option Explicit statement is located.)

As an example, suppose that you want to declare the CurrentValue variable so that it's available to all the procedures in your module. All you need to do is use the Dim statement in the Declarations section

```
Option Explicit

Dim CurrentValue As Integer
Dim First As Long

Sub MySub1( )
    '... [The procedure's code goes here] ...
End Sub

Sub MySub2( )
    '... [The procedure's code goes here] ...
End Sub
```

With this declaration in place — and in the proper place — the CurrentValue variable can be used from any other procedure within the module, and it retains its value from one procedure to another.

(iii) Public variables

If you need to make a variable available to all the procedures in all your VBA modules in a workbook, declare the variable at the module level (in the Declarations section) by using the **Public** keyword. Here's an example:

```
Public CurrentRate As Long
```

The Public keyword makes the CurrentRate variable available to any procedure in the workbook — even those in other VBA modules.

(iv) Static variables

Normally, when a procedure ends, all the procedure's variables are reset. *Static variables* are a special case because they retain their value even when the procedure ends. You declare a static variable at the procedure level. A static variable may be useful if you need to track the number of times you execute a procedure. You can declare a static variable and increment it each time you run the procedure. Here's an example:

```
Sub TestMacro()
    Static Counter As Integer
    Counter = Counter + 1
    MsgBox Counter
End Sub
```

3.3 Working with Arrays

Most programming languages support arrays. An *array* is a group of variables that share a common name. You refer to a specific variable in the array by using the array name and an index number in parentheses.

3.3.1 Declaring arrays

You declare an array with a Dim or Public statement, just as you declare a 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 — all inside parentheses. The following example shows how to declare an array of 100 integers:

```
Dim MyArray(1 To 100) As Integer
```

When you declare an array, you can choose to specify only the upper index. If you omit the lower index, VBA assumes that it's 0. Therefore, both of the following statements declare the same 101-element array:

```
Dim MyArray (0 To 100) As Integer
```

```
Dim MyArray (100) As Integer
```

3.3.2 Multidimensional arrays

The arrays created in the previous examples are all one-dimensional arrays. Arrays you create in VBA can have as many as 60 dimensions — although you rarely need more than two or three dimensions in an array. The following example declares an 81-integer array with two dimensions:

```
Dim MyArray (1 To 9, 1 To 9) As Integer
```

You can think of this array as occupying a 9 x 9 matrix. Here's how to declare a three-dimensional array, with 1,000 elements:

```
Dim My3DArray (1 To 10, 1 To 10, 1 To 10) As Integer
```

3.3.3 Dynamic arrays

You can also create *dynamic* arrays. A dynamic array doesn't have a preset number of elements. Declare a dynamic array with an empty 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. Usually, the number of elements in the array is determined while your code is running. You can use the ReDim statement any number of times, changing the array's size as often as needed. The following example demonstrates how to change the number of elements in a dynamic array. It assumes that the NumElements variable contains a value, which your code calculated.

```
ReDim MyArray (1 To NumElements)
```

When you redimension an array by using ReDim, you wipe out any values currently stored in the array elements. You can avoid destroying the old values by using the **Preserve** keyword. The following example shows how you can preserve an array's values when you redimension the array:

```
ReDim Preserve MyArray(1 To NumElements)
```

If MyArray currently has ten elements and you execute the preceding statement with NumElements equaling 12, the first ten elements remain intact, and the array has room for two additional elements (up to the number contained in the variable NumElements). If NumElements equals 7 however, the first seven elements are retained but the remaining three elements meet their demise.

Part 4: Working with Range Objects

Excel is all about cells, and the Range object is a container for cells. Why do you need to know so much about Range objects? Because much of the programming work you do in Excel focuses on Range objects.

4.1 The Range Object

A Range object represents a range contained in a Worksheet object. Range objects, like all other objects, have properties (which you can examine and sometimes change) and methods (which perform actions on the object).

A Range just means a range of cells on a worksheet. A Range can be an individual cell, or a group of cells. The Range you want to refer to goes between round brackets. Inside the round brackets you surround your Range of cells with double quotes. Here's a Range object that just refers to the cell A1:

```
Range("A1")
```

And here is a Range property that refers to a group of cells:

```
Range("A1:B7")
```

The first cell reference is the top left cell, while the second cell reference is the bottom right cell in your selection - a square, in other words. Another way to refer to more than one cell is to separate the two cell references with a comma. Each cell is then surrounded with double quotes. Like this:

```
Range("A1","B7")
```

As shown in the following example, you can refer to a range outside the active sheet by qualifying the range reference with a worksheet name from the active workbook:

```
Worksheets("Sheet1").Range("A1:C5")
```

If you need to refer to a range in a different workbook (that is, any workbook other than the active workbook), you can use a statement like this:

```
Workbooks("Budget.xlsx").Worksheets("Sheet1").Range("A1:C5")
```

The following expression refers to a two-area noncontiguous range. Notice that a comma separates the two areas.

↓ notsheet

```
Range("A1:B8 , D9:G16")
```

4.2 Other Ways to Refer to a Range

4.2.1 The Cells Property

Rather than use the VBA Range keyword, you can refer to a range via the Cells property.

Notice that I wrote Cells *property,* not Cells *object* or even Cells *collection*. Although Cells may seem like an object (or a collection), it's really not. Rather, Cells is a property that VBA evaluates. VBA then returns an object (more specifically, a Range object).

The Cells property takes two arguments: a row number and a column number. Both of these arguments are numbers, even though we usually refer to columns by using letters. For example, the following expression refers to cell C2,

```
Cells(2, 3)
```

You can also use the Cells property to refer to a multi-cell range. The following example demonstrates the syntax you use:

```
Range(Cells(1, 1), Cells(10, 8))
```

This expression refers to an 80-cell range that extends from cell A1 (row 1, column 1) to cell H10 (row 10, column 8).

4.2.2 The Offset Property

The Offset property provides another handy means for referring to ranges. This property, which operates on a Range object and returns another Range object, lets you refer to a cell that is a particular number of rows and columns away from another cell.

Like the Cells property, the Offset property takes two arguments. The first argument represents the number of rows to offset; the second represents the number of columns to offset.

The following expression refers to a cell one row below cell A1 and two columns to the right of cell A1. In other words, this refers to the cell commonly known as C2:

```
Range("A1").Offset(1, 2)
```

The Offset property can also use negative arguments. A negative row offset refers to a row above the range. A negative column offset refers to a column to the left of the range. The following example refers to cell A1:

```
Range("C2").Offset(-1, -2)
```

And, as you may expect, you can use 0 as one or both of the arguments for Offset. The following expression refers to cell A1:

```
Range("A1").Offset(0, 0)
```

Here's a statement that inserts the time of day into the cell to the right of the active cell:

```
ActiveCell.Offset(0,1) = Time
```

When you record a macro in relative mode, Excel uses the Offset property quite a bit.

To manipulate these cell values, Properties and Methods are used.

- **Property**: A property stores information about the object
- **Method**: A method is an action of the object it will perform. Range object can perform actions like selected, copied, cleared, sorted, etc

4.3 The Range Property

A Range object has dozens of properties. For example, the value property, the text property, the count property, the address property and many more. The following describes some of the properties.

4.3.1 The Value Property

The Value property represents the value contained in a cell. It's a read-write property, so your VBA code can either read or change the value.

The following statement displays a message box that shows the value in cell A1 on Sheet1:

```
MsgBox Worksheets("Sheet1").Range("A1").Value
```

It stands to reason that you **can read the Value property only for a single-cell Range object**. For example, the following statement generates an error:

```
MsgBox Worksheets("Sheet1").Range("A1:C3").Value
```

You can, however, change the Value property for a range of any size. The following statement enters the number 123 into each cell in a range:

```
Worksheets("Sheet1").Range("A1:C3").Value = 123
```

Value is the default property for a Range object. In other words, if you omit a property for a Range, Excel uses its Value property. The following statements both enter a value of 75 into cell A1 on the active worksheet:

```
Range("A1").Value = 75
Range("A1") = 75
```

4.3.2 The Count Property

The Count property returns the number of cells in a range. It counts all cells, not just the nonblank cells. Count is a read-only property, just as you would expect. The following statement accesses a range's Count property and displays the result in a message box:

```
MsgBox Range("A1:C3").Count
```

4.3.3 The Column and Row Properties

The Column property returns the column number of a single-cell range. Its sidekick, the Row property, returns the row number of a single-cell range. Both are read-only properties. For example, the following statement displays 6 because cell F3 is in the sixth column:

```
MsgBox Sheets("Sheet1").Range("F3").Column
```

The next expression displays 3 because cell F3 is in the third row:

```
MsgBox Sheets("Sheet1").Range("F3").Row
```

If the Range object consists of more than one cell, the Column property returns the column number of the first column in the range, and the Row property returns the row number of the first row in the range.

4.4 The Range Method

As you know, a VBA method performs an action. A Range object has dozens of methods but, again, you won't need most of these.

4.4.1 The Select Method

Use the Select method to select a range of cells. The following statement selects a range on the active worksheet:

```
Range("A1:C12").Select
```

Before selecting a range, it's often a good idea to use one additional statement to ensure that the correct worksheet is active. For example, if Sheet1 contains the range you want to select, use the following statements to select the range:

```
Sheets("Sheet1").Activate
Range("A1:C12").Select
```

4.4.2 The Clear Method

The Clear method deletes the contents of a range, plus all the cell formatting. For example, if you want to zap everything in column D, the following statement does the trick:

```
Columns("D:D").Clear
```

You should be aware of two related methods. The **ClearContents** method deletes the contents of the range but leaves the formatting intact. The **ClearFormats** method deletes the formatting in the range but not the cell contents.

4.4.3 The Copy and Paste Methods

You can perform copy and paste operations in VBA by using the Copy and Paste methods. Note that two different objects come into play. The Copy method is applicable to the Range object, but the Paste method applies to the Worksheet object. It actually makes sense: You copy a range and paste it to a worksheet.

This short macro (courtesy of the macro recorder) copies range A1:A12 and pastes it to the same worksheet, beginning at cell C1:

```
Sub CopyRange()
    Range("A1:A12").Select
    Selection.Copy
    Range("C1").Select
    ActiveSheet.Paste
End Sub
```

Notice that in the preceding example, the ActiveSheet object is used with the Paste method. This is a special version of the Worksheet object that refers to the currently active worksheet. Also notice that the macro selects the range before copying it. However, you don't have to select a range before doing something with it. In fact, the following procedure accomplishes the same task as the preceding example by using a single statement:

```
Sub CopyRange2()
        Range("A1:A12").Copy Range("C1")
End Sub
```

This procedure takes advantage of the fact that the Copy method can use an argument that corresponds to the destination range for the copy operation. That's something that you can find out by checking with the Help system.

Part 5: Controlling Program Flow and Making Decisions

5.1 The IF-THEN Structure

Use the If-Then structure when you want to execute one or more statements conditionally. The optional Else clause, if included, lets you execute one or more statements if the condition you're testing is *not* true.

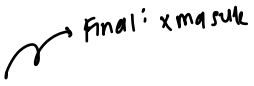
You use Conditional Logic all the time in your daily life. You say things like this:

IF I buy these shoes I will be happier
IF I eat this ice cream I will ruin my diet
IF I go online I might have some emails

The structure of a VBA IF statement looks like this:

```
If Condition_To_Test Then
        ' CODE HERE
ElseIf Condition_To_Test Then
        ' CODE HERE
Else
        ' CODE HERE
End If
```

5.2 The FOR-NEXT Loops



The simplest type of loop is a For-Next loop. The looping is controlled by a counter variable, which starts at one value and stops at another value. The statements between the For statement and the Next statement are the statements that get repeated in the loop.

This is how the syntax looks like in VBA:

```
For iteration_variable = start_value To end_value
    ' CODE HERE
Next iteration_variable
```

The following example uses a For-Next loop to sum the first 1,000 positive integers. The Total variable starts out as zero. Then the looping occurs. The variable Cnt is the loop counter. It starts out as 1 and is incremented by 1 each time through the loop. The loop ends when Cnt is 1,000.

This example has only one statement inside of the loop. This statement adds the value of Cnt to the Total variable. When the loop finishes, a MsgBox displays the summed integers.

```
Sub AddNumbers()
      Dim Total As Double
      Dim Cnt As Integer
      Total = 0

      For Cnt = 1 To 1000
           Total = Total + Cnt
      Next Cnt
```

```
    MsgBox Total
```
End Sub

Part 6: Functions

6.1 Using Built-In VBA Functions

VBA provides numerous built-in functions. Some of these functions take arguments, and some do not.

