Supplemental lecture notes for: "Introduction to Excel VBA Programming"

Written by:
Paul Nissenson, Ph.D.
Department of Mechanical Engineering
California State Polytechnic University, Pomona

Introduction to the course

In this course, you will learn some of the basics of computer programming using Excel and the programming language VBA (Visual Basic for Applications).

No computer programming language is perfect for every application. Here are examples of other languages:
- MATLAB is great at manipulating matrices and is very user-friendly.
- Fortran and C are great at number crunching, but are not as user friendly.
- Maple and Mathematica are great at symbolic math and are very user-friendly.

Some advantages of Excel

- Very user-friendly
- Contains many built-in functions
- Already familiar to many students
- Many students own a copy of Excel already

Some advantages of VBA

- It is a relatively simple language to learn
- Allows you to manipulate data in Excel spreadsheets, so you can add functionality to Excel
- Introduces students to object-oriented programming (although we won't be focusing on this aspect in the course)
- Allows you to create graphical user interfaces, which make programs user-friendly

VBA is a "high-level" language. You will be writing code as text that will be translated into a lower-level language the computer can understand in order to perform calculations.

Humans are great at thinking about complicated, abstract concepts in which many things are changing at once. However, computers are only able to execute commands one line at a time. One of the great challenges of computer programming is taking an abstract concept, such as finding the third largest number in a data set, and breaking that concept down to a set of commands that will be executed one at a time. If you want a 5 year old child (who could use a calculator perfectly) to find the third largest number in a data set, what commands would you give to the child? You have to dumb yourself down in an intelligent manner.

About these notes

These notes were created using Excel 2007 for Windows, but most of the concepts are valid for Excel 2010 for Windows, Excel 2013 for Windows, and Excel 2011 for Macs as well.

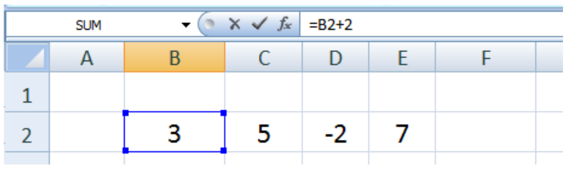
This is the first draft of the notes, so please forgive any minor errors that may exist. If you find errors, please inform the author at paul.m.nissenson@gmail.com

Topic 1: Introduction to the Excel workbook environment

After starting Excel, you will be presented with a worksheet. Each worksheet consists of many cells. Each cell has a row number and column letter. For example, cell B3 is in the 3rd row, 2nd column. Data can be entered and manipulated in these cells.

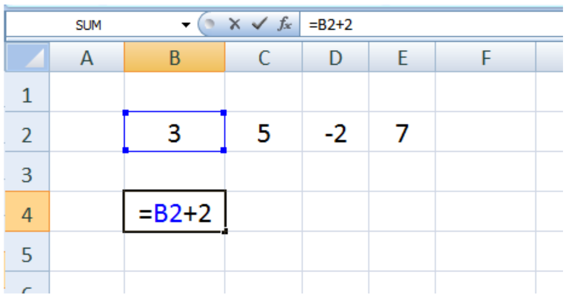
Entering and manipulating data

You can enter data into a cell by selecting a cell, then typing a number in the cell and hitting the Enter key. In this example, four values are input into cells B2 to E2.



	A	B	C	D	E	F
1						
2		3	5	-2	7	

Here are the mathematical operators in Excel:
+ addition
– subtraction
*** multiplication**
/ division
^ exponentiation



	A	B	C	D	E	F
1						
2		3	5	-2	7	
3						
4		=B2+2				
5						

In cell B4, type **=B2+2** and hit the Enter key (be sure to include the equal sign). The value in B2, which is 3, is added to 2. The resulting value, 5, is displayed in B4.

Try changing the value in B2 and see what happens to the value in B4.

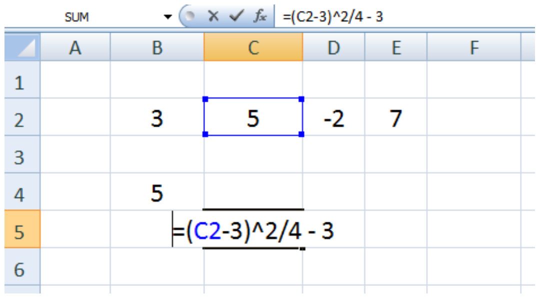
If you want to store very large or very small numbers, you can use **scientific notation**.
1.25E5 is equivalent to 1.25*10^(+5), or 125000
1.25E-5 is equivalent to 1.25*10^(-5), or 0.0000125

Order of operations

Use the Please Excuse My Dear Aunt Sally rules from algebra.

Parentheses	**(highest priority)**
Exponentiation	
Multiplication and Division	**(same priority, executed left to right)**
Addition and Subtraction	**(same priority, executed left to right)**

In C5, type **=(C2-3)^2/4 - 3** and hit enter. Do you understand why the answer is -2?



SUM | =(C2-3)^2/4 - 3

	A	B	C	D	E	F
1						
2		3	5	-2	7	
3						
4		5				
5			=(C2-3)^2/4 - 3			
6						

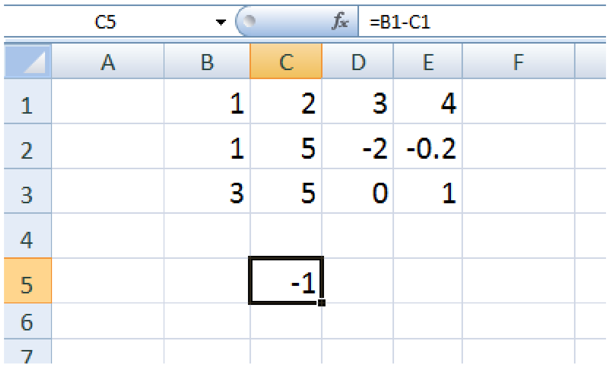
Saving Excel workbooks

You can save the data stored in an excel workbook by typing **Ctrl+s** or by clicking on the **Office** button and choosing the **Save** option.

Excel workbooks usually have a **.xls** or **.xlsx** extension by default. Later, when we create programs called macros, we will be saving Excel workbooks with a **.xlsm** extension.

Relative referencing versus absolute referencing

12 values are input into the worksheet below.

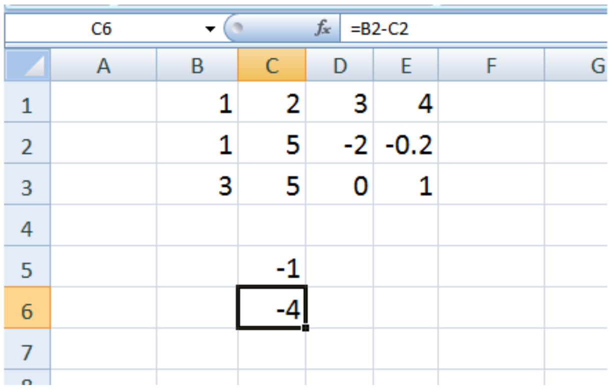


C5 | =B1-C1

	A	B	C	D	E	F
1		1	2	3	4	
2		1	5	-2	-0.2	
3		3	5	0	1	
4						
5			-1			
6						
7						

The value 1 is located in cell B1 and the value 2 is located in cell C1. We can use this data to perform calculations by clicking C5 and typing **=B1-C1** (make sure that you include the equal sign). This formula takes the value stored in C1 and subtracts it from the value in B1. The answer, -1, is displayed in C5.

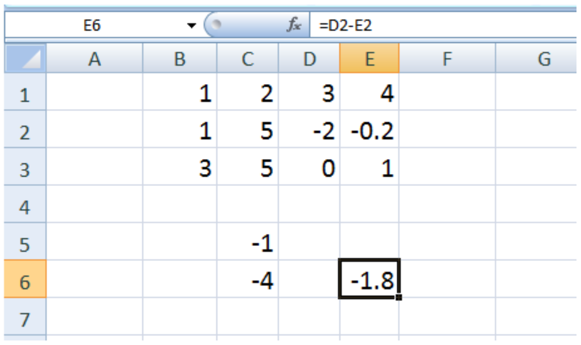
What happens if we click on C5, then copy (**Ctrl+c**) and paste (**Ctrl+v**) the contents of the cell into D6?



C6 =B2-C2

	A	B	C	D	E	F	G
1		1	2	3	4		
2		1	5	-2	-0.2		
3		3	5	0	1		
4							
5			-1				
6			-4				
7							

The value stored in C6 is not B1-C1, but rather B2-C2. What if we copy and paste the contents of C5 into E6?



E6 =D2-E2

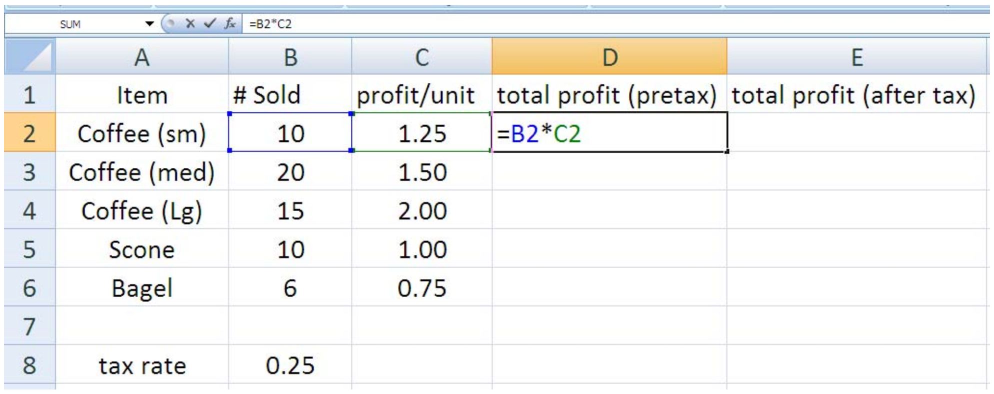
	A	B	C	D	E	F	G
1		1	2	3	4		
2		1	5	-2	-0.2		
3		3	5	0	1		
4							
5			-1				
6			-4		-1.8		
7							

The value stored in E6 is D2-E2, not B1-C1.