6.1.1 VBA Function Examples

- ***Date*** - Displaying the system date or time
  Example: to display the current system date in a message box

```
Sub ShowDate()
    MsgBox "Today is: " & Date
End Sub
```

- ***Len*** – Finding a string length
  Example: measure the length of a text string

```
Sub GetLength()
    Dim MyName As String
    Dim StringLength As Integer
    MyName = Application.UserName
    StringLength = Len(MyName)
    MsgBox MyName & " has " & StringLength & " characters."
End Sub
```

- ***MonthName*** – Displaying the name of a month
  Example: Return the name of a month in a message box

```
Sub ShowMonthName()
    Dim ThisMonth As Long
    ThisMonth = Month(Date)
    MsgBox MonthName(ThisMonth)
End Sub
```

  Notice that the function uses one argument.

- ***FileLen*** - Determining a file size
  Example: Display the size, in bytes, of the Excel executable file

```
Sub GetFileSize()
    Dim TheFile As String
    TheFile = Application.Path & "\EXCEL.EXE"
    MsgBox FileLen(TheFile)
End Sub
```

And the list continues ….
It will take forever to list all of them here.

How do you find out which functions VBA provides? The best source is the Excel Visual Basic Help system. Another way is to type VBA, followed by a period. You get a list of items. Those with a green icon are functions. If this feature isn't working, choose the VBE's Tools ➪ Options command, click the Editor tab, and place a checkmark next to Auto List Members.

More built-in functions here: *https://www.excelfunctions.net/vba-functions.html*

6.2 Using Worksheet Functions in VBA

Although VBA offers a decent assortment of built-in functions, you might not always find exactly what you need. Fortunately, you can also use most of Excel's worksheet functions in your VBA procedures. The only worksheet functions that you cannot use are those that have an equivalent VBA function.

VBA makes Excel's worksheet functions available through the WorksheetFunction object, which is contained in the Application object. Here's an example of how you can use Excel's SUM function in a VBA statement:

```
Total = Application.WorksheetFunction.Sum(Range("A1:A12"))
```

You can omit the Application part or the WorksheetFunction part of the expression. In either case, VBA will figure out what you're doing. In other words, these three expressions all work exactly the same:

```
Total = Application.WorksheetFunction.Sum(Range("A1:A12"))

Total = WorksheetFunction.Sum(Range("A1:A12"))

Total = Application.Sum(Range("A1:A12"))
```

Here's an example that shows how to use Excel's MAX worksheet function in a VBA procedure. This procedure displays the maximum value in column A on the active worksheet:

```
Sub ShowMax()
    Dim TheMax As Double
    TheMax = WorksheetFunction.MAX(Range("A:A"))
    MsgBox TheMax
End Sub
```

And, as you might expect, you can use other worksheet functions in a similar manner. For example, you can use the LARGE function to determine the *k*th-largest value in a range. The following expression demonstrates this:

```
SecondHighest = WorksheetFunction.LARGE(Range("A:A"),2)
```

Notice that the LARGE function uses two arguments; the second argument represents the *k*-th part — 2 in this case (the second-largest value).

6.3 User Defined Functions (UDF)

The third category of functions you can use in your VBA procedures is User Defined Function (also known custom functions). UDF is one you develop yourself by using VBA. To use a custom function, you must define it in the workbook in which you use it.

Here's an example of defining a simple Function procedure and then using it in a VBA Sub procedure:

```
Function MultiplyTwo(num1, num2) As Double
    MultiplyTwo = num1 * num2
End Function
```

```
Sub ShowResult()
    Dim n1 As Double, n2 As Double
    Dim Result As Double
    n1 = 123
    n2 = 544
    Result = MultiplyTwo(n1, n2)
    MsgBox Result
End Sub
```

The custom function MultiplyTwo has two arguments. The ShowResult Sub procedure uses this Function procedure by passing two arguments to it (in parentheses). The ShowResult procedure then displays a message box showing the value returned by the MultiplyTwo function.

You can also use custom functions in your worksheet formulas. For example, if MultiplyTwo is defined in your workbook, you can write a formula such as this one:

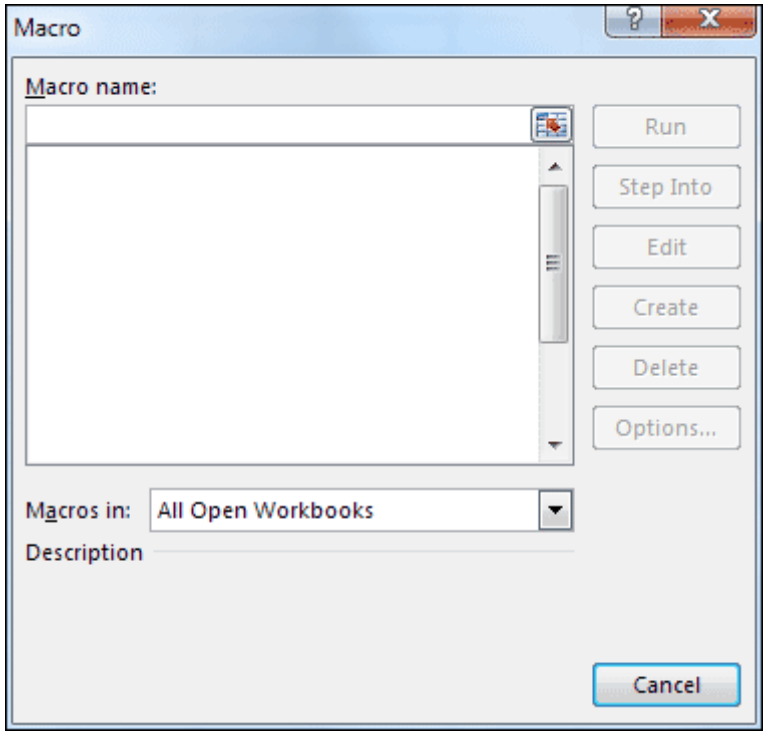
```
=MultiplyTwo(A1,A2)
```

This formula returns the product of the values in cells A1 and A2.

There are five items on the Code panel: Visual Basic, Macros, Record Macro, Use Relative References, and Macro Security. You've already seen the Macro Security item. If you can't run your Macros then click this item and change the security settings.

Clicking the **Visual Basic** item opens up the Visual Basic editor. You can also press ALT + F11 on your keyboard as a shortcut.

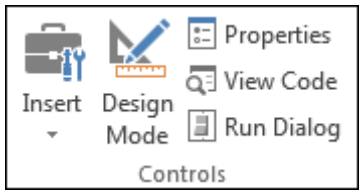
Clicking the Macros item on the Code panel brings up the following dialogue box:



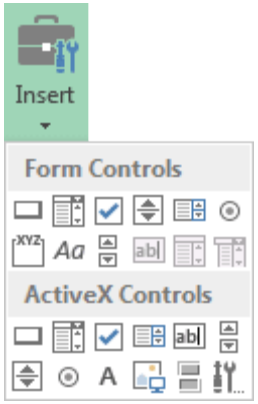
Once you create a few Macros, they will appear in the list. You can then run them by selecting a Macro Name and clicking the Run button.

The other two items are Record Macro and Use Relative References. We'll skip over these two, as we won't have too much use for them.

The panel we'll use a lot, though, is the Controls panel. It looks like this:



The first item, **Insert**, is expanded when you click on it. Doing so will reveal the following:



These are all the controls you can add to an Excel spreadsheet or a user form. We'll be adding one of these controls to a spreadsheet shortly. But back to the **Controls** panel. The other items are:

The **Design Mode** item is used to edit a control on a spreadsheet.

The **Properties** item shows you properties for a selected control.

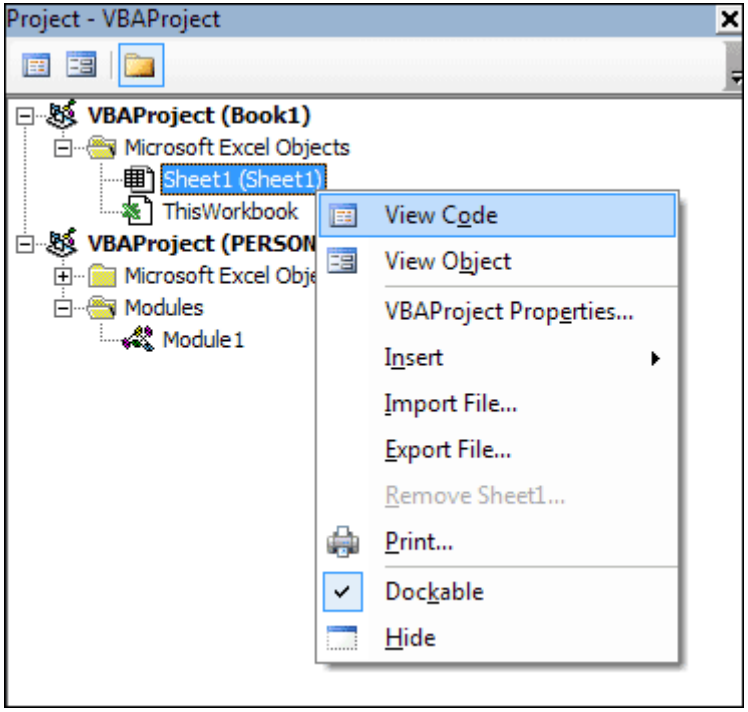
The **View Code** item takes you to the Visual Basic editor again.

The **Run Dialog** item can be ignored, as we won't be using it.

We'll now create a Macro that selects a range of cells. These cells will be selected when we click a button on a spreadsheet.

To create the Macro, click the **Visual Basic** item on the **Code** panel. The Visual Basic Editor will open up. We want this Macro to be only for **Sheet1** in the current Workbook. On the left hand side of the Editor, locate the Project Explorer panel. (If you can't see it, Click **View > Project Explorer** from the menu at the top of the Editor.)

In the Project Explorer right click the Sheet1 item under **VBAProject (Book1)**:

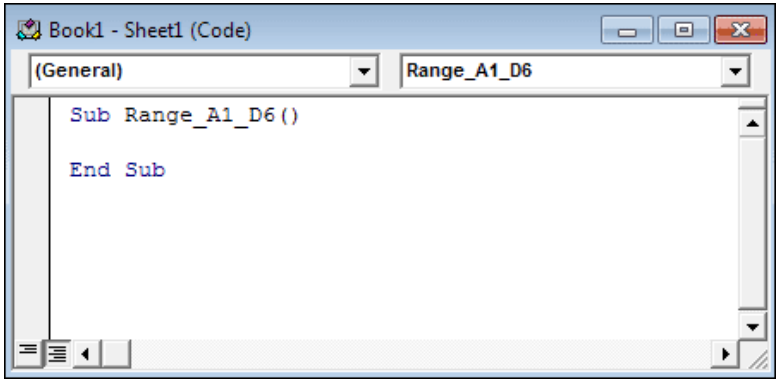


From the menu that appears, select **View Code**. A blank coding window will open up. (You can also double click on Sheet1 to open up the code window.)

What we want to do here is to create a Subroutine. This Subroutine will be our Macro. Type the following line into the white coding area:

Sub Range_A1_D6()

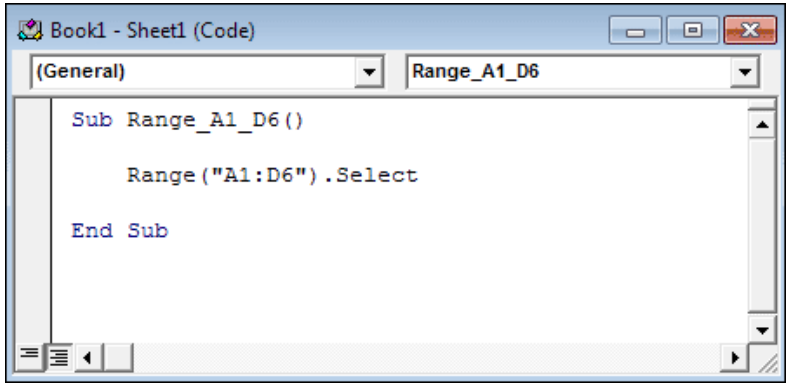
Press the Enter key on your keyboard and the Editor should add the corresponding End Sub for you. Your coding window will then look like this:



So we now have a Subroutine (a Macro) with the name **Range_A1_D6**. In between **Sub** and **End Sub** enter the following code:

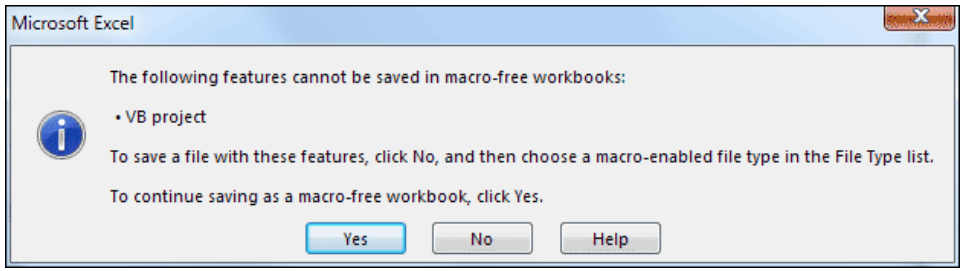
Range("A1:D6").Select

Your coding window should then look like this:



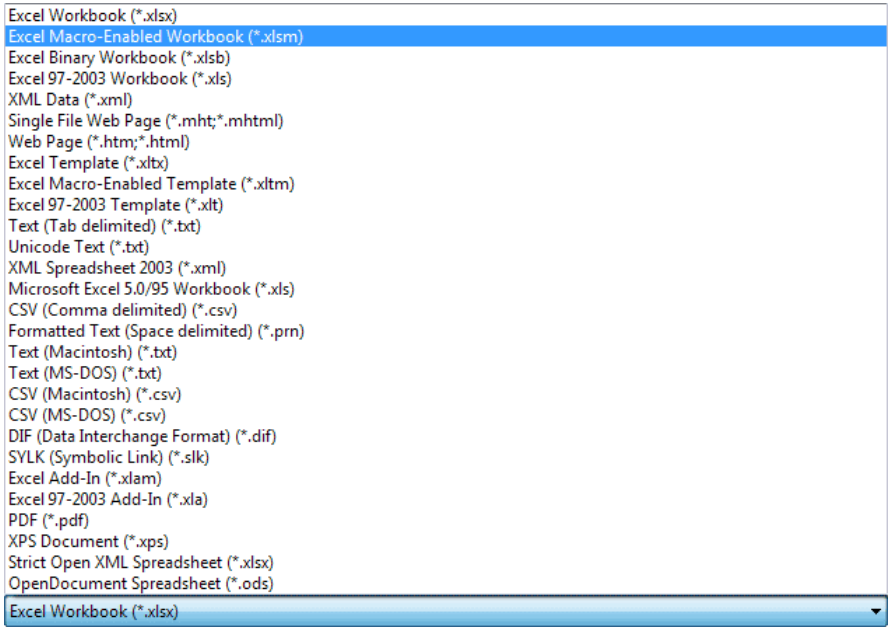
As you can guess, this code selects a range of cells. It selects the cells A1 to D6. Before we try it out, you need to save your work.

From the menu bar at the top of the Editor, click **File > Save Book1**. The Save As dialogue box should appear. Change the name of the file to **Range_Exercises.xlxs**. Click the **Save** button and you should see an error message appear. This one:



Click **No** to return to the Save As dialogue box.

You get the error message because your file contains Macros. Excel can't save Macros in a file that end in **xlxs**. You need to change the file ending. To do that, click the **Save As Type** dropdown list. From the list, select **Excel Macro-Enabled Workbook(*.xlsm)**:



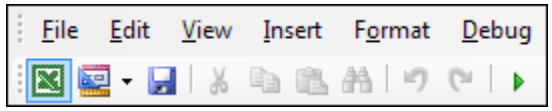
Click the Save button again. When you click the Save button this time, the file should save OK. But note that the file ending is now xlsm.

Now that you know how to save a spreadsheet with a Macro in it, it's time to do something with that Macro. We'll add a button to a spreadsheet and activate our Sub when the button is clicked. We'll do that in the next part of this lesson below.

8.7 Adding a Button to an Excel Spreadsheet

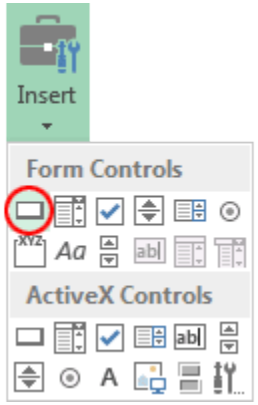
In the previous lesson, you created a simple Sub in the code window. In this lesson, you'll activate that Sub from a button on a spreasheet.

At the top of the VBA Editor, locate the Excel icon, just under the File menu:



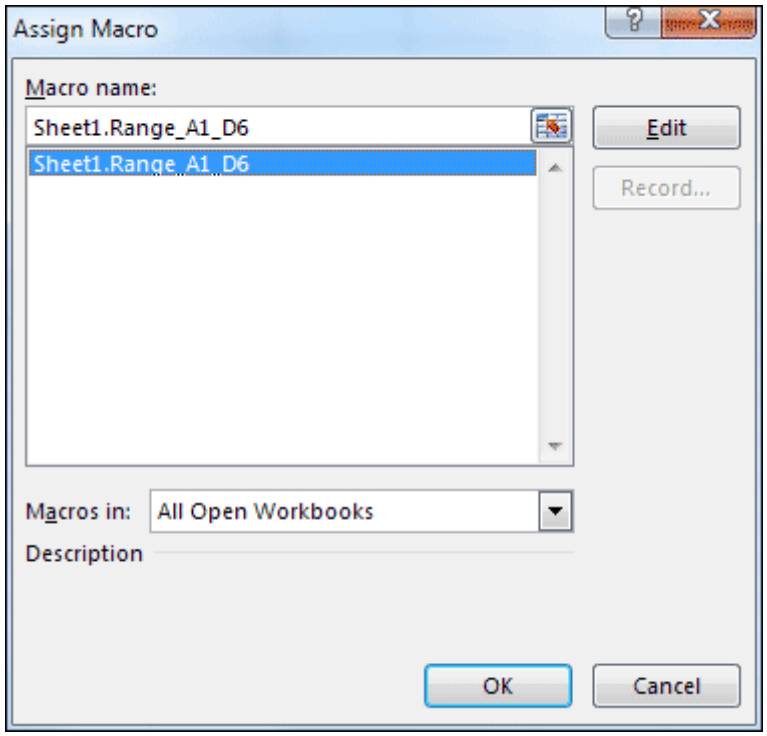
Click this icon to return to your spreadsheet. We'll now place a button control on the spreadsheet.

Locate the **Controls** panel on the **Developer** toolbar, and then click the **Insert** item. From the Insert menu, click the first item, which is a button:

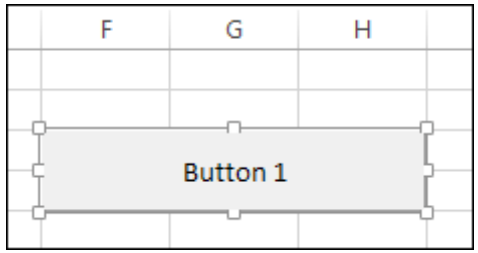


Now move your mouse to your spreadsheet. Hold down your left mouse button somewhere on the F column (F3 will do). Keep it held down and draw out a rectangular button. Let go of the left mouse button when your cursor is on H4

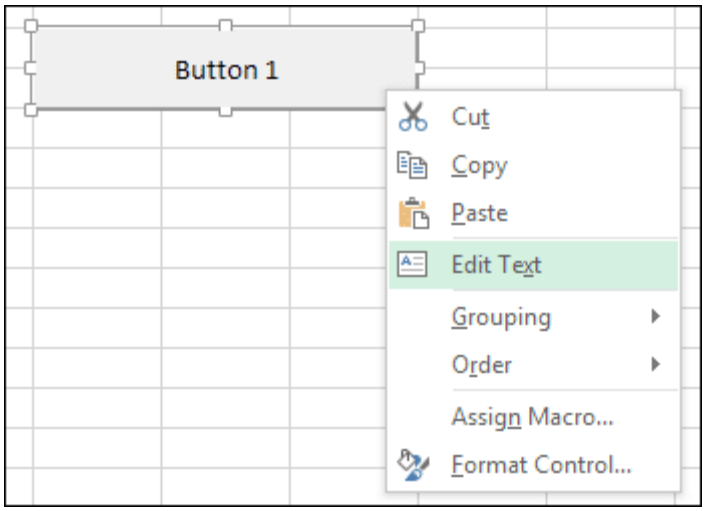
As soon as you let go of the left mouse button you'll see the **Assign Macro** dialogue box appear:



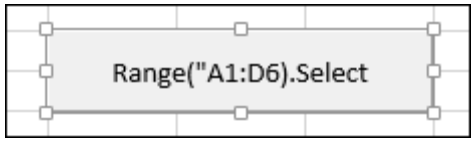
Select your Macro from the list and click OK. The button on your spreadsheet should now look like this:



You can edit the text on a button quite easily. Right click the button to see a menu appear. From the menu, select **Edit Text**:

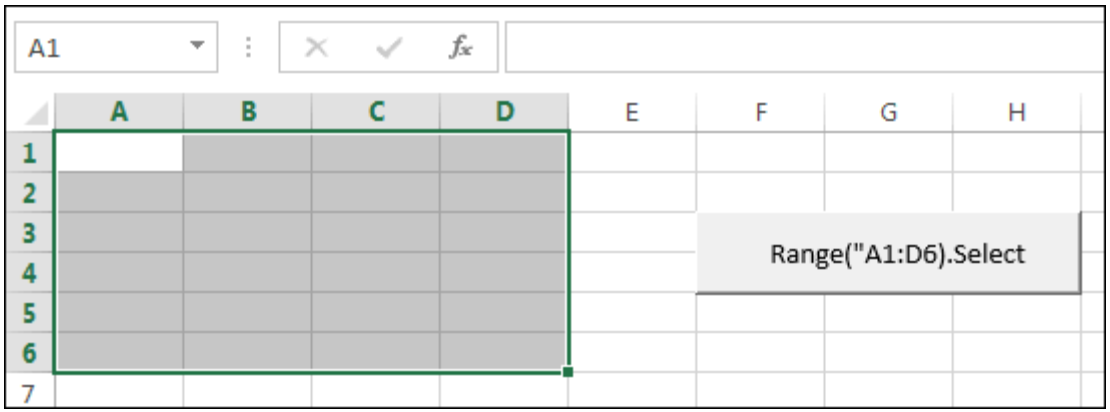


When you select **Edit Text**, a cursor will appear at the start of the text. Use the arrow keys on your keyboard to move the cursor to the end of the line. Delete the text **Button 1** and type**Range("A1:D6").Select** instead (If you accidentally click away from the button, click on it again with right mouse button and not the left button. This will select your button again.):



Click away from the button to exit edit mode and you'll see the sizing handles disappear.

You can now test your button out. Give it a click and you'll see the cells A1 to D6 highlighted:



Congratulations! You have now written Excel VBA code to select a range of cells on a spreadsheet. And all with the click of a button!

Now return to the Visual Basic editor (From the Developer toolbar, click Visual Basic on the Code panel.) Type a single quote before your Range line. The line should turn green:

```
Sub Range_A1_D6()

    'Range("A1:D6").Select

End Sub
```

The reason it turns green is because a single quote is used for comments. When the line is commented out it means Visual Basic will no longer see it as code, so doesn't do anything with it. You can add comments to remind yourself what your code does, as in the image below:

```
Sub Range_A1_D6()

    '================================
    '   HIGHLIGHT A RANGE OF CELLS
    '================================
    Range("A1:D6").Select

End Sub
```

Adding comments to your code is a good habit to get in to. Especially when you come back to your code after a few weeks or so. If you haven't added comments you may not quite understand what it was you were trying to do.

Back to the Range code, though. Notice how we referred to the range of cells A1 to D6:

```
Range("A1:D6")
```

Another way to refer to the same range is like this:

```
Range("A1", "D6").Select
```

This time, the start cell A1 and the end cell D6 are enclosed with double quotes. In between the two we have a comma.

Both the examples above do the same thing: they first select the top left cell of the range, and then the bottom right cell of the range. It's entirely up to you which you use. But with the second version you can use something called the ActiveCell.

ActiveCell

Instead of typing the name of a cell you can also refer to which cell on your spreadsheet is currently highlighted. The currently highlighted cell is called the **ActiveCell** (no spaces and with capital letters for the "A" and "C"). You can use this in your code. Let's see how it works.