This occurs because Excel is assuming we want to use **relative references**.

Although we typed **=B1+C1** in cell C5, Excel interprets this expression as "take the value 1 column to the left and 4 rows up, and subtract from that the value in the same column and 4 rows up." When we copy and paste C5 into E6, we are copying those instructions.

Relative referencing can be very useful. For example, say you own a bagel shop that sells five items. You know how much of each item you sold and how much each item costs, and you want to find out how much money you made per item. Start with the first item (small coffee) and multiply the number of items sold in B2 with the cost per item in C2 by typing the formula **=B2*C2** into cell D2.

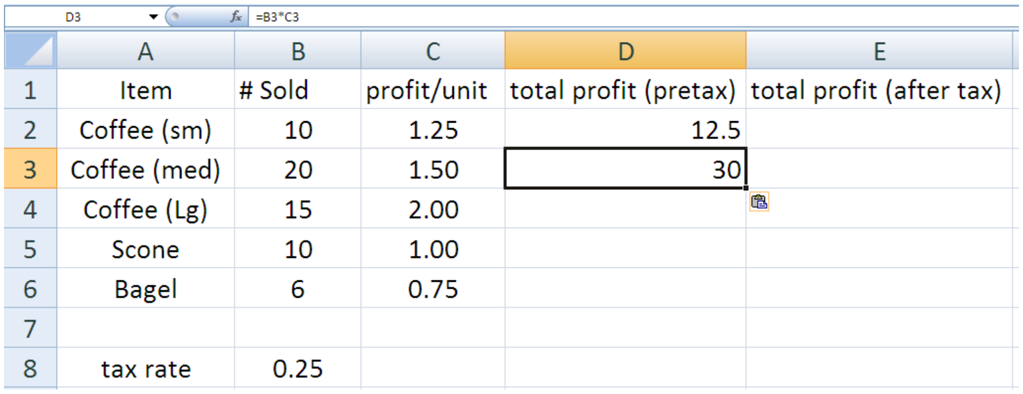


SUM =B2*C2

	A	B	C	D	E
1	Item	# Sold	profit/unit	total profit (pretax)	total profit (after tax)
2	Coffee (sm)	10	1.25	=B2*C2	
3	Coffee (med)	20	1.50		
4	Coffee (Lg)	15	2.00		
5	Scone	10	1.00		
6	Bagel	6	0.75		
7					
8	tax rate	0.25			

Remember that **=B2*C2** actually means that you want to take the value in the cell that is 2 columns to the left and in the same row (B2), and multiply it by the value in the cell that is 1 column to the left and in the same row (C2).

If you copy D2 and paste the contents into D3, you will obtain the product of the cell that is 2 columns to the left and in the same row (B3), and multiply it by the value in the cell that is 1 column to the left and in the same row (C3).



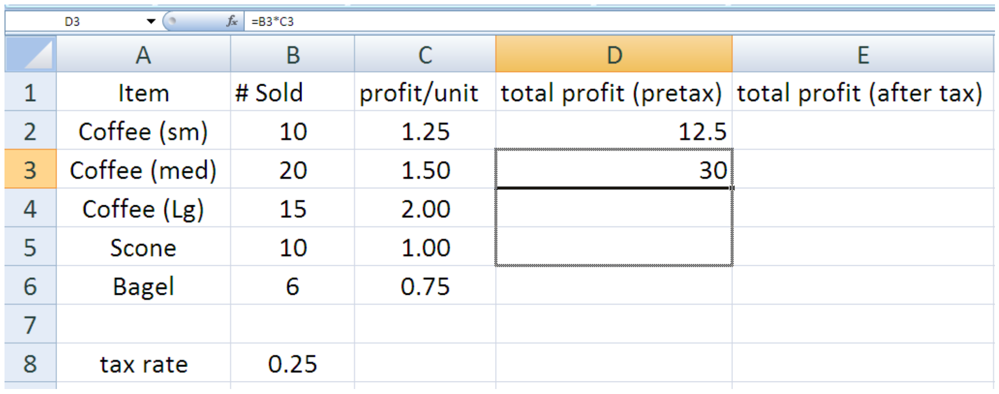
D3 =B3*C3

	A	B	C	D	E
1	Item	# Sold	profit/unit	total profit (pretax)	total profit (after tax)
2	Coffee (sm)	10	1.25	12.5	
3	Coffee (med)	20	1.50	30	
4	Coffee (Lg)	15	2.00		
5	Scone	10	1.00		
6	Bagel	6	0.75		
7					
8	tax rate	0.25			

In this example, we only have five items we need to calculate the total cost for. Copying and pasting four times (for D3, D4, D5, and D6) would not take very long. But what if we sold a lot of items in our bagel shop?

We can copy and paste much faster by using the **fill handle**.

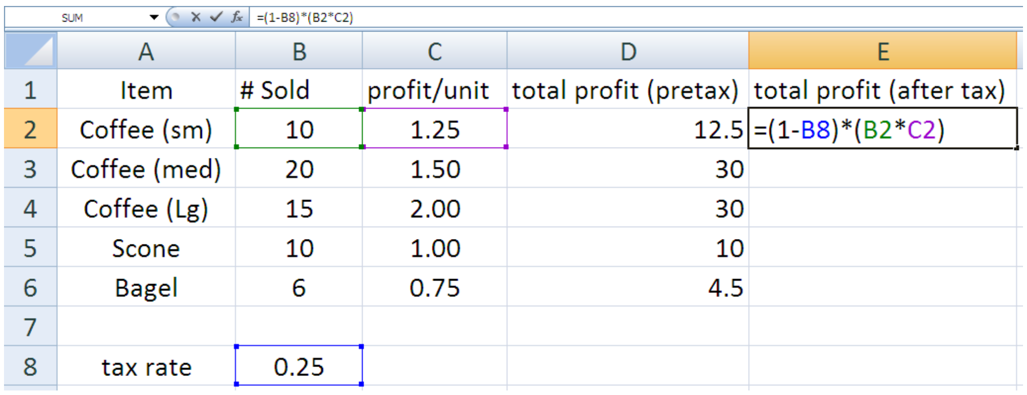
Click the cell you would like to copy (D3) and place the cursor over the lower right corner of that cell. The cursor should change into a black cross. Click and hold while simultaneously dragging the cursor downward, filling in all the cells you would like to paste into.



D3 =B3*C3

	A	B	C	D	E
1	Item	# Sold	profit/unit	total profit (pretax)	total profit (after tax)
2	Coffee (sm)	10	1.25	12.5	
3	Coffee (med)	20	1.50	30	
4	Coffee (Lg)	15	2.00		
5	Scone	10	1.00		
6	Bagel	6	0.75		
7					
8	tax rate	0.25			

Now we will calculate the total profit after taxes.



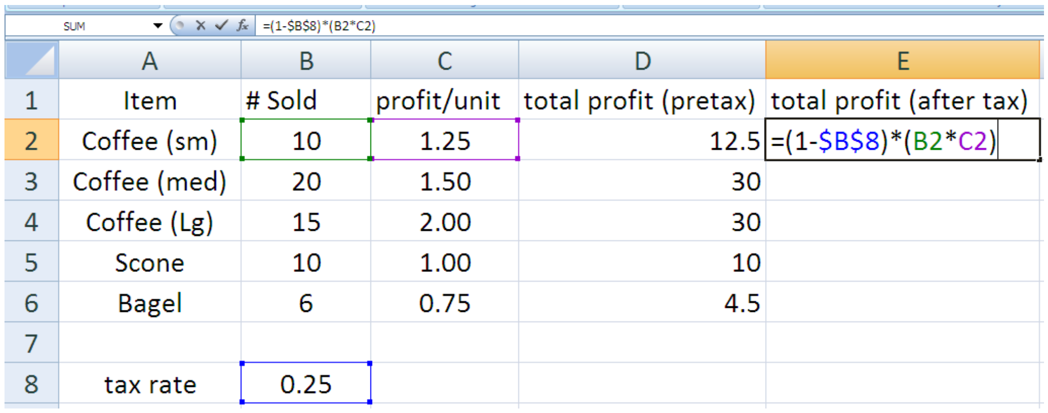
SUM =(1-B8)*(B2*C2)

	A	B	C	D	E
1	Item	# Sold	profit/unit	total profit (pretax)	total profit (after tax)
2	Coffee (sm)	10	1.25	12.5	=(1-B8)*(B2*C2)
3	Coffee (med)	20	1.50	30	
4	Coffee (Lg)	15	2.00	30	
5	Scone	10	1.00	10	
6	Bagel	6	0.75	4.5	
7					
8	tax rate	0.25			

Try to copy and paste the E2 to E3. The answer is 30 because the value in B9 is 0.

We need to make sure the value for the tax rate does not change. If you want to use data from a specific row, column, or cell that never changes, you can use **absolute references**.

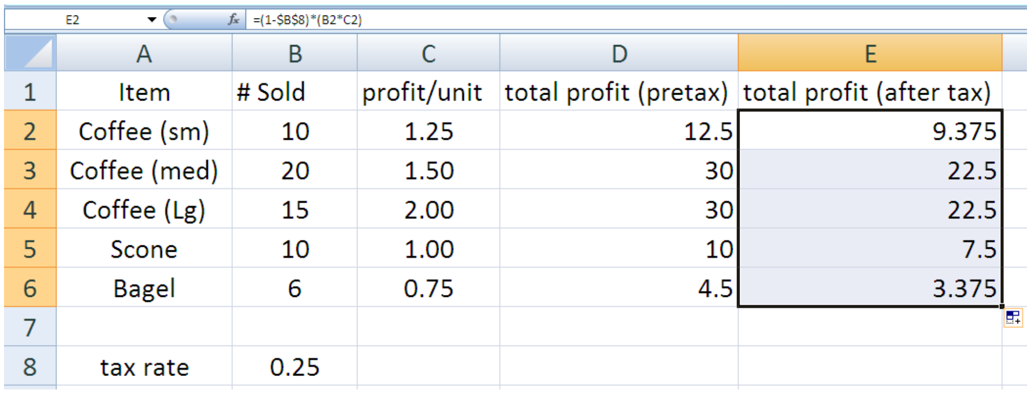
Absolute references require the use a dollar sign (**$**) in front of the row and/or column. Let's go back to the bagel shop problem and see what happens if we use absolute references instead of relative references for the tax rate.