After the End Sub of your first Subroutine, add the following:

```
Sub ActiveCell_Example()
```

Press the enter key on your keyboard to let the VB editor add the End Sub for you. Now add the following line between the Sub and End Sub of your code:

```
Range(ActiveCell, "D6").Select
```

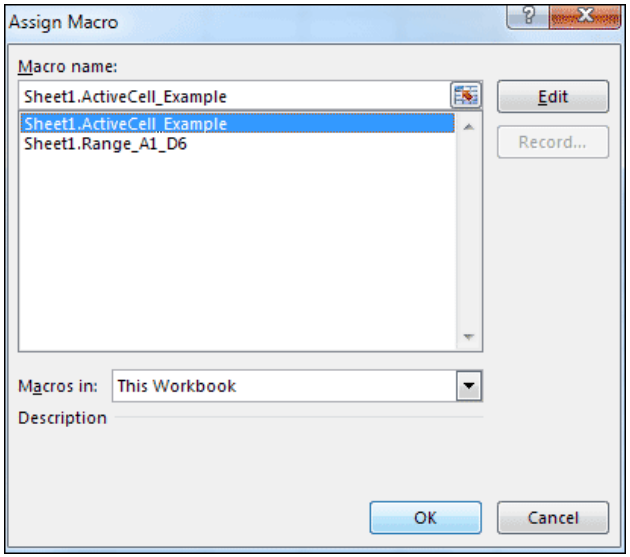
Your coding window will then look like this:

```
Sub Range_A1_D6()

    '================================
    '   HIGHLIGHT A RANGE OF CELLS
    '================================
    Range("A1", "D6").Select

End Sub
```

```
Sub ActiveCell_Example()

    Range(ActiveCell, "D6").Select

End Sub
```

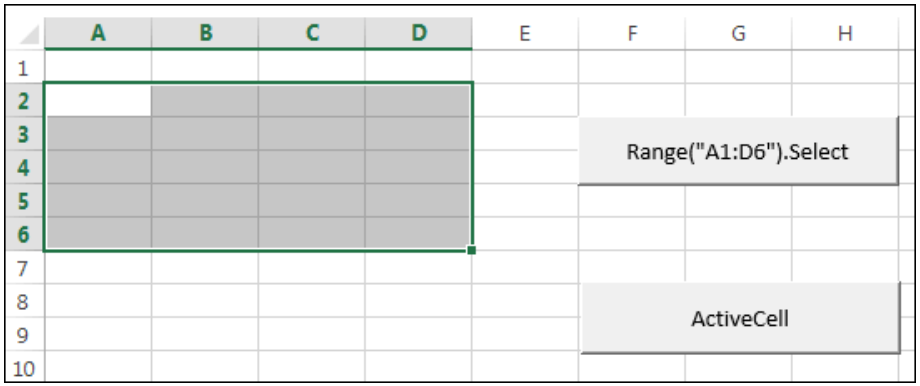
So the top left cell we want to select is the ActiveCell, which is whatever cell you clicked in on your spreadsheet. The bottom right cell we want to select is D6.

Click the icon to return to your Excel spreadsheet. Now draw another button on your form, just below the first one. You should see the Assign Macro dialogue box appear again:



Select your new Macro (your Sub) from the list and click OK.

When you get back to your spreadsheet, edit the text of the button again. Type **ActiveCell** as the text. When you have finished editing the text, click away. Click inside another cell on your spreadsheet, cell A2 for example. Now click your button. You should see the cells A2 to D6 highlighted:



Click inside any other cell on your spreadsheet and click the button again. The cells from your active cell to D6 will be selected.

In the next part of this tutorial, we'll take a look at the Offset property. So save you work before moving on.

8.8 The Excel VBA Offset Property

Another useful way to refer to a cell or group of cells is with the **Offset** property. This is used with the Range property so that you can specify a new location. But the new location is based on cells that you specify. As an example, examine this code:

```
Range("A1").Offset(RowOffSet:=1, ColumnOffset:=1).Select
```

The above code will select cell B2. The location we've used is cell A1. We've then typed a dot followed by the word **Offset**. In between round brackets, you tell Excel the new location. This is done with the parameters **RowOffSet** and **ColumnOffSet**. (The two are separated by a comma.) In other words, move one row from cell A1 and one column from cell A1.

You can use a shorthand instead of RowOffSet and ColumnOffSet. The shorthand just uses the numbers:

Range("A1").Offset(1, 1).Select

You can also just specify the Rows and not the columns:

Range("A1").Offset(1).Select

Here, we've missed off the column parameter and its comma. Excel takes this to mean you want to move one row down from the cell you specified with Range. The cell selected will now be A2. (We've gone down a row but stayed in the same column.)

Similarly, you can specify the columns but not the rows:

Range("A1").Offset(, 1).Select

Now, the row position is blank and the column position is filled in. Notice that we have left in the comma, though. Excel takes this to mean that you don't want a new position for the rows, just the columns. The cell selected will now be B1. (We've gone across one column but stayed in the same row.)

You can also specify negative numbers:

Range("B2").Offset(-1, -1).Select

This time, we're starting in cell B2. In between the round brackets of **Offset** we have -1, -1. A negative number means go up one row or column from the starting position. Going up one row and one column from cell B2 takes you to cell A1.

You can also specify more than one cell for the Range:

Range("A1:C3").Offset(1, 1).Select

Here, we've started with the range of cells A1 to C3. We want to offset this entire range by one row and one column. The new range of cells in the line of code above will be B2 to D4. When specifying a range of cells with offset, not only does the first cell (A1 above) get moved across and down 1 but the last cell (C3) gets moved across 1 and down 1.

If all this is confusing, let's clear it up with a few practical examples.

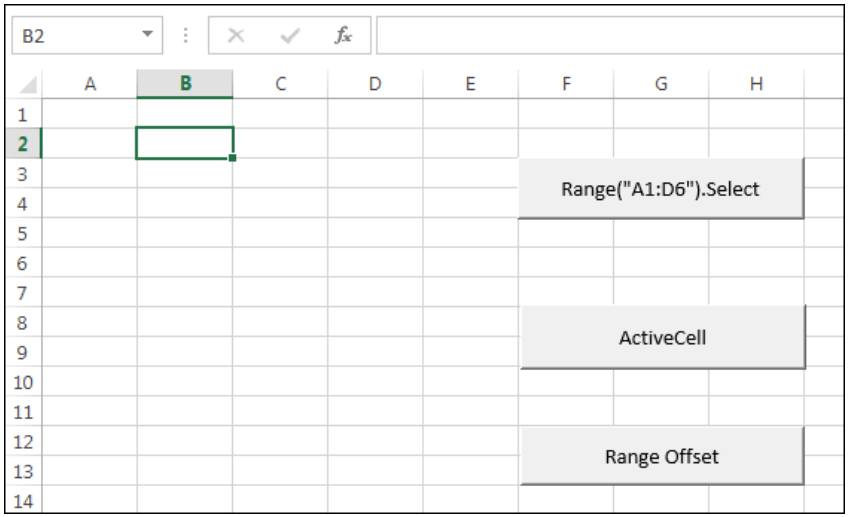
Return to your coding window from the previous lesson. Create another Sub and give it the name**Range_Offset**. Add the following code:

Range("A1").Offset(RowOffSet:=1, ColumnOffset:=1).Select

Your coding window will then look like this (we've added some comments):

```
Sub Range_Offset()

    '=================================
    '   OFFSET BY 1 ROW AND 1 COLUMN
    '=================================
    Range("A1").Offset(RowOffSet:=1, ColumnOffset:=1).Select

End Sub
```

Return to Excel and draw out another button on your spreadsheet. When the Assign Macro dialogue box appears, select your new Sub, **Range_Offset**. Change the text on the button, as well. To test it out, select the cell A1 on your spreadsheet. Then click your button. You should find that Excel now selects cell B2:



Return to your coding window. Now change the code for your new Sub to this:

```
Range("A1").Offset(2, 2).Select
```

Test out the new code by returning to Excel and clicking your button. Which cell does Excel select now?

Try out the other Offset positions we've mentioned in this lesson. Amend your code and try each of these in turn:

```
Range("A1").Offset(3).Select
Range("A1").Offset(, 2 ).Select
Range("B2").Offset(-1, -1 ).Select
Range("A1:C3").Offset(1, 1).Select
```

Now try these two exercises.

Exercise
Suppose you've used Range to reference the cell A2. How would you get to cell C5 using Offset?

Exercise
If you've used Range to reference the cell E8, how would to get to cell B3 using Range and Offset?

OK, we'll move on. Before doing so, make sure you understand how Offset works. We'll take a look at the Resize property in the next lesson below.

8.9 The Excel VBA Resize Property

When you use Offset, as we did in the previous lesson, you're selecting a new range of cells based on a starting position. But the starting position will change. If you want to change the size of a range of cells, but keep the starting position, then Resize is the better option. Take the following line of code as an example:

```
Range("A1").Resize(RowSize:=2, ColumnSize:=2).Select
```

This is almost identical to the Offset code you used earlier. The only difference is the word Resize instead of Offset. Again we start with a Range (cell A1). In between the round brackets of Resize you type a RowSize and ColumnSize. Like before, though, you can shorten this and just have the numbers between the round brackets:

Range("A1").Resize(2, 2).Select

If you only want to resize the rows then you can miss out the ColumnSize parameter:

Range("A1").Resize(2).Select

If you only want to resize the columns then you can miss out the RowSize parameter:

Range("A1").Resize(, 2).Select

Let's see how Resize works in practice.

Return to your coding window. Set up a new Sub and call it **Range_Resize**. In between **Sub** and **End Sub**, type the following line:

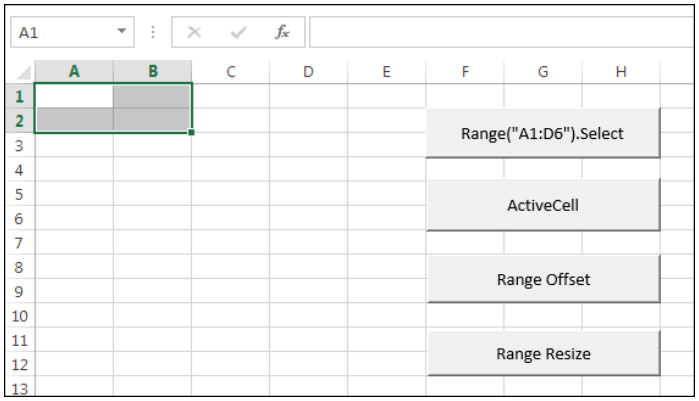
Range("A1").Resize(RowSize:=2, ColumnSize:=2).Select

Your code will then look like this:

```
Sub Range_Resize()

    '====================================
    '   RESIZE BY 2 ROWS AND 2 COLUMNS
    '====================================
    Range("A1").Resize(RowSize:=2, ColumnSize:=2).Select

End Sub
```

Go back to your spreadsheet and add a new button. When the Assign Macro dialogue box appears, select your **Range_Resize** Sub from the list. Click OK to return to your spreadsheet. Change the text of the button to **Range Resize**.

To test out your new button, click inside cell A1 just to highlight your starting position. Now click the button. You should find that the cells A2 to B2 are highlighted:



To get a better understanding of how Resize works, try the following (amend your code one example at a time):

```
Range("A1").Resize(3, 4).Select
Range("A1").Resize(3).Select
Range("A1").Resize(, 4).Select
Range("A1:A3").Resize(, 2).Select
```

Notice that the last example uses the cells A1 to A3 for the Range property. The others use A1 as the starting position.
You can also resize AND offset. It's a little trickier to understand, though:

```
Range("B1").Offset(, -1).Resize(, 3).Select
```

In the code above, we start with the Range B1. We then offset this by one column to the left (the -1). The Resize then stretches the range of cells 3 columns. Try it and see which cells are selected in the line above.

Exercise
Suppose the selected cell is A2, use Resize to select cells A2 to C2.

OK, we'll move on from Ranges. In the next section, we'll have a look at something called programming variables, and how they work with the Excel VBA language.

8.10 Excel VBA and Variables

In computer programming, you need to store things in memory, things like numbers and strings of text. You store them so that you can retrieve them later and manipulate the numbers or strings of text in some way. To store things in memory you use a **variable**. Variables have a name (something that you yourself come up with) and a type (such as Integer or String). In the Visual Basic programming language you also need to declare that you are setting up a variable. This is done (mostly) with the word **Dim**. As an example, study the following:

```
Dim MyNumber As Integer
```

The above code sets up a variable with the name **MyNumber**. The type of variable is a whole number (**As Integer**). The **Dim** keyword goes at the start, and tells the programme to set up a variable of this name and type.

However, there's nothing stored inside of the **MyNumber** variable - VBA has just allocated the memory at this point. To store something inside of a variable, you use the equal sign (=), also known as the assignment operator. To store a value of 10, say, inside of the variable called **MyNumber**, you do it like this:

```
MyNumber = 10
```

The above line reads, "Assign a value of 10 to the variable called MyNumber". So the name of your variable comes first, then the equal sign. After the equal sign you type the value that you want to store inside of your variable.

Sadly, you can't set up a variable and assign a value all on the same line. (You can in Visual Basic .NET but not in Visual Basic for Applications.) So you can't do this:

```
Dim MyNumber As Integer = 10
```

Variable Names

You can call your variable just about anything you like. But there are a few things you're not allowed to do. They are:

- You can't start a variable name with a number
- You can't have spaces in your variable names, or full stops (periods)
- You can't use any of the following characters: !, %, ï¿½, #, $

So these variable names are OK

MyVariable
My_Variable
myvariable2

But these variable names will get you an error:

2MyVariable
My Variable
$myvariable

The first one above starts with a number, the second one has a space, and the third starts with a dollar sign.

With variable names, it's best to come up with something relevant. So if you're storing a discount rate, then call it DiscountRate and not something odd like var1.

In the next lesson, you'll get some practice using variables in Excel VBA.

8.11 Excel VBA Variable Practice

In the previous section, you learnt about variables. To get the hang of variables, create a new blank spreadsheet. Click on the **Developer** ribbon at the top of Excel. On the **Controls** panel, click on **View Code**. This will open up the Visual Basic Editor with the Sheet1 coding window already open for you. Set up a new Sub and call it **Variable_Practice**. Add the following two lines for the Sub:

```
Dim MyNumber As Integer

MyNumber = 10
```

Your coding window will then look like this:

```
Sub Variable_Practice()

    Dim MyNumber As Integer

    MyNumber = 10

End Sub
```

So this code just sets up a variable called **MyNumber**. We then store a value of 10 into this variable. This is important in programming: whatever you put on the right hand side of an equal sign is what you are trying to store; whatever you have on the left of the equal sign is the place where you're trying to store it.

However, you'd want to actually do something with this value. After all, what's the point of storing a value if you're not going to use it? What we can do is to transfer our stored value to a cell on a spreadsheet.

You have been using Range in previous lessons. One property of Range is Value. The Value property can get or set the value of a cell, or group of cells. To set a Value, the Range goes before an equal sign:

Worksheets(1).Range("A1").Value =

Here, we're first accessing the **Worksheet 1** object. We want to point to the **Range A1**. After a dot, we then type the **Value** property. This is followed by the equal sign. After the equal sign, you can type your value:

Worksheets(1).Range("A1").Value = 10

Now, the **Value** for **Range A1** on **Worksheet 1** is set to **10**.

Instead of typing a number, you can type the name of a variable. We have already stored a value of 10 inside of the **MyNumber** variable. So we can just use this:

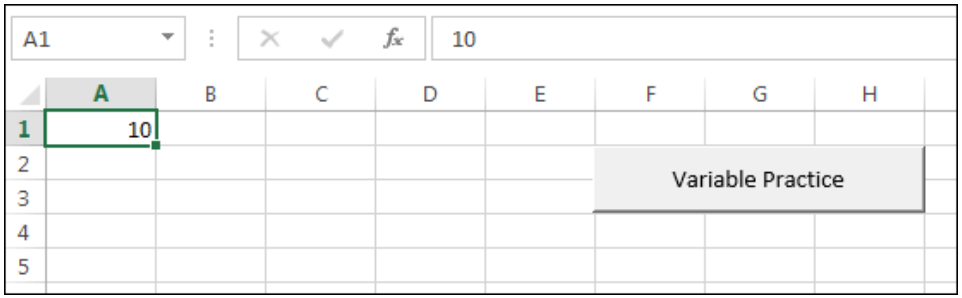
Worksheets(1).Range("A1").Value = MyNumber

VBA sees the variable called **MyNumber** and then fetches whatever value is stored inside of it. That value is then stored inside of whatever is on the left of the equal sign.

Add that line to your code, just below the other two. Your coding window will then look like this:

```
Sub Variable_Practice()

    Dim MyNumber As Integer

    MyNumber = 10

    Worksheets(1).Range("A1").Value = MyNumber

End Sub
```

Now try it out. Go back to your spreadsheet and add a button. When the Assign Macro dialogue box appears, select your **Variable Practice** Sub from the list. Change the text on the button to **Variable Practice**. Deselect your button by clicking away. Now click it again to test it. You should find that the number 10 appears in cell A1 on your spreadsheet:



Now return to your code. Locate the **Dim** line from your **Variable Practice** Sub. This one:

Dim MyNumber As Integer

Comment the line out by typing a single quote mark before it. You code will then look like this:

```
Sub Variable_Practice()

    'Dim MyNumber As Integer

    MyNumber = 10

    Worksheets(1).Range("A1").Value = MyNumber

End Sub
```

A comment, remember, means that the line will be ignored. The code that VBA will execute is now only this:

MyNumber = 10

Worksheets(1).Range("A1").Value = MyNumber

The question is, will the code still run, or will it throw up an error?

Try it out by returning to Excel and clicking your button. You should find that it runs OK, with no problems at all. But why? We didn't set a variable name with the Dim keyword, as this is now commented out. So what's going on?

Well, you can actually set up a variable like this:

Variable_Name = Value_Here

In other words, you don't actually need to say **Dim Variable_Name As Ineteger**. You can miss it out entirely. However, if you do VBA sets the variable type up as something called a Variant. **Variant** data types can hold just about any kind of value. VBA will decide, when your programme runs, what type of data to store in your variable.

So if you don't need to set up variables with the **Dim** keyword, why bother using them at all? Well, it turns out that using the Variant data type will cause your code to run really slowly compared to setting up variables with the **Dim** keyword and using a named type like **As Integer**.

We'll be setting up our variables with the Dim keyword wherever possible.

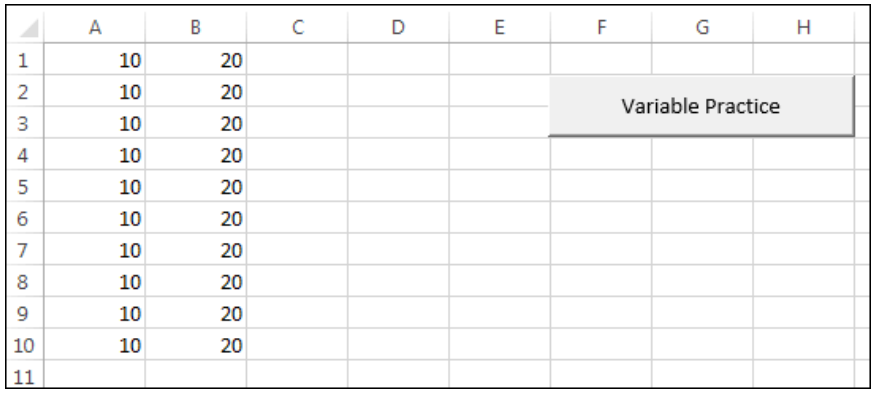
Exercise
Uncomment your Dim line by deleting the single quote. Now change the Range from "A1" to "A1:A10". Return to Excel and run your code again. What happens?

You can set up more than one variable, of course. Amend your code to this (the new lines are in bold):

```
Dim MyNumber As Integer
Dim MyOtherNumber As Integer

MyNumber = 10
MyOtherNumber = 20

Worksheets(1).Range("A1:A10").Value = MyNumber
Worksheets(1).Range("B1:B10").Value = MyOtherNumber
```

We've set up another Integer variable (**As Integer**) and called it **MyOtherNumber**. A value of 20 is being stored in this new variable. The Value of the Range B1 to B10 is being set to whatever value is stored in **MyOtherNumber**.

When you click the button on your spreadsheet you should now see this:



In the next lesson, you'll see learn about something called Option Explicit.

8.12 Using variables to add and subtracting Excel VBA

When you store numbers inside of variables, one of the things you can do with them is mathematical calculations. We'll start with simple addition.

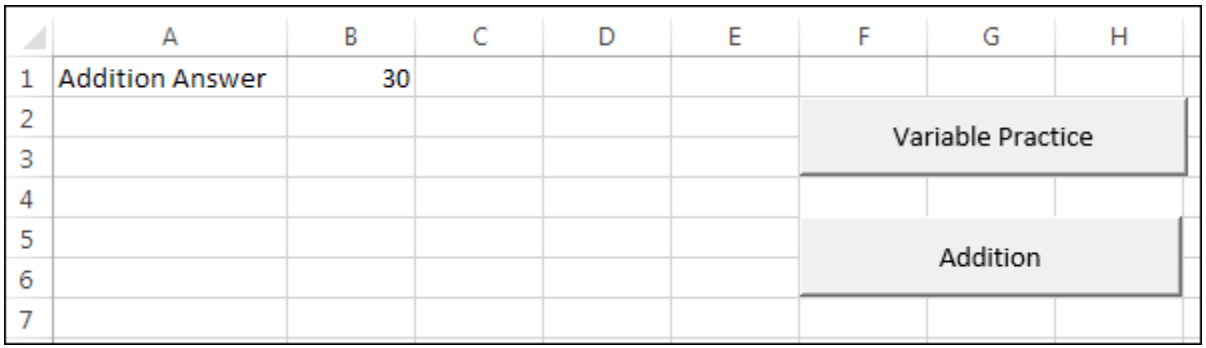
Return to your code from the previous section. Set up another Sub and call it **Add_Numbers**. Inside of **Sub** and **End Sub** type the following code:

```
Dim Number_1 As Integer
Dim Number_2 As Integer

Number_1 = 10
Number_2 = 20

Worksheets(1).Range("A1").Value = "Addition Answer"
Worksheets(1).Range("B1").Value = Number_1 + Number_2
```

Your coding window will then look like this:

```
Sub Add_Numbers()

    Dim Number_1 As Integer
    Dim Number_2 As Integer

    Number_1 = 10
    Number_2 = 20

    Worksheets(1).Range("A1").Value = "Addition Answer"
    Worksheets(1).Range("B1").Value = Number_1 + Number_2

End Sub
```

Return to Excel. Select any cells that have numbers in them, then press the Delete key on your keyboard to get rid of them. Now add a new button. From the Assign Macro dialogue box select your**Add_Numbers** Sub. Change the button text to **Addition**. Click away from the button to deselect it. Now click the button again. The result should be this (you'll need to widen the **A** column):



We've done two things here: we've used VBA to add some text inside of cell A1, and we've put the result of our addition into cell B1. The line that puts the text into cell A1 is this:

Worksheets(1).Range("A1").Value = "Addition Answer"

This time after the equal sign we have some direct text. If you want direct text to the right of the equal sign you need to enclose it between double quotation marks. We have the text **Addition Answer**between double quotes. The **Value** of **Range A1** on **Worksheets(1)** is then set to this new text.

The part that does the adding up is this line:

Worksheets(1).Range("B1").Value = Number_1 + Number_2

To the right of the equal sign we now have this:

Number_1 + Number_2

We have already stored a value of 10 inside of the variable called **Number_1**. Inside of the variable called **Number_2** we have stored a value of 20. Just like in ordinary mathematics, the plus symbol is used to add up. VBA will add the two values for you and store the answer into the cell B1.

You can add up more than two numbers, of course. If we had a third variable, we could have done this:

Worksheets(1).Range("B1").Value = Number_1 + Number_2 + Number_3

You can add up numbers that aren't inside of variables. So this is perfectly fine:

```
Worksheets(1).Range("B1").Value = Number_1 + Number_2 + 30
```

Or this:

```
Worksheets(1).Range("B1").Value = 10 + 20 + 30
```

Or you can store your addition in just one variable:

```
Number_1 = 10 + 20 + 30
Worksheets(1).Range("B1").Value = Number_1
```

Subtraction

In the VBA programming language, the minus sign (-) is used to subtract one value from another. Again, you can use actual values, values stored in variables, or a combination of the two.

Go back to your code. Set up another Sub and call it **Subtract_Numbers**. Add the following lines of code:

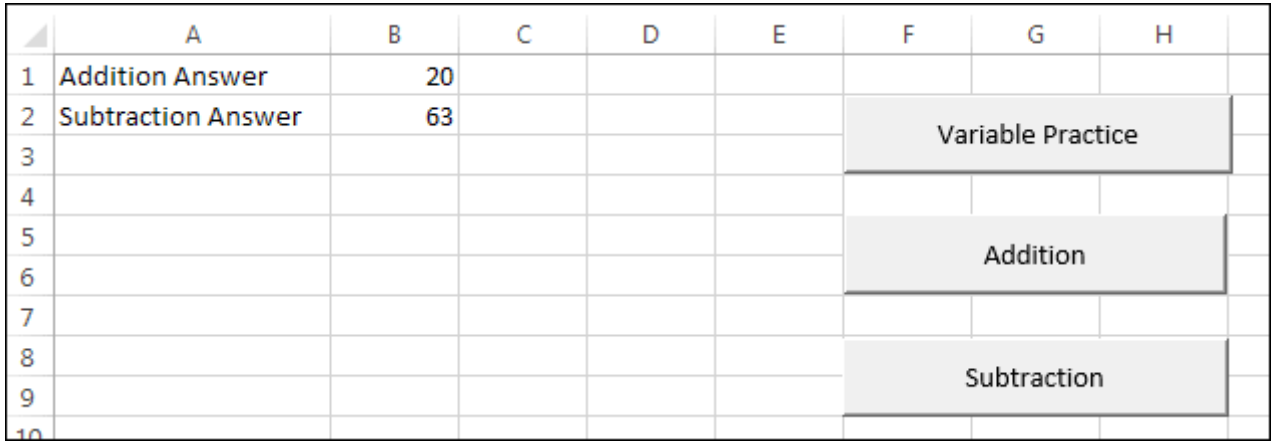
```
Dim Number_1 As Integer
Dim Number_2 As Integer

Number_1 = 450
Number_2 = 387

Worksheets(1).Range("A2").Value = "Subtraction Answer"
Worksheets(1).Range("B2").Value = Number_1 - Number_2
```

Your coding window will then look like this:

```
Sub Subtract_Numbers()

    Dim Number_1 As Integer
    Dim Number_2 As Integer

    Number_1 = 450
    Number_2 = 387

    Worksheets(1).Range("A2").Value = "Subtraction Answer"
    Worksheets(1).Range("B2").Value = Number_1 - Number_2

End Sub
```

Notice that we're using the same variable names, here: **Number_1** and **Number_2**. This is perfectly acceptable as both are enclosed within their own **Sub** and **End Sub**. The thing you can't do is to set up two variables with the same name between the same and **Sub** and **End Sub** lines. But if they are in two different Subs, that's OK. (This is known as Variable Scope.)