SUM =(1-$B$8)*(B2*C2)

	A	B	C	D	E
1	Item	# Sold	profit/unit	total profit (pretax)	total profit (after tax)
2	Coffee (sm)	10	1.25	12.5	=(1-$B$8)*(B2*C2)
3	Coffee (med)	20	1.50	30	
4	Coffee (Lg)	15	2.00	30	
5	Scone	10	1.00	10	
6	Bagel	6	0.75	4.5	
7					
8	tax rate	0.25			

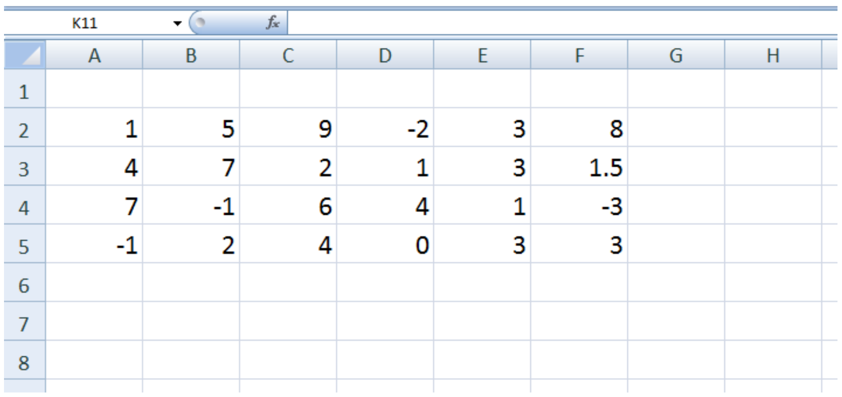
Use the fill handle to calculate the total profit per item quickly.



E2 =(1-$B$8)*(B2*C2)

	A	B	C	D	E
1	Item	# Sold	profit/unit	total profit (pretax)	total profit (after tax)
2	Coffee (sm)	10	1.25	12.5	9.375
3	Coffee (med)	20	1.50	30	22.5
4	Coffee (Lg)	15	2.00	30	22.5
5	Scone	10	1.00	10	7.5
6	Bagel	6	0.75	4.5	3.375
7					
8	tax rate	0.25			

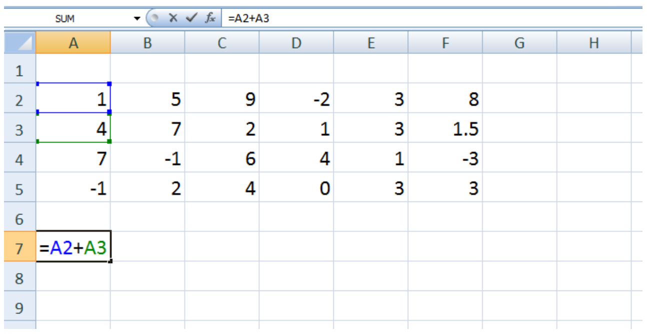
The next example shows how Excel handles a combination of relative and absolute references. 24 values have been input into the Excel spreadsheet below.



K11 | fx

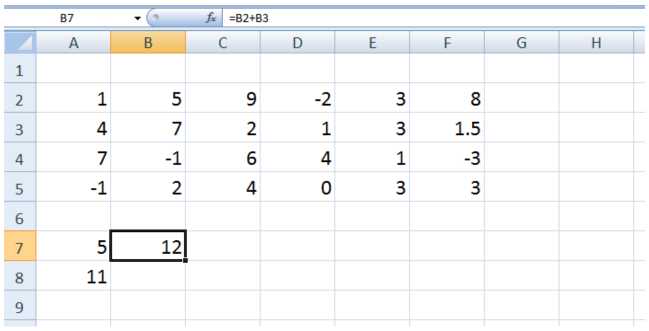
	A	B	C	D	E	F	G	H
1								
2	1	5	9	-2	3	8		
3	4	7	2	1	3	1.5		
4	7	-1	6	4	1	-3		
5	-1	2	4	0	3	3		
6								
7								
8								

Type **=A2+A3** into cell A7, hit enter, then the copy and paste the contents of A7 into cells A8 and B7.



SUM | fx =A2+A3

	A	B	C	D	E	F	G	H
1								
2	1	5	9	-2	3	8		
3	4	7	2	1	3	1.5		
4	7	-1	6	4	1	-3		
5	-1	2	4	0	3	3		
6								
7	=A2+A3							
8								
9								

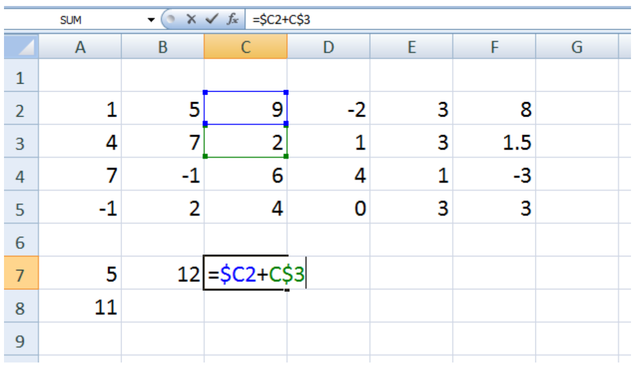


B7 | fx =B2+B3

	A	B	C	D	E	F	G	H
1								
2	1	5	9	-2	3	8		
3	4	7	2	1	3	1.5		
4	7	-1	6	4	1	-3		
5	-1	2	4	0	3	3		
6								
7	5	12						
8	11							
9								

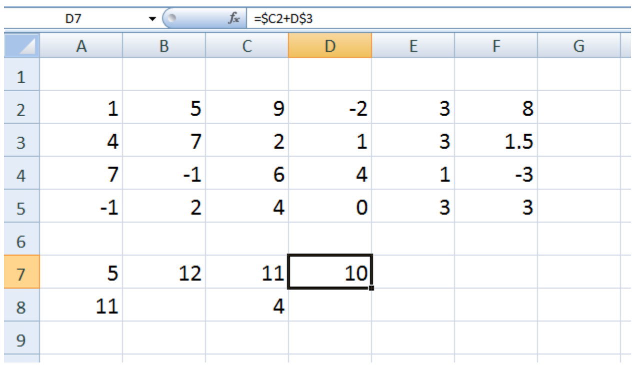
Remember that we are using relative references. The instructions **=A2+A3** mean "take the value stored in the cell that is in the same column, 5 rows up and add it to the value stored in the cell that is in the same column, 4 rows up." The value stored in B7 is the sum of B2 and B3. The value stored in A8 is the sum of A3 and A4.

Now go to cell C7 and type the statement **=$C2+C$3**, hit the Enter key, and copy and paste the contents of C7 into cells D7 and C8.



SUM | fx =$C2+C$3

	A	B	C	D	E	F	G
1							
2	1	5	9	-2	3	8	
3	4	7	2	1	3	1.5	
4	7	-1	6	4	1	-3	
5	-1	2	4	0	3	3	
6							
7	5	12	=$C2+C$3				
8	11						
9							



D7 | fx =$C2+D$3

	A	B	C	D	E	F	G
1							
2	1	5	9	-2	3	8	
3	4	7	2	1	3	1.5	
4	7	-1	6	4	1	-3	
5	-1	2	4	0	3	3	
6							
7	5	12	11	10			
8	11		4				
9							

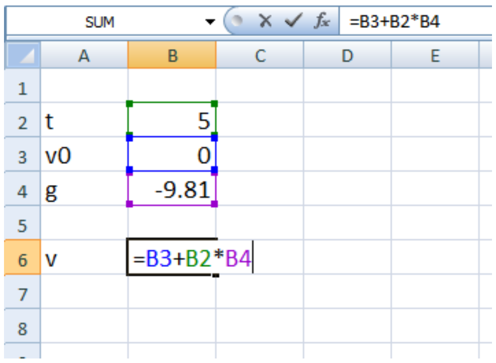
Here we apply relative and absolute references to columns and rows. The statement **=$C2+C$3** means "take the value stored in column C, 5 rows up and add it by the value stored in the same column, row 3."

The value stored in cell D7 (10) was calculated by adding the value stored in column C, 5 rows up (cell C2, value is 9) to the value stored in the same column, row 3 (cell D3, value is 1).

The value stored in cell C8 (4) was calculated by adding the value stored in column C, 5 rows up (cell C3, value is 2) to the value the value stored in the same column, row 3 (cell C3, value is 5).

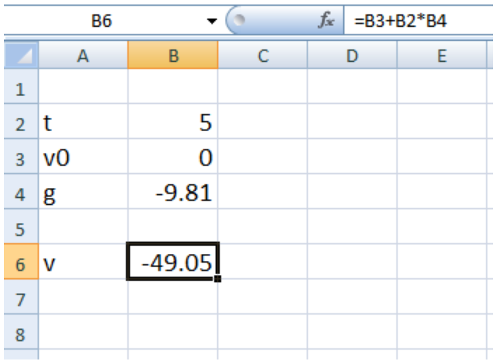
Defining cell names

It is often convenient to use symbols to represent cells instead of writing cell addresses. For example, say you have information about a ball dropping from a building. You know the initial velocity (**v0**), the gravitational acceleration (**g**), and time since the ball was released (**t**). If you want to calculate the velocity at time t using the equation v0+g*t, you can use cell addresses.