Return to your spreadsheet and add a new button. From the Assign Macro dialogue box select your**Subtract_Numbers** Sub. Change the button text to **Subtraction**. Test it out and you should see a new line appear on your spreadsheet:



So we set up two variables and stored values of 450 and 387 in them. We added some direct text to cell A2, and used the following subtraction for cell B2:

```
Worksheets(1).Range("B2").Value = Number_1 - Number_2
```

The only difference between this and the addition code (apart from the B2 cell reference) is the use of the subtraction symbol (-) in place of the addition symbol (+). When the code is run, VBA looks at the values in the two variables. It then deducts one from the other. The answer is then stored as the Value for the Range on Worksheets(1).

Just like addition, you can use more than one variable, a mixture of variables, or no variables at all, as in the following lines:

```
Number_1 - Number_2 - Number_3
Number_1 - 10
300 - 200
```

You can also mix the addition and subtraction. Amend the code for your **Subtract_Numbers** to this (the new or amended lines are in bold):

```
Dim Number_1 As Integer
Dim Number_2 As Integer
Dim Number_3 As Integer
Dim Answer As Integer

Number_1 = 50
Number_2 = 40
Number_3 = 30

Answer = Number_1 + Number_2 - Number_3

Worksheets(1).Range("A2").Value = "Subtraction Answer"
Worksheets(1).Range("B2").Value = Answer
```

We've set two new variables here:

```
Dim Number_3 As Integer
Dim Answer As Integer
```

After setting up the variables, we have this:

Number_1 = 50
Number_2 = 40
Number_3 = 30

Answer = Number_1 + Number_2 - Number_3

The first three lines just store the numbers into the variables. The fourth line is where we perform a calculation. The first thing we do is add the value stored in Number_1 (50) to the value stored in Number_2 (40). Once this addition is performed (50 + 40), we deduct the value stored in the variable called Number_3 (30).

Return to your spreadsheet and click your button. You should see a value of 60 appear in cell B2. The reason it does so is because of this line:

Worksheets(1).Range("B2").Value = Answer

The answer to our calculation has been stored in the variable called Answer. We then use this variable as the Value for the Range B2 in Worksheets(1).

But what we're really doing is just this:

Dim Answer As Integer

Answer = 50 + 40 - 30

Worksheets(1).Range("B2").Value = Answer

In the next lesson, you'll see how to multiply and divide with Excel VBA.

8.13 Using variables to multiply and divide in Excel VBA

In the last lesson, you used variables to add and subtract with Excel VBA code. In this lesson, you'll learn how to multiply and divide.

Multiplication

In programing languages, the multiplication sign is the asterisk (*). So if you want to multiply 10 by 5 in VBA you could do it like this:

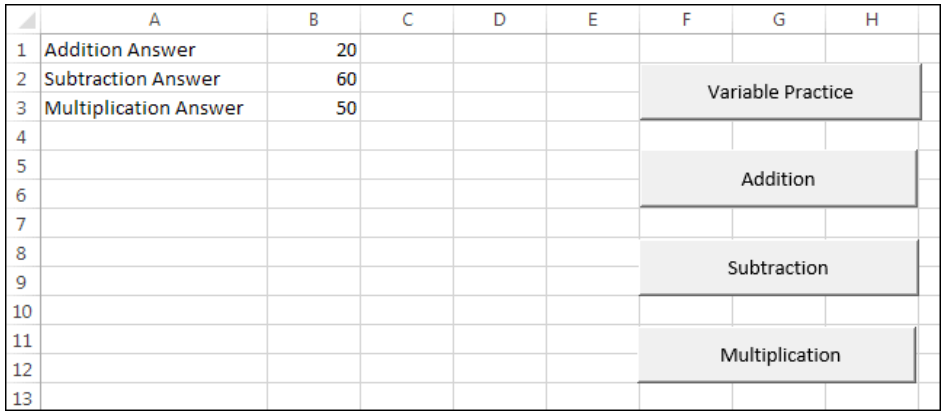
```
Dim Number_1 As Integer
Dim Number_2 As Integer
Dim Answer As Integer

Number_1 = 10
Number_2 = 5

Answer = Number_1 * Number_2

Worksheets(1).Range("A3").Value = "Multiplication Answer"
Worksheets(1).Range("B3").Value = Answer
```

Try it out for yourself. Return to your coding window. Add another Sub and call it **Multiply_Numbers**. In between **Sub** and **End Sub** type the code above.

The code is more or less the same as before. The only differences are the cell references (A3 and B3) and the multiplication sign (*). Your coding window should look like this:

```
Sub Multiply_Numbers()

    Dim Number_1 As Integer
    Dim Number_2 As Integer
    Dim Answer As Integer

    Number_1 = 10
    Number_2 = 5

    Answer = Number_1 * Number_2

    Worksheets(1).Range("A3").Value = "Multiplication Answer"
    Worksheets(1).Range("B3").Value = Answer

End Sub
```

Once you have added the code, return to your spreadsheet. Add a new button and select**Multiply_Numbers** from the Assign Macro dialogue box. Change the text on the button as before. When you click your button, you should see a new line added:



As with Addition and Subtraction, you can use more than two numbers or variables in your calculations. So these are fine:

Answer = Number_1 * 10
Answer = Number_1 * Number_2 * Number_3
Answer = Number_1 * Number_2 * 10

You can mix the Addition, Subtraction and Multiplication, but you need to take care. For example, what is the correct answer to the sum below?

Answer = 10 * 2 + 5

If you do the sum from left to right you'd first multiply the 10 and the 2 to get 20. Now add the 5 to get and answer of 25. However, if you work form right to left, you'd first add the 5 and the 2 to get 7. Multiply 7 by 10 and you'd get 70, a totally different answer!

VBA works things out from left to right. But you can force the answer you need by using round brackets:

Answer = 10 * (2 + 5)

The round brackets above surround the 2 + 5. VBA takes this to mean you want to add these two numbers first. Once it has an answer it will then do the rest of the calculation. It's a good idea to use round brackets to avoid any confusion.

Division

The symbol to use when you want to divide numbers is the forward slash (/). (Quite bizarrely, though, you can also use the back slash (\) without getting any error messages. You may get errors in your calculations, though)

Try out some division for yourself. Return to your coding window and add a new Sub. Call it**Divide_Numbers**. In between Sub and End Sub, type the following code:

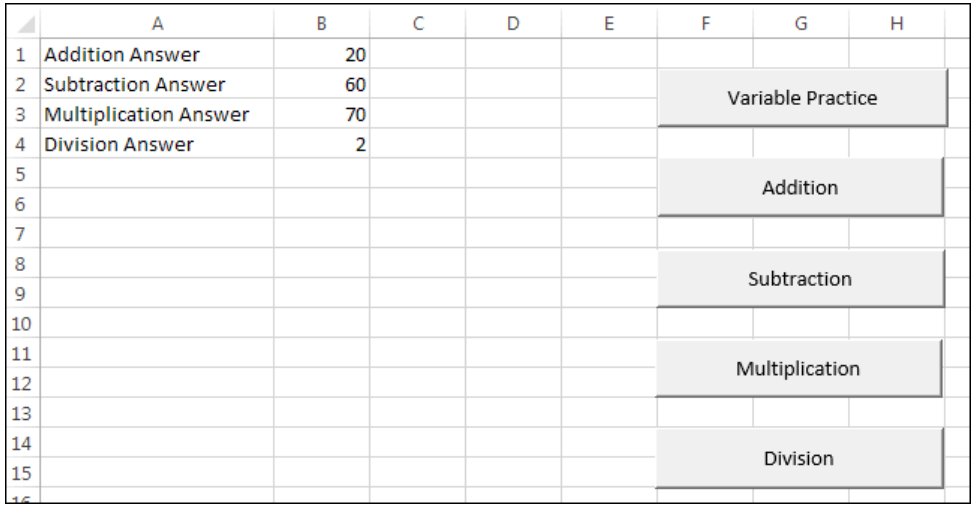
```
Dim Number_1 As Integer
Dim Number_2 As Integer
Dim Answer As Integer

Number_1 = 10
Number_2 = 5

Answer = Number_1 / Number_2
```

Worksheets(1).Range("A4").Value = "Division Answer"
Worksheets(1).Range("B4").Value = Answer

Your coding window will then look like this:

```
Sub Divide_Numbers()

    Dim Number_1 As Integer
    Dim Number_2 As Integer
    Dim Answer As Integer

    Number_1 = 10
    Number_2 = 5

    Answer = Number_1 / Number_2

    Worksheets(1).Range("A4").Value = "Division Answer"
    Worksheets(1).Range("B4").Value = Answer

End Sub
```

Return to Excel and add a new button to your spreadsheet. From the Assign Macro dialogue box select your **Divide_Numbers** Sub. Change the text on the button. When you click your new button, you should see a new line appear:



Now go back to your code. Change **Number_2** from 5 to 4:

```
Number_1 = 10
Number_2 = 4
```

So we're now dividing 10 by 4. Return to Excel and click your Division button. What answer do you get? Instead of the expected 2.5 you still get 2! The reason VBA has chopped off the .5 at the end is because we're using **As Integer** in our code. When you use the **As Integer** variable type you only get whole numbers, not fractions. To get a "point something" you need to use a different variable type. You'll learn more about the different variable types shortly. For now, change your code to this (the new lines are in bold):

```
Dim Number_1 As Integer
Dim Number_2 As Integer
Dim Number_3 As Integer
Dim Answer As Integer

Number_1 = 8
Number_2 = 8
Number_3 = 4

Answer = Number_1 + Number_2 / Number_3

Worksheets(1).Range("A4").Value = "Division Answer"
Worksheets(1).Range("B4").Value = Answer
```

What we're doing here is mixing some addition with division. Our sum is really this:

```
Answer = 8 + 8 / 4
```

You may think that this says "first add 8 and 8 then divide by 4". The answer you'd be expecting is 16 / 4, which is 4. However, try out the code by clicking the button on your spreadsheet and you'll find that the answer you actually get is not 4 but 10! So what's going on?

The reason you get 10 and not 4 is because of something called **operator precedence**. All this means is which of the mathematical operators (+, -, * /) has priority. VBA sees division as more important than addition, so it does the dividing first. Replace the / symbol with the * symbol and you'll find that multiplication is also more important than addition. So the answer to this sum:

Answer = 8 + 8 * 4

is 40, according to VBA, and not 64.

With **operator precedence** you have to take into account the following:

- Division and Multiplication are done before addition and subtraction
- Division and Multiplication have equal priority and so are calculated from left to right, as long as there's no addition and subtraction to do
- Addition and subtraction have equal priority and so are calculated from left to right, as long as there's no division and multiplication to do

If the above is somewhat confusing, just remember to use round brackets to make it clear to VBA what you want to do:

Answer = (8 + 8) / 4

In the above sum, VBA will now add 8 to 8, because of the round brackets. The answer to whatever is between the round brackets will then get divided by 4.

In the next part, you'll learn about some more variable types in Excel VBA.

8.14 Other Variable Types in Excel VBA

As well as declaring a variable to be of type **Integer**, as you have been doing so far, you can also have the following numerical variable types:

As Long
As Single
As Double
As Currency

The difference between all these numerical data types is how many digits they can hold, and whether or not you want a decimal point (there is actually an **As Decimal** variable type, but it's a bit fiddly to use).

The first one on the list, **As Long**, is a great alternative to **As Integer**. We've been using As Integer exclusively up until now. The problem with the Integer variable type, however, is that it can only hold numbers up to a value of 32, 767. At the lower level the Integer variable type can only hold negative numbers up to -32, 768.

If you want store bigger numbers then clearly 32, 767 may not be enough. This is where **As Long** comes in. The Long variable type can hold positive numbers up to a value of 2, 147, 483, 647. The lowest negative number is -2, 147, 483, 648.

But Integer and Long are both used to store whole numbers. They would be no good if you wanted divide 10 by 3, say. If you want a remainder, you'll need a different variable type.

The variable types you can use for greater precision are **As Single** and **As Double**. The difference between the two are how many digits they can hold. **As Single** holds 4 bytes of data while **As Double**can hold 8 bytes. If you want a really, really long floating point number (a number with "point something" at the end) then use **As Double**, otherwise just use **As Single**.

Let's test some of this theory out, though. You can create a new spreadsheet for this, if you want. Or use the one you currently have open. But create a new Sub in a coding window (you should know how to do this by now). Call it **RowCount**. Add the following code:

Dim NumberOfRows As Long

NumberOfRows = Worksheets(1).Rows.Count

MsgBox NumberOfRows

Your coding window will then look like this:

```
Sub RowCount()

    Dim NumberOfRows As Long

    NumberOfRows = Worksheets(1).Rows.Count

    MsgBox NumberOfRows

End Sub
```

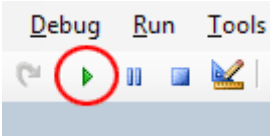
We've set up a variable called **NumberOfRows**. Instead of **As Integer**, we've used **As Long**. The second line uses **Rows.Count** to get the number of rows on **Worksheet 1**. When VBA gets this number, it stores it in our **NumberOfRows** variable. The third line is this:

MsgBox NumberOfRows

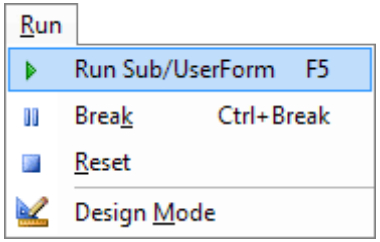
The **MsgBox** stands for Message Box. We just want to quickly test some values here, so there's no need to add a button to the spreadsheet. You'll learn more about Message Boxes a little later. But the Message box will display whatever value is in the variable **NumberOfRows**.

Rather than a long explanation about MsgBox, try it out to see what it does.

Make sure your cursor is flashing between the **Sub** and **End Sub** of your new code. Now locate the green arrow on the toolbars at the top of the coding window:

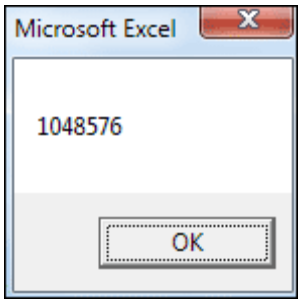


This is the Run icon. When you click it, VBA will try to run your Subs inside of the coding environment. You can also click **Run Sub/User Form** from the Run menu at the top:



A shortcut to running your Subs is to press the F5 key on your keyboard.

When you run your Sub, though, you should see a message box appear:

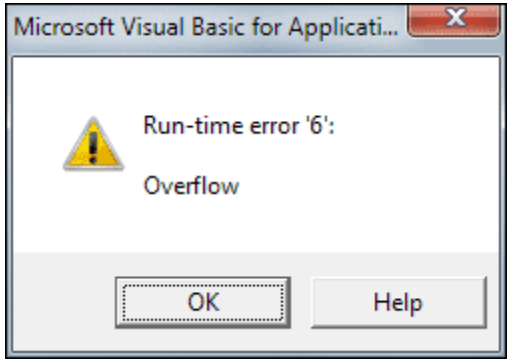


The number 1048576 is coming from our **NumberOfRows** variable. In other words, just over a million rows in this version of Excel (2013).

Click OK on your Message Box to return to your coding window. Now change **As Long** to **As Integer**:

```
Dim NumberOfRows As Integer
```

Try running your Sub again. This time, you should see an error message:



The error is **Overflow**. You get this error because the value you're trying to store in your variable is too big for the variable type. The **As Integer** variable type can only hold numbers up to a value of 32, 767. Storing a value of over a million causes the programme to bail out with Overflow error.

To test out floating point numbers, add a new Sub and call it **Floats**. Add the following code:

```
Dim FloatingPoint As Single
```

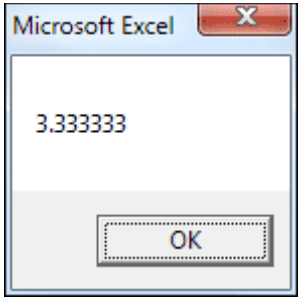
FloatingPoint = 10 / 3

MsgBox FloatingPoint

Your coding windows will then look like this:

```
Sub Floats()

    Dim FloatingPoint As Single

    FloatingPoint = 10 / 3

    MsgBox FloatingPoint

End Sub
```

With your cursor inside of the **Sub** and **End Sub** code, Press F5 on your keyboard to run it. You should see a Message Box appear:



The value in the variable called **FloatingPoint** is being displayed in the Message Box. It is showing the answer to 10 divided by 3 to six decimal places.

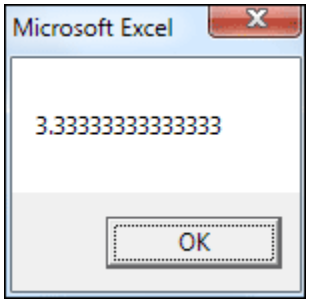
Now change this line in your code:

Dim FloatingPoint As Single

To this:

Dim FloatingPoint As Double.

Run your code again and the Message Box will display the following:



Now the **FloatingPoint** variable is showing the answer to 10 divided by 3 to fourteen decimal places.

So if you want greater accuracy in your calculation, use **As Double** rather **As Single**. Here's one final example of that.

Change your code to this:

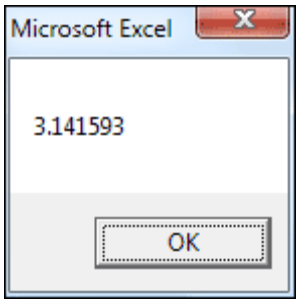
Dim FloatingPoint As Single

FloatingPoint = WorksheetFunction.Pi

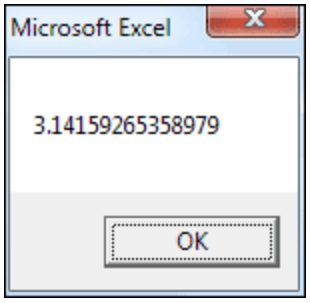
MsgBox FloatingPoint

This time, we're using **WorksheetFunction**. After a dot, you'll see a list of Excel functions you can use. Select **Pi** from the list.

When you run your code, the Message Box will be this:



Now change **As Single** to **As Double**. When you run your code this time, the Message Box will be as follows:



In the **As Single** version, the sixth number after the floating point is a 3. In the **As Double** version, the sixth number is a 2. VBA has rounded the value up for **As Single**.

So, again, if your calculation need to precise, and you don't want Excel to automatically round things up or down, then use **As Double** rather than **As Single**.

One final variable type that can come in handy is **Variant**:

Dim FloatingPoint As Variant

Variant is used when you're not sure what value will be returned. It can hold numbers, text, and objects. However, using it too much can slow down your programmes, as it uses 16 bytes of data for numbers and 22 bytes for text.

Variable Types and Text

If you want your variables to hold strings of text then the variable type to use is As String:

Dim MyText As String

There are lots of inbuilt methods you can use on strings of text, and you'll learn about these later in the course. Mastering these methods will greatly improve your VBA programming skills.

In the next section, you'll learn about Conditional Logic.

8.15 If Statements in Excel VBA

Conditional Logic is all about the IF word. It's about saying what should happen IF a certain condition is met, or what should happen if it's not met.

If Statements

You use Conditional Logic all the time in your daily life. You say things like this:

IF I buy these shoes I will be happier

IF I eat this ice cream I will ruin my diet

IF I go online I might have some emails

Programming in any language is heavily reliant on Conditional Logic like the IF Statement. It allows you to go down different paths, depending on an initial condition.

The structure of a VBA IF Statement looks like this:

If Condition_To_Test Then

```
        'CODE HERE

    End If
```

You start with the word **If** (uppercase "I" lowercase "f"). After a space, you have a condition that you want to test. This conditional is something that can either be TRUE or FALSE. After your condition, you type a space followed by the word **Then** (uppercase "T"). An If Statement ends with the words**End If**.

In Between **If** and **End If** is where you type your code. But this code will only be executed IF your condition is TRUE. If it's FALSE then VBA skips past the End If and continues on its way.
Let's clear things up with a few coding examples. You can start a new spreadsheet for this. When you save the new file, don't forget to save it as an Excel Macro- Enable Workbook. The file ending will then be **XLSM**.

Click the **Developer** tab in Excel, then the **View Code** item on the **Controls** panel:

Clicking **View Code** will open the VBA Editor. If the coding window for Sheet1 is not already open, double click Sheet1 in the Project Explorer on the left. (If you can't see the Project Explorer, click **View > Project Explorer** from the menu at the top.)

Create a new Sub in your coding window, and call it **If_Test_1**. Add the following code for your Sub:

```
    Dim MyNummber As Integer

    MyNumber = 10

    If MyNumber = 10 Then

        MsgBox "Number = 10"

    End If
```

Your coding window will then look like this:

```
Sub If_Test_1()

    Dim MyNummber As Integer

    MyNumber = 10

    If MyNumber = 10 Then
        MsgBox "Number = 10"
    End If

End Sub
```

The first two lines set up an Integer variable called **MyNumber**. We're storing the number 10 in**MyNumber**. Notice the first equal sign:

MyNumber = 10

The equal sign here means "Assign a value of". So we're assign a value of 10 to the variable called**MyNumber**.

The first line of the If statement is this:

If MyNumber = 10 Then

We have the word "If" and the word "Then". Between the two we have the condition we wish to test:

MyNumber = 10

You might think that this is the same as line two from our code. But it's not. It means something entirely different. When you use an equal sign in an If statement it doesn't mean "Assign a value of" anymore, it means "has a value of". What you're saying now is "If MyNumber has a value of 10". The equal sign in an If Statement is known as a Conditional Operator. You'll meet more of these later.

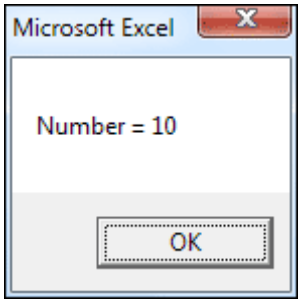
But by saying "If MyNumber has a value of 10" you're creating a statement that can either be TRUE or FALSE. That's what VBA will be checking for: "Can this statement be evaluated as either TRUE or FALSE?"

If **MyNumber** does indeed have a value of 10 then the If Statement is TRUE. In which case the code between **If** an **End If** will get executed. For us, this is just a simple Message Box:

```
MsgBox "Number = 10"
```

We have surrounded Number = 10 with double quotes. This will turn it into text.

With your cursor flashing between the **Sub** and **End Sub** of your code, press F5 on your keyboard to run it. You should see the Message Box display:



Click OK on the message box to return to your coding window. Now change the second line of your code from **MyNumber = 10** to **MyNumber = 11**. Leave the IF Statement alone, though. Now press F5 again to run the code. You should find that nothing happens.

The reason nothing happens is that the message box is enclosed between an **If** and an **End If**. The message box line will only get executed if the If Statement evaluates to TRUE. Since **MyNumber** now equals 11 the If statement will now evaluate to FALSE. A value of FALSE means that VBA can skip past the If statement without executing any of the code between **If** and **End If**.

In the next lesson, you'll see how to extend your If Statements by using Else and ElseIf.

8.16 Excel VBA - Else ... ElseIf

In the last lesson, you saw how to set up an If Statement. In this lesson, we expand the If Statement with Else and ElseIf.

Else

Between **If** and **End If** you can also add an **Else** part. The structure of an **If … Else** Statement looks like this:

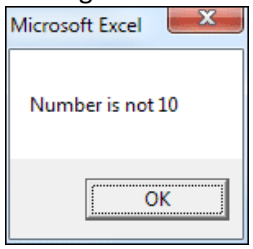
```
If Condition_To_Test Then
Else
End If
```

In our code from the previous lesson, nothing happened because we set **MyNumber** to a value of 11. Our IF Statement only executes when **MyNumber** has a value of 10, making the condition TRUE.

However, instead of VBA skipping past the entire IF Statement, we can add an **Else** part. The Else part is where you tell VBA what should happen if the opening line is FALSE. Change your code from the last lesson to this (the new lines are in bold):

```
Dim MyNummber As Integer

MyNumber = 11

If MyNumber = 10 Then

    MsgBox "Number = 10"

Else

    MsgBox "Number is not 10"

End If
```

What we're saying is, "**If** MyNumber has a value of 10 Then display one message box, **Else** display another message box".
Run your code by pressing F5 on your keyboard. You should see the message from the Else part appear:



Click OK on the message box to return to your coding window. Change the number from the second line of your code from 11 back to 10. When you run your code again, you'll see the first message box.

Using an Else part for your If Statements is handy programming technique to learn.

ElseIf

Not every condition can be reduced to a simple either/or. Quite often, you'll have more than two options you want to check for. In our code, we may want to check for a value of 10, 11, and any other number. This is where **ElseIf** comes in. Here's the structure of an **ElseIf** Statement:

```
If Condition_To_Test Then

ElseIf Condition_To_Test Then

End If
```

You can have more than one **ElseIf** parts, as many as you need, in fact:

```
If Condition_To_Test Then

ElseIf Condition_To_Test Then

ElseIf Condition_To_Test Then
```

```
End If
```

Notice that the only difference between **If** and **ElseIf** is the word **Else** tacked on to the word **If**. You still have a condition to test and the keyword **Then** at the end.