SUM =B3+B2*B4

	A	B	C	D	E
1					
2	t	5			
3	v0	0			
4	g	-9.81			
5					
6	v	=B3+B2*B4			
7					
8					

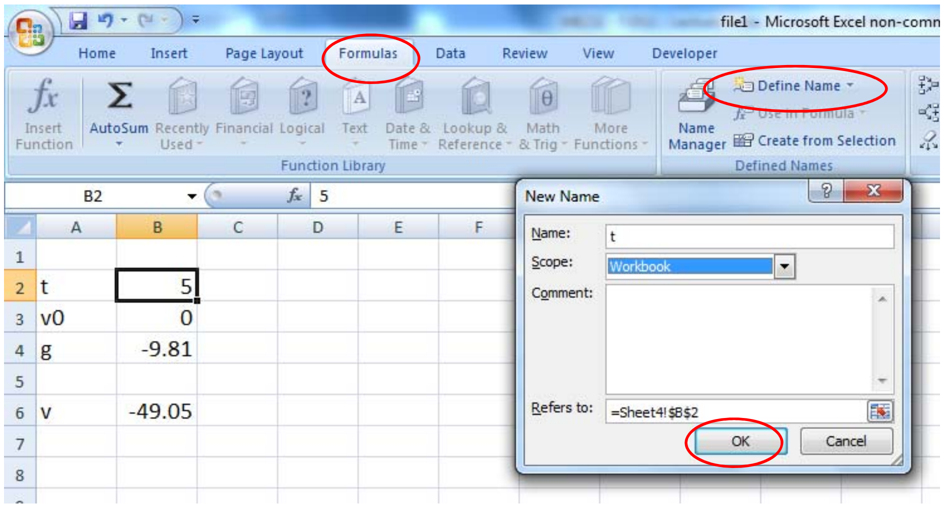


B6 =B3+B2*B4

	A	B	C	D	E
1					
2	t	5			
3	v0	0			
4	g	-9.81			
5					
6	v	-49.05			
7					
8					

Or you can define names to cells B2, B3, and B4. There are three ways to define a cell name.

Option 1: Select cell B2, click on the Formulas tab at the top of the screen, then click on "Define Name". Choose the Name of the cell (t), then select OK. Notice that the cell name t refers to **=Sheet4!$B$2**. This just means that cell B2 of Sheet4 contains the value that will be associated with t (On my computer, I created this example on a worksheet called "Sheet4"). Do not worry about the Scope dropdown menu.



B2 5

	A	B	C	D	E	F
1						
2	t	5				
3	v0	0				
4	g	-9.81				
5						
6	v	-49.05				
7						
8						

New Name

Name: t

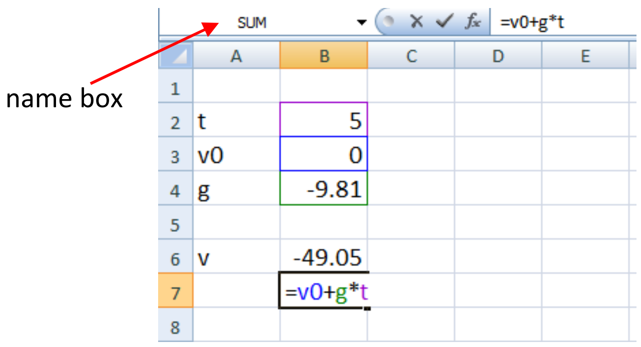
Scope: Workbook

Comment:

Refers to: =Sheet4!$B$2

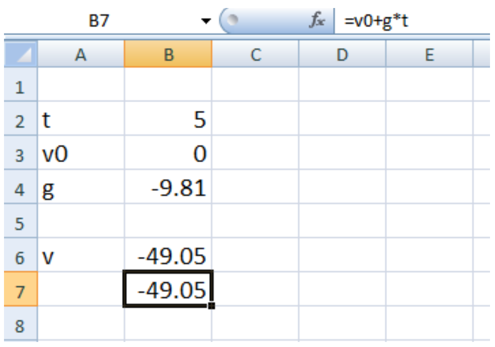
Option 2: Right-click on the cell, then select "Name a Range". Give cell B3 the name "v0"

Option 3: Type the cell name into the **name box**. Give cell B4 the name "g".



SUM | =v0+g*t

	A	B	C	D	E
1					
2	t	5			
3	v0	0			
4	g	-9.81			
5					
6	v	-49.05			
7		=v0+g*t			
8					

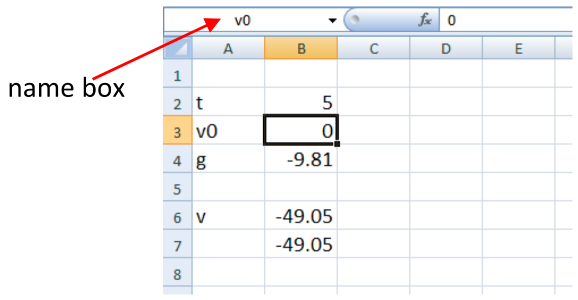


B7 | =v0+g*t

	A	B	C	D	E
1					
2	t	5			
3	v0	0			
4	g	-9.81			
5					
6	v	-49.05			
7		-49.05			
8					

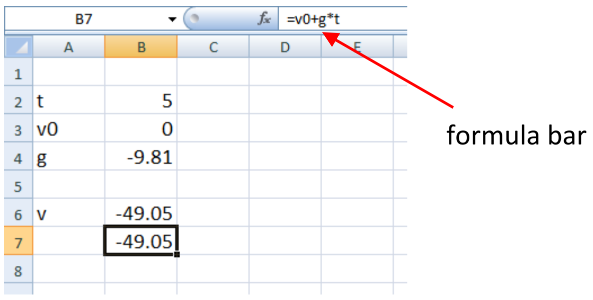
Some rules about defining names:
- The first character of a name must be a letter, an underscore character (_), or a backslash (\). Remaining characters in the name can be letters, numbers, periods, and underscore characters.
- You cannot use a cell name (e.g., A3, T23, v1, AC1)
- Names cannot be the same as a cell reference, such as Z$100
- Spaces are not allowed as part of a name. Use the underscore character (_) and period (.) as word separators, such as, Sales_Tax or First.Quarter
- A name can contain up to 255 characters
- Excel does not distinguish between uppercase and lowercase characters in names. For example, if you created the name "Sales" and then create another name called "SALES" in the same workbook, Excel prompts you to choose a unique name.

You can see if a cell has a name associated with it by selecting a cell and looking at the name box. You can see the expression used in a given cell by selecting a cell and looking at the **formula bar**.



v0 | 0

	A	B	C	D	E
1					
2	t	5			
3	v0	0			
4	g	-9.81			
5					
6	v	-49.05			
7		-49.05			
8					



B7 | =v0+g*t

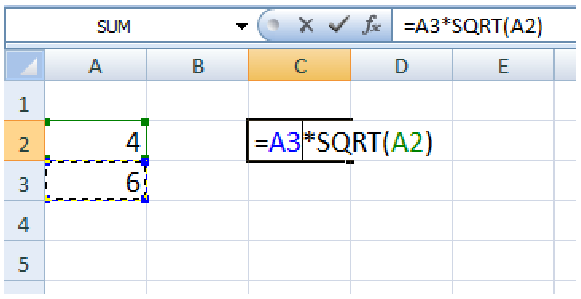
	A	B	C	D	E
1					
2	t	5			
3	v0	0			
4	g	-9.81			
5					
6	v	-49.05			
7		-49.05			
8					

Excel's built-in functions

Excel contains many built-in functions. Here are some common math functions:

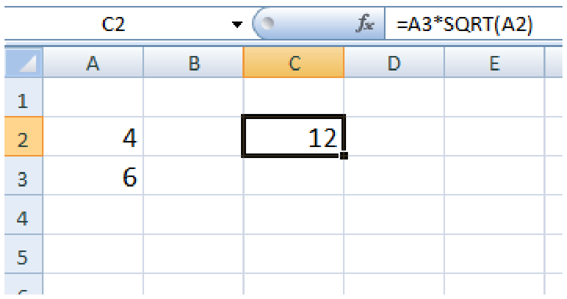
SIN(x), COS(x), SQRT(x), EXP(x), LN(x), LOG10(x)

Each of these functions take as input an **argument** (x) and calculates a single value which can be used in a formula. For example, in the example below, the value of A2 (4) is the argument of the SQRT() function. The value of SQRT(4), which is 2, is calculated and multiplied by the value stored in A3, which is 6.



SUM | =A3*SQRT(A2)

	A	B	C	D	E
1					
2	4		=A3*SQRT(A2)		
3	6				
4					
5					



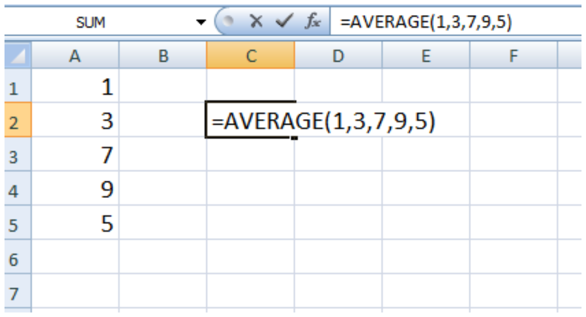
C2 | =A3*SQRT(A2)

	A	B	C	D	E
1					
2	4		12		
3	6				
4					
5					

Note: The arguments of all trigonometric functions (e.g., SIN(), COS(), etc.) must be in radians, not degrees.

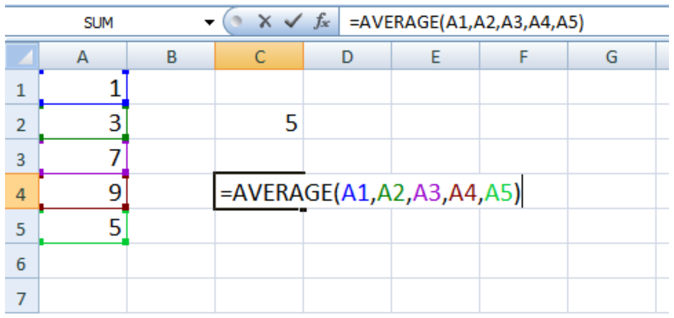
There are some functions with multiple arguments. For example, the AVERAGE() function takes as input one or more arguments and calculates their mean. Multiple arguments are separated by commas.

For example, say I want to calculate the average of cells A1 through A5. I could…
(1) Type the values in manually
(2) Type the cell addresses manually



SUM | =AVERAGE(1,3,7,9,5)

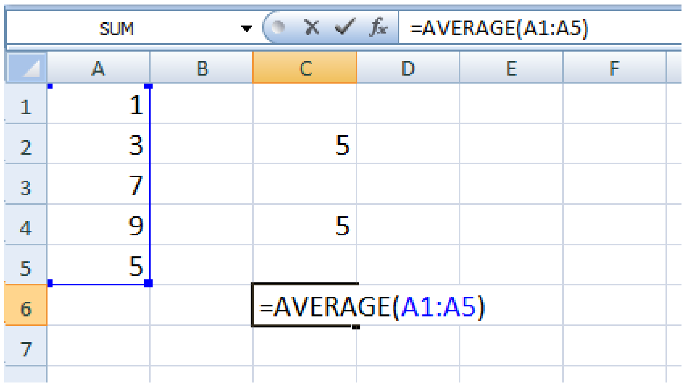
	A	B	C	D	E	F
1	1					
2	3		=AVERAGE(1,3,7,9,5)			
3	7					
4	9					
5	5					
6						
7						



SUM | =AVERAGE(A1,A2,A3,A4,A5)

	A	B	C	D	E	F	G
1	1						
2	3		5				
3	7						
4	9		=AVERAGE(A1,A2,A3,A4,A5)				
5	5						
6							
7							

(3) Use the colon operator (:) to select all the values between A1 and A5.



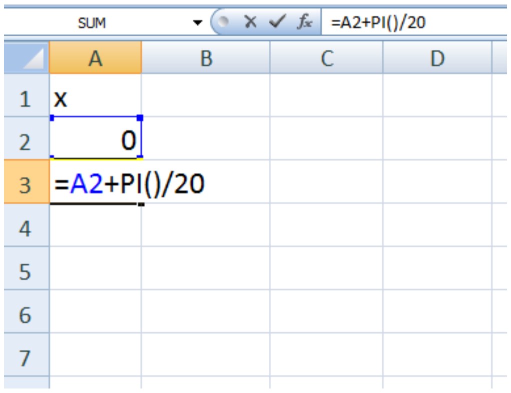
SUM | =AVERAGE(A1:A5)

	A	B	C	D	E	F
1	1					
2	3		5			
3	7					
4	9		5			
5	5					
6			=AVERAGE(A1:A5)			
7						

Plotting

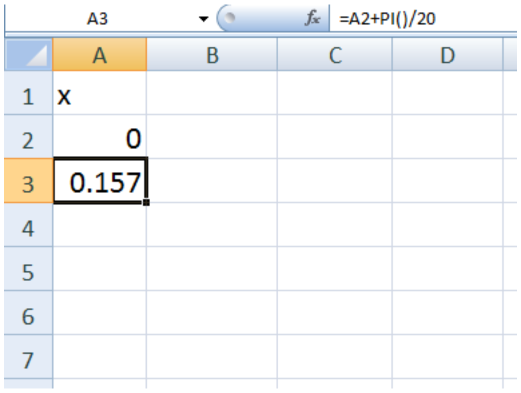
Excel allows you to create many different types of plots very easily. First, we create a table of x values going from x=0 to x=π in step sizes of π/20. Create a column heading called x (A1), then type 0 in the cell directly below the heading (A2).

We want the value to x to increase by π/20 in each subsequent row. This can be accomplished by clicking on A3 and typing **=A2+PI( )/20**.



SUM =A2+PI()/20

	A	B	C	D
1	x			
2	0			
3	=A2+PI()/20			
4				
5				
6				
7				

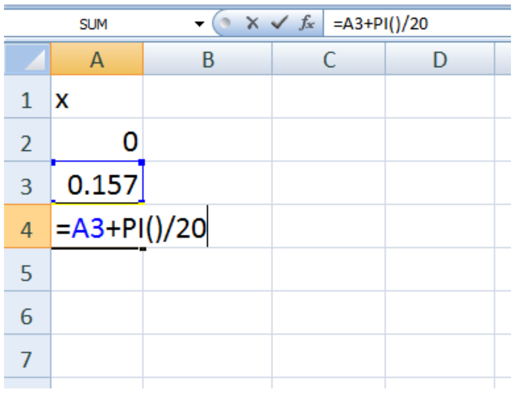


A3 =A2+PI()/20

	A	B	C	D
1	x			
2	0			
3	0.157			
4				
5				
6				
7				

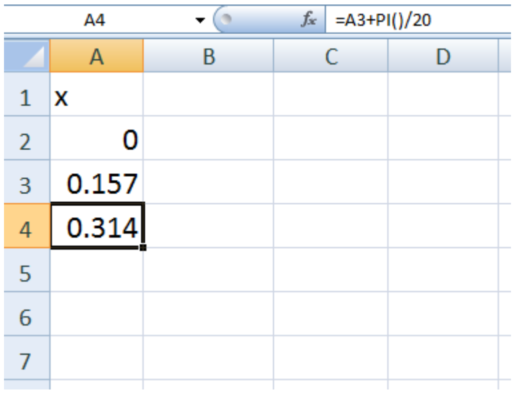
PI() is a built-in function that has no arguments (nothing is put in the parentheses) and returns the value of π, which is 3.141592653... and so on. Notice that the command **=A2+PI( )/20** uses relative references. The command is translated as "take the value in the cell in same column and one row above and add PI()/20 to it."

If we copy the formula in cell A3 and paste it in cell A4, we will obtain the value PI()/20 + PI()/20,



SUM =A3+PI()/20

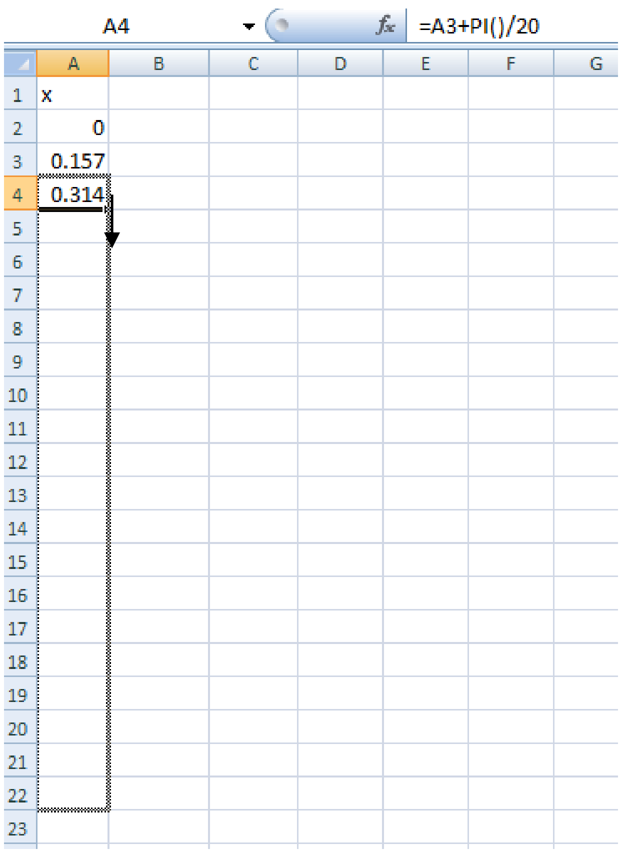
	A	B	C	D
1	x			
2	0			
3	0.157			
4	=A3+PI()/20			
5				
6				
7				



A4 =A3+PI()/20

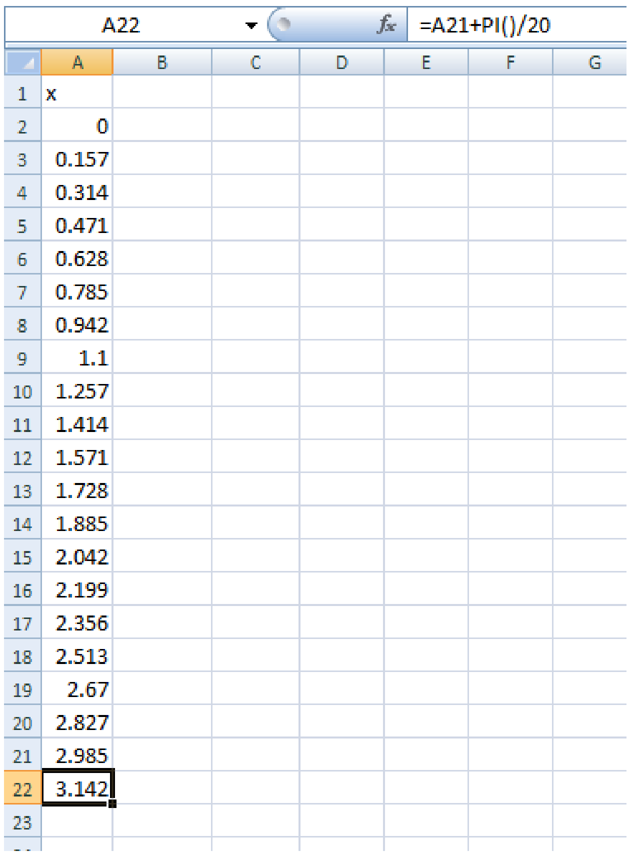
	A	B	C	D
1	x			
2	0			
3	0.157			
4	0.314			
5				
6				
7				

Use the fill handle to calculate the remaining values of x from 0 to π.



A4 =A3+PI()/20

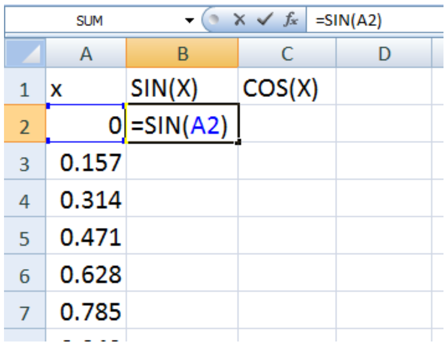
	A	B	C	D	E	F	G
1	x						
2	0						
3	0.157						
4	0.314						
5							
6							
7							
8							
9							
10							
11							
12							
13							
14							
15							
16							
17							
18							
19							
20							
21							
22							
23							



A22 =A21+PI()/20

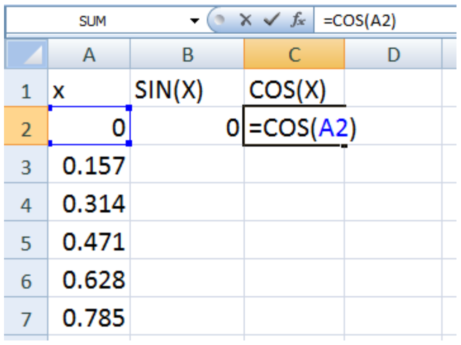
	A	B	C	D	E	F	G
1	x						
2	0						
3	0.157						
4	0.314						
5	0.471						
6	0.628						
7	0.785						
8	0.942						
9	1.1						
10	1.257						
11	1.414						
12	1.571						
13	1.728						
14	1.885						
15	2.042						
16	2.199						
17	2.356						
18	2.513						
19	2.67						
20	2.827						
21	2.985						
22	3.142						
23							

Now create two more column headings named SIN(X) and COS(X) in B1 and C1. Use the following formulas in the cells just below the headings (B2 and C2).



SUM =SIN(A2)

	A	B	C	D
1	x	SIN(X)	COS(X)	
2	0	=SIN(A2)		
3	0.157			
4	0.314			
5	0.471			
6	0.628			
7	0.785			



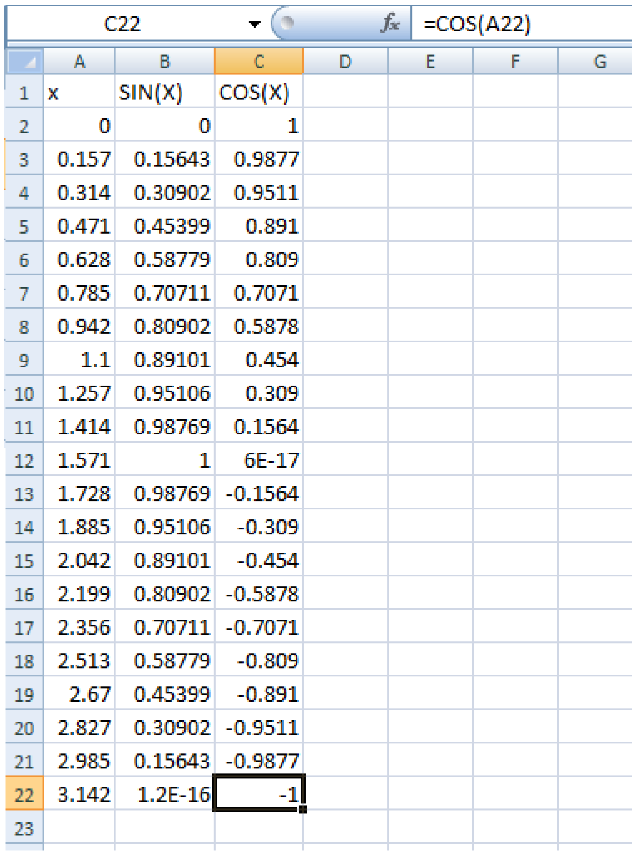
SUM =COS(A2)

	A	B	C	D
1	x	SIN(X)	COS(X)	
2	0	0	=COS(A2)	
3	0.157			
4	0.314			
5	0.471			
6	0.628			
7	0.785			

The formula in the cell B2 is translated as "take the value in the cell that is 1 column to the left and in the same row, and use it as the argument for the SIN() function."

The formula in the cell C2 is translated as "take the value in the cell that is 2 columns to the left and in the same row, and use it as the argument for the COS() function. "

Use the fill handle to calculate sine and cosine of all values of x.

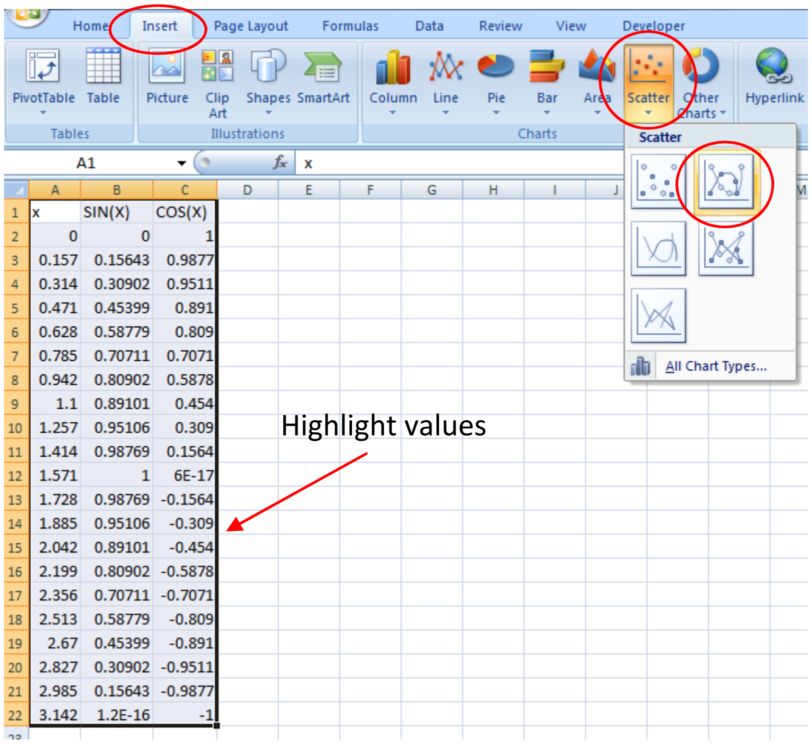


C22 =COS(A22)

	A	B	C
1	x	SIN(X)	COS(X)
2	0	0	1
3	0.157	0.15643	0.9877
4	0.314	0.30902	0.9511
5	0.471	0.45399	0.891
6	0.628	0.58779	0.809
7	0.785	0.70711	0.7071
8	0.942	0.80902	0.5878
9	1.1	0.89101	0.454
10	1.257	0.95106	0.309
11	1.414	0.98769	0.1564
12	1.571	1	6E-17
13	1.728	0.98769	-0.1564
14	1.885	0.95106	-0.309
15	2.042	0.89101	-0.454
16	2.199	0.80902	-0.5878
17	2.356	0.70711	-0.7071
18	2.513	0.58779	-0.809
19	2.67	0.45399	-0.891
20	2.827	0.30902	-0.9511
21	2.985	0.15643	-0.9877
22	3.142	1.2E-16	-1

Notice how the value of SIN(A22) and COS(A12) are almost zero. This is because π, like most numbers with decimals, cannot be stored to an infinite level of precision. But this concept is beyond the scope of the course. 1.2E-16 and 6E-17 are close enough to zero for most applications anyways.

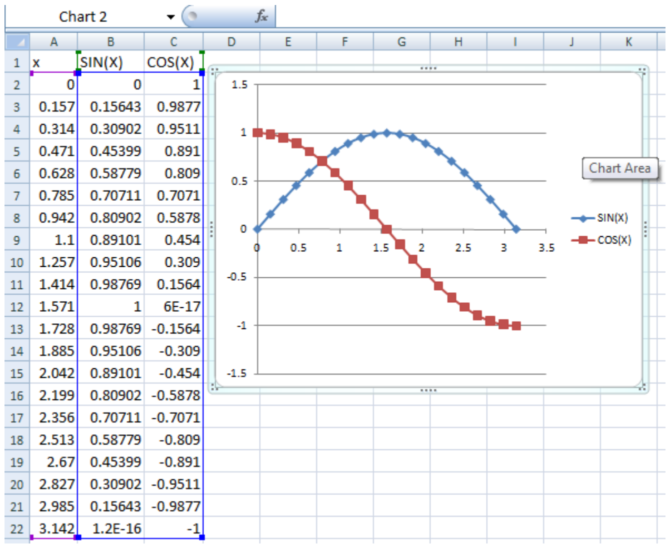
Let's plot SIN(x) and COS(x) versus x. Highlight the values you want to plot, click on the Insert tab, click on Scatter, then click on the type of scatter plot you want create.



A1 x

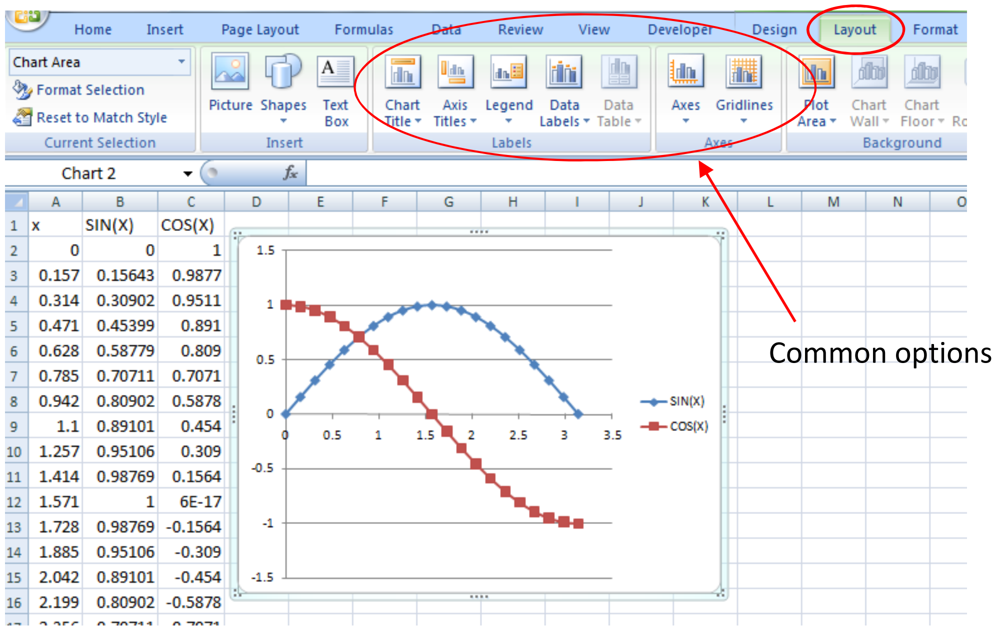
	A	B	C
1	x	SIN(X)	COS(X)
2	0	0	1
3	0.157	0.15643	0.9877
4	0.314	0.30902	0.9511
5	0.471	0.45399	0.891
6	0.628	0.58779	0.809
7	0.785	0.70711	0.7071
8	0.942	0.80902	0.5878
9	1.1	0.89101	0.454
10	1.257	0.95106	0.309
11	1.414	0.98769	0.1564
12	1.571	1	6E-17
13	1.728	0.98769	-0.1564
14	1.885	0.95106	-0.309
15	2.042	0.89101	-0.454
16	2.199	0.80902	-0.5878
17	2.356	0.70711	-0.7071
18	2.513	0.58779	-0.809
19	2.67	0.45399	-0.891
20	2.827	0.30902	-0.9511
21	2.985	0.15643	-0.9877
22	3.142	1.2E-16	-1

A chart should appear with a legend.



	A	B	C
1	x	SIN(X)	COS(X)
2	0	0	1
3	0.157	0.15643	0.9877
4	0.314	0.30902	0.9511
5	0.471	0.45399	0.891
6	0.628	0.58779	0.809
7	0.785	0.70711	0.7071
8	0.942	0.80902	0.5878
9	1.1	0.89101	0.454
10	1.257	0.95106	0.309
11	1.414	0.98769	0.1564
12	1.571	1	6E-17
13	1.728	0.98769	-0.1564
14	1.885	0.95106	-0.309
15	2.042	0.89101	-0.454
16	2.199	0.80902	-0.5878
17	2.356	0.70711	-0.7071
18	2.513	0.58779	-0.809
19	2.67	0.45399	-0.891
20	2.827	0.30902	-0.9511
21	2.985	0.15643	-0.9877
22	3.142	1.2E-16	-1

You can adjust the minimum and maximum values of the x- and y-axes, add a title, add labels to the x and y-axes, etc... by selecting the Layout tab and choosing various options.



	A	B	C
1	x	SIN(X)	COS(X)
2	0	0	1
3	0.157	0.15643	0.9877
4	0.314	0.30902	0.9511
5	0.471	0.45399	0.891
6	0.628	0.58779	0.809
7	0.785	0.70711	0.7071
8	0.942	0.80902	0.5878
9	1.1	0.89101	0.454
10	1.257	0.95106	0.309
11	1.414	0.98769	0.1564
12	1.571	1	6E-17
13	1.728	0.98769	-0.1564
14	1.885	0.95106	-0.309
15	2.042	0.89101	-0.454
16	2.199	0.80902	-0.5878

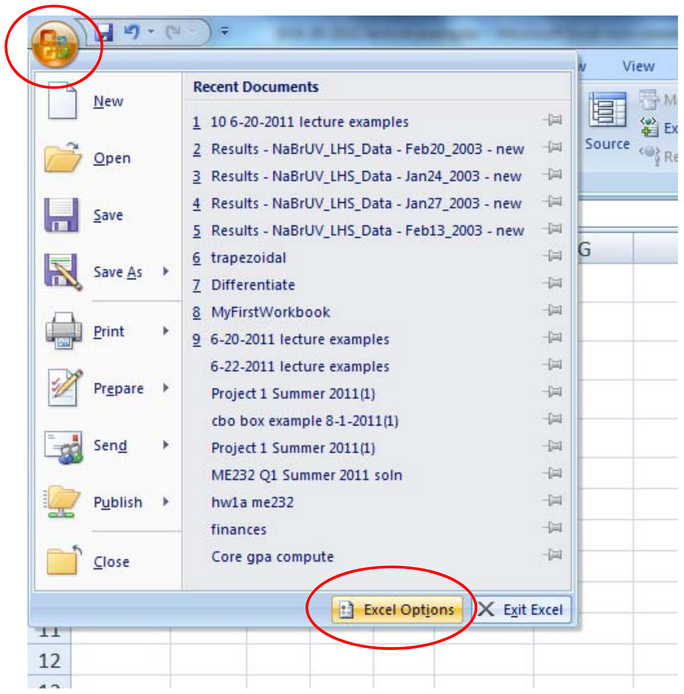
Topic 2: Introduction to the Visual Basic for Applications (VBA) environment

Excel contains two environments that are able to interact with each other:
(1) The familiar workbook environment that consists of cells, worksheets, and charts
(2) The Visual Basic for Applications environment that allows you to write your own programs (macros) and create graphical user interfaces.

We will be creating programs called macros using the VBA language. There are a few tasks that we need to complete before we start learning how to program. The instructions below are for Excel 2007, but the steps are similar in 2010 and 2013.

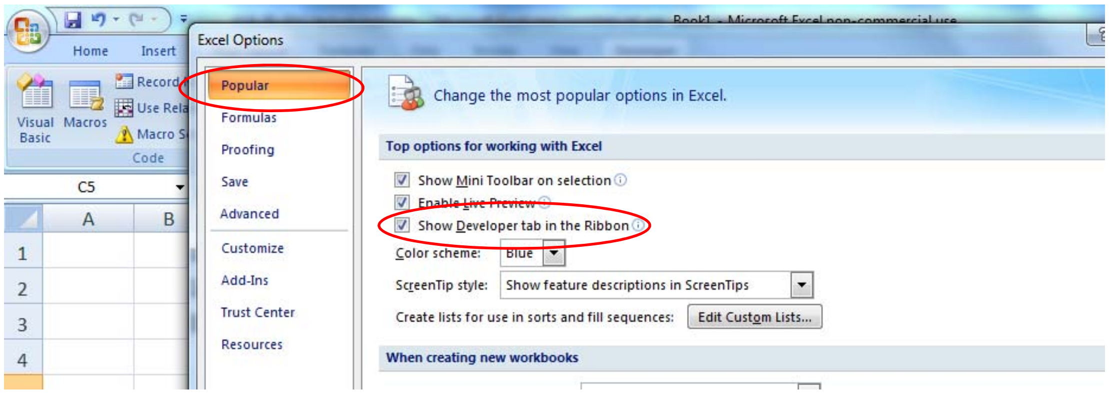
Showing the Developer Tab

It will be very convenient in this course to include the Developer tab at the top of the screen because you will need to access it many times. You can include this ribbon by clicking the **Office** button in the upper left corner of the Excel screen, then selecting **Excel Options**.



New
Open
Save
Save As
Print
Prepare
Send
Publish
Close
Recent Documents
1 10 6-20-2011 lecture examples
2 Results - NaBrUV_LHS_Data - Feb20_2003 - new
3 Results - NaBrUV_LHS_Data - Jan24_2003 - new
4 Results - NaBrUV_LHS_Data - Jan27_2003 - new
5 Results - NaBrUV_LHS_Data - Feb13_2003 - new
6 trapezoidal
7 Differentiate
8 MyFirstWorkbook
9 6-20-2011 lecture examples
6-22-2011 lecture examples
Project 1 Summer 2011(1)
cbo box example 8-1-2011(1)
Project 1 Summer 2011(1)
ME232 Q1 Summer 2011 soln
hw1a me232
finances
Core gpa compute
Excel Options
Exit Excel
View
Source
G
12

Under the **Popular** tab, make sure that **Show Developer tab in the Ribbon** option is selected.

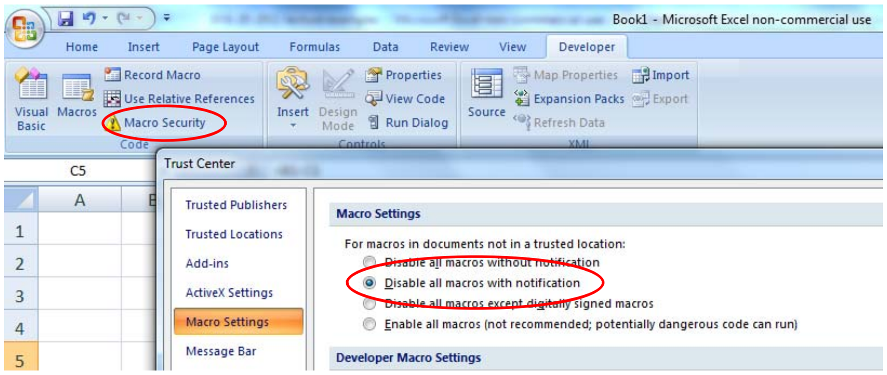


[✓] Show Mini Toolbar on selection
[✓] Enable Live Preview
[✓] Show Developer tab in the Ribbon
Color scheme: Blue
ScreenTip style: Show feature descriptions in ScreenTips
Create lists for use in sorts and fill sequences:

Setting the security level

You may need to lower the default security level of Excel. This is because you will be creating and running macros. Some bad people out there have developed malicious macros which can do bad things to your computer. These macros can be included in Excel files and other Microsoft Office files. Although the macros you create in this class will be harmless, Excel and other programs often prevent macros from starting when you open an Excel file. You can override this by doing the following:

Under the **Developer** tab, click **Macro Security** and select **Disable all macros with notification**. Now you will have the option of enabling macros when you open an Excel file.

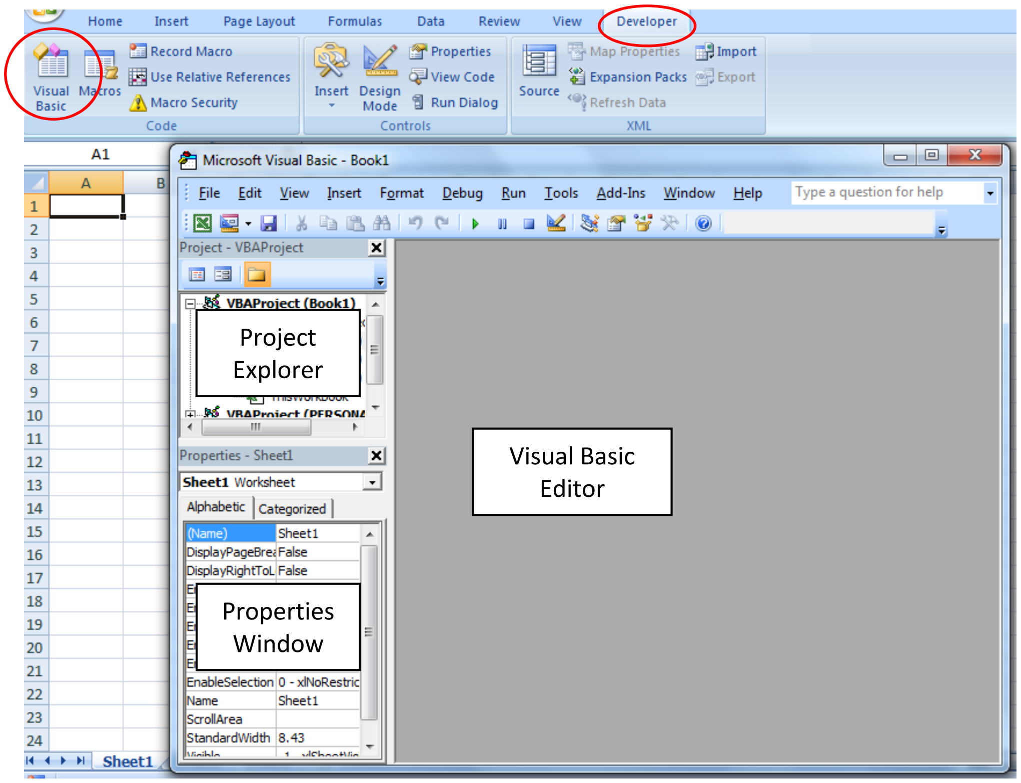


For macros in documents not in a trusted location:
- ○ Disable all macros without notification
- ● Disable all macros with notification
- ○ Disable all macros except digitally signed macros
- ○ Enable all macros (not recommended; potentially dangerous code can run)

If you create macros, you must save your excel file with the **.xlsm** extension, not the usual .xls or .xlsx extensions. When saving an Excel file with Macros, click the Office button > Save As > **Excel Macro-Enabled Workbook**.

Examining the VBA Environment

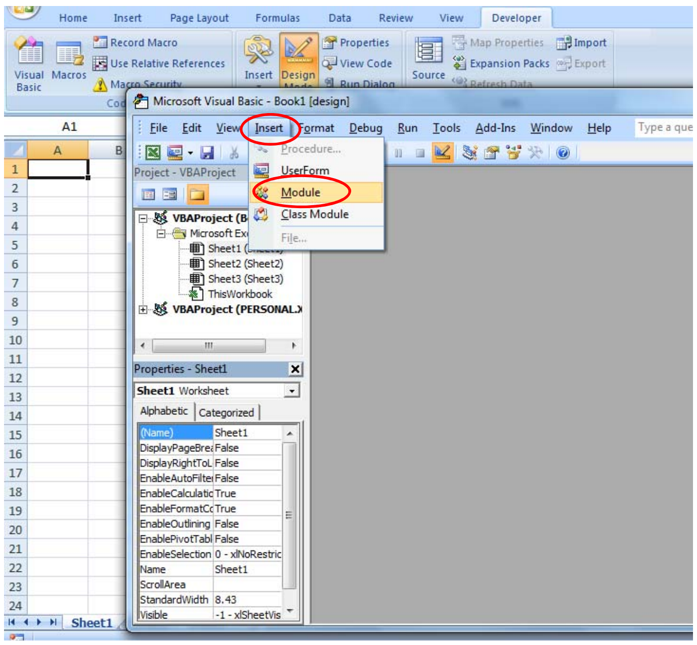
You can access the VBA Environment in two ways:
(1) Selecting the **Developer** tab, then clicking **Visual Basic**
(2) Typing **Alt+F11**.

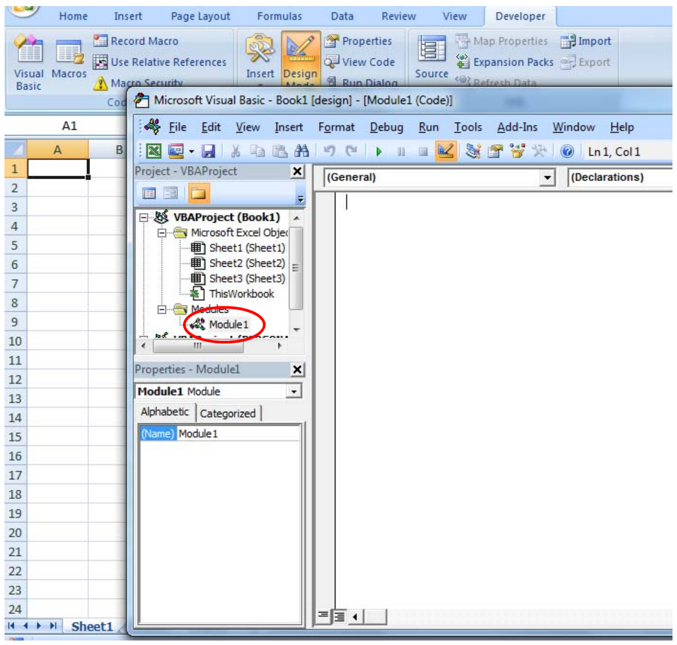


We now have access to the VBA Environment, which consists of three main sections:
- The Project Explorer which contains the components of your project
- The Properties Window which contains the properties of the components
- Visual Basic Editor which contains the programs you write

VBA organizes the code you write in the following manner:
- The programs you create are called **macros** (sometimes called **procedures**). A macro is written in as human-readable text that is **compiled** into a set of instructions the computer can understand when you execute the code.
- Macros are contained within **modules**. Initially, the editor is blank because no modules have been created.
- Modules are grouped together in a **project**. When a new workbook file is created in Microsoft Excel, a new VBA project is automatically created and associated with that workbook. Each workbook may contain only one project.

Let's create our first macro. First, we need to add a module. In the VBA Environment, click **Insert** > **Module**. You can think of a module as a container that holds one or more macros. A white text editor should open up. Our code goes into this text editor.



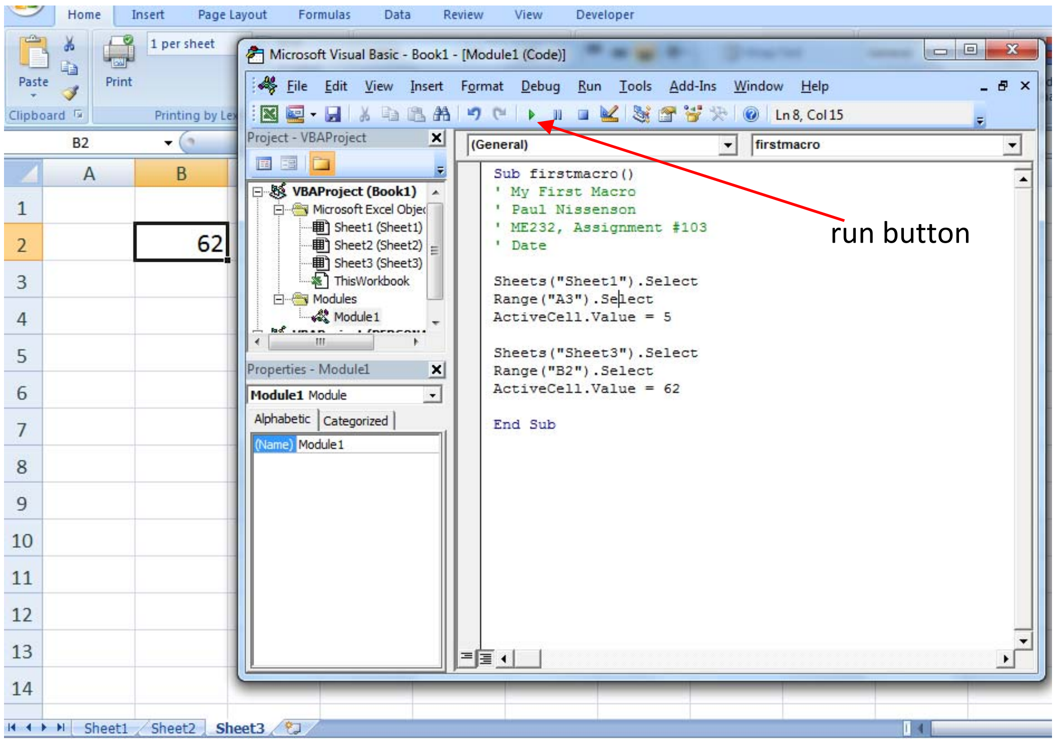


Alternatively, you could right-click in the Project Explorer window and select **Insert Module**.

Example: Using macros to put data into the worksheets

Our first macro will store a value in cell A3 in Sheet1 and a value in cell B2 in Sheet3. Type the following code into the text editor and click the run button (the green triangle). Make sure that both Sheets exist or you will get an error message when running the macro. Alternatively, you could run the macro by typing F5.

```
Sub firstmacro()
' My First Macro
' Paul Nissenson

Sheets("Sheet1").Select
Range("A3").Select
ActiveCell.Value = 5

Sheets("Sheet3").Select
Range("B2").Select
ActiveCell.Value = 62

End Sub
```



```
Sub firstmacro()
' My First Macro
' Paul Nissenson
' ME232, Assignment #103
' Date

Sheets("Sheet1").Select
Range("A3").Select
ActiveCell.Value = 5

Sheets("Sheet3").Select
Range("B2").Select
ActiveCell.Value = 62

End Sub
```

The number 5 should now be stored in cell A3 of Sheet1 and the number 62 should be stored in cell B2 of Sheet3. Let's analyze the code.

The macro program begins with,

Sub firstmacro()

The word **Sub** is short for subroutine "procedure" (another name for a "macro"). Sub procedures are different from Function procedures, which we will be learning about in Topics 4 and 5. The name of the macro is **firstmacro()**. If the program contains arguments