You can also add an Else part, to catch anything you may have missed:

```
If Condition_To_Test Then

ElseIf Condition_To_Test Then

Else

End If
```

To test all this out, add a new Sub. Call it **If_Test_2**. Then add the following code:

```
Dim MyNummber As Integer

MyNumber = 11

If MyNumber = 10 Then

    MsgBox "Number = 10"

ElseIf MyNumber = 11 Then

    MsgBox "Number = 11"

End If
```

Your coding window should look like this:

```
Sub If_Test_2()

    Dim MyNummber As Integer

    MyNumber = 11

    If MyNumber = 10 Then
        MsgBox "Number = 10"

    ElseIf MyNumber = 11 Then
         MsgBox "Number = 11"

    End If

End Sub
```

This is almost the same as before. The difference is the ElseIf part:

ElseIf MyNumber = 11 Then

MsgBox "Number = 11"

First of all, VBA checks the variable **MyNumber** for a value of 10. If this evaluates to TRUE then the code for the IF statement gets executed.
If **MyNumber** doesn't have a value of 10 then VBA drops down to the **ElseIf** part. It then checks **MyNumber** for a value of 11. If this is TRUE then the second message box will display. If neither is TRUE then VBA will skip to the **End If** and continue on its way.

Test it out. Run your code and you'll see **Number = 11** appear in a message box. Now return to your coding window and change the second line from **MyNumber = 11** to **MyNumber = 10**. Run your code again and the first message box displays.

Return to your coding window again and change **MyNumber** to 12. If you try to run your code now, nothing will happen. Nothing happens because both the IF and the ElseIf line evaluate to FALSE.

If MyNumber = 10 Then

MsgBox "Number = 10"

```
ElseIf MyNumber = 11 Then

    MsgBox "Number = 11"

Else

    MsgBox "Not 10 or 11"

End If
```

Run your code again and you'll find that the third message box display. By adding **Else** you've guaranteed that at least one part of your If statement will be TRUE.

In the next lesson, you'll learn about the conditional operators that are used with If Statements.

8.17 Excel VBA Conditional Operators

In the previous lessons on If Statements, you've already used one conditional operator - the equal sign. But there are others. Here's a list:

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

By using the various Operators, you can set up more sophisticated conditions for your If Statements.

Add another Sub to your code from the previous lessons. Call it **If_Test_3**. As the code for your new Sub, add the following:

Dim MyNummber As Integer

MyNumber = 10

If MyNumber < 20 Then

MsgBox MyNumber & " is Less than 20"

End If

Your coding window should look like this:

```
Sub If_Test_3()

    Dim MyNummber As Integer

    MyNumber = 10

    If MyNumber < 20 Then
        MsgBox MyNumber & " is Less than 20"
    End If

End Sub
```

Again, we've set up an Integer variable called **MyNumber** and stored a value of 10 in it. Look at the first line of the If Statement, though:

If MyNumber < 20 Then

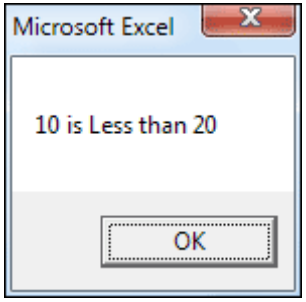
If you consult the table above, you'll see the **<** symbol means **Less Than**. So the condition to test is, "If MyNumber is less than 20". If **MyNumber** is indeed less than 20 then the condition evaluates to TRUE. In which case, the code between **If** and **End If** gets executed. If it's FALSE then VBA will skip to any lines after **End If**.

One other thing to note about the code is the message box line:

MsgBox MyNumber & " is Less than 20"

After MsgBox we have the variable **MyNumber** then a space then an ampersand (**&**) symbol. The **&**symbol is used to concatenate (join together) things. After the **&** symbol we then have another space followed by some direct text in double quotes. So whatever is in the variable **MyNumber** will get joined with the direct text in double quotes.

Run your code and you should see the message box display the following:



Return to your coding window and change **MyNumber = 10** on the second line to **MyNumber = 20**.

Run your code again and you'll find that nothing happens.

The reason nothing happens is that 20 is not less than 20, so the If Statement is FALSE. We haven't got **ElseIf** or **Else** parts to check for a TRUE value, so the code just ends.

Now change the **<** symbol to **<=**. The **<=** symbols together mean "Less than or equal to". Change you message to this:

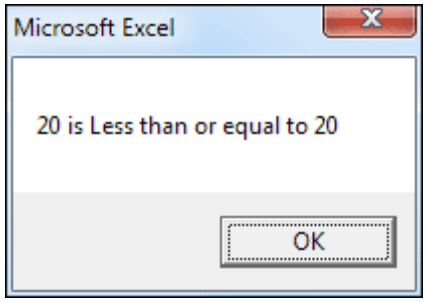
MsgBox MyNumber & " is Less than or equal to 20"

Your code should now be the same as ours below:

```
Dim MyNummber As Integer

MyNumber = 20
```

```
If MyNumber <= 20 Then

    MsgBox MyNumber & " is Less than or equal to 20"

End If
```

When you run your code, here's what the message box will look like

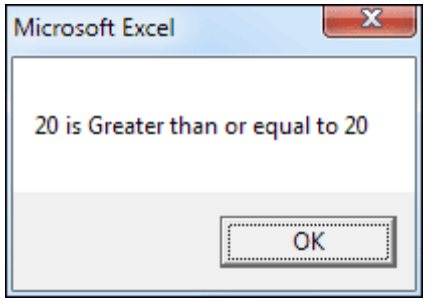


Click OK and return to your code. Change the **<=** symbol to **>=**. The **>=** symbols together mean "Greater than or equal to".

Change your message text to this:

```
MsgBox MyNumber & " is Greater than or equal to 20"
```

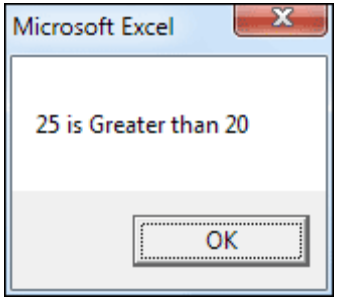
When you run your code, you'll see the message box appear again:



To test out the **Greater Than** sign by itself delete the = sign next to the > symbol. Change MyNumber to 25. Amend the text in your message. Your code will then look like this:

```
Dim MyNummber As Integer

MyNumber = 25

If MyNumber > 20 Then

    MsgBox MyNumber & " is Greater than 20"

End If
```

Run your code again to see the new message:



You can have ElseIf and Else parts with your Conditional Logic. Take the following as an example:

```
Dim MyNummber As Integer

MyNumber = 19

If MyNumber = 20 Then

    MsgBox MyNumber & " is 20"
```

```
ElseIf MyNum > 20

    MsgBox MyNumber & " is Greater Than 20"

Else

    MsgBox MyNumber & " is below 20"

End If
```

In the code above, we have an **If** and **ElseIf** and **Else**. The If Statement tests for a value of exactly 20. The **ElseIf** tests if the variable **MyNumber** is Greater Than 20. The **Else** part catches any other value. Try it out and see how it works. Change the value of the variable **MyNumber** and run your code again.

In the next lesson, we'll take a look at the Logic Operators.

8.18 Logical Operators in Excel VBA

In the previous lesson, you used the Conditional Operators. In this lesson, you'll learn about the Logic Operator.

You can have more than one condition on the same line of your If and ElseIf Statements. To test for more than one condition, you need the Logic Operators. Here's a table of them:

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

Only the first three are used regularly. Let's see how they works. Before doing so, however, there's one more important variable type we need to discuss - Boolean values.

Boolean Values

You can set up something called a Boolean variable. Boolean variables have only two values: either**true** or **false**. You set them up like this:

```
Dim BooleanFlag As Boolean

BooleanFlag = True
```

Here, we set up a variable called **BooleanFlag**. Instead of setting the variable to **As Integer** or **As Long** we've set this one up with **As Boolean**. The second line puts something in the variable. But notice that that something is a value of **True**. The only other value we could have used here is **False**.

You can use your Boolean values like this:

```
If BooleanFlag = True Then

    MsgBox "It's True"

Else

    MsgBox "It's False"

End If
```

The first line test if the variable called **BooleanFlag** is equal to a value of **True**. If it is, then we display a message. Because there are only two options, we can just have an Else part to test for all other values, as any other value will be **False**.

You can miss out the **= True** part, if you like, and just have this:

```
If BooleanFlag Then
```

VBA will then take this to mean "If BooleanFlag has a value of True".

With all that in mind, let's take a look at the Not operator.

Logical Not

The Not operator is used to test if a value or variable is NOT something. We'll use our **BooleanFlag**variable from above to demonstrate this concept.

Create a new Sub in your code code window from the previous section. Call it **Bool_Test** and add the following code:

```
Dim BooleanFlag As Boolean

BooleanFlag = True

If BooleanFlag = True Then

    MsgBox "It's True"

Else

    MsgBox "It's False"

End If
```

Your coding window will then look like this:

```
Sub Bool_Test()

    Dim BooleanFlag As Boolean
    BooleanFlag = True

    If BooleanFlag = True Then

        MsgBox "It's True"

    Else
        MsgBox "It's False"

    End If

End Sub
```

Press F5 to run your code and you should see the first message box appear saying "It's True".

Now change the first line of your If Statement to this:

If Not BooleanFlag = True Then

We're using the Logic Operator **Not** after the word If. Before we added **Not**, the If Statement read "If BooleanFlag has a value of True". By place the word **Not** before **BooleanFlag** we're now saying, "If BooleanFlag DOES NOT has a value of True". Since **BooleanFlag** actually does have a value of True then the Else part gets executed.

Run your code again and you should see the second message box appear, saying "It's False".

Logical And

The **And** operator test if two or more conditions are true. If, for example, you wanted to check if a number was greater than 20 but less than 30 you can use the And operator to test this condition:

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
```

The word And goes between your two conditions. Be careful of doing this, though:

```
If Age > 20 And < 30 Then
```

Here, we've only used the variable **Age** once. But you need to type your variable twice, once before the **And** and once after it.

Only if both conditions evaluate to TRUE does the entire line become TRUE. If one of them is FALSE then the entire line is FALSE.

You can test for more than two conditions:

```
If Age > 20 And Age < 30 And BooleanFlag = True Then
```

In the code above, we're test three things: **Age** has to be greater than 20, **Age** has to be less than 30,**BooleanFlag** has to have a value of true. All three conditions have to evalute to TRUE before the entire line is TRUE.

Logical Or

With Logical And, you're testing if two or more conditions are true. Only if BOTH conditions are true does the entire If Statement become true. Logical Or, on the other hand, tests if just one of two or more conditions are true. The whole of the If Statement becomes true if at least one of the conditions are true.

To clear this up, create a new Sub in your code window. Call it **Test_Or**. Add the following between Sub and End Sub:

```
Dim FirstNumber As Integer
Dim SecondNumber As Integer

FirstNumber = 10
SecondNumber = 20

If FirstNumber = 10 Or SecondNumber = 20 Then

    MsgBox "Valid Number"

Else

    MsgBox "Non Valid Number"

End If
```

The code just sets up two integer variables, **FirstNumber** and **SecondNumber**. We have a value of 10 in **FirstNumber** and a value of 20 in **SecondNumber**. The IF Statement is trying to test what's in these numbers:

```
If FirstNumber = 10 Or SecondNumber = 20 Then
```

The valid numbers are 10 and 20. We don't really care if **FirstNumber** AND **SecondNumber** hold 10 and 20. Just as long as at least one of them holds the correct number, then that's OK.
Run your code and you should see the first message box display, "Valid Number". Now change your code so that FirstNumber holds a value of 30. Run the programme again and you'll find the first message box still displays. However, change the value of **SecondNumber** to 40 and now neither number holds the correct value. In which case, the second message box will display when the programme is run.

Nested If

You can nest one (or even more than one) If Statement inside another. Examine the following code:

```
Sub Nested_Ifs()

    Dim score As Integer

    score = 27

    If score > 20 And score < 30 Then

        If score < 25 Then
            MsgBox "A-"
        Else
            MsgBox "A+"
        End If

    Else

        MsgBox "Not between 20 and 30"

    End If

End Sub
```

An Integer variable has been set up called **score**. This has been set to a value of 27. The first If Statement, the outer one, is this:

If score > 20 And score < 30 Then

So we want to test score to see if it's greater than 20 AND less than 30. If it's not, then we have an Else part:

Else

MsgBox "Not between 20 and 30"

However, If score is indeed between 20 and 30 then the outer If Statement evaluates to TRUE. In which case, the code that gets executed is another If Statement. This one:

If score < 25 Then

MsgBox "A-"

```
    Else

        MsgBox "A+"

    End If
```

This new If Statement checks to see if **score** is less than 25. If it is, then we know that the variable called **score** is between 20 and 25. A score of between 20 and 25 gets the message ""A-". Any other result and we know that score is between 25 and 30. In which case, the message is "A+".

Nested If Statements are a great way to narrow down your search for a particular value. They can be tricky to use, though. But stick with them and your programming skills will come on a treat!

In the next part below, we'll take a look at some more practical ways to use the lessons you have learned in this and previous sections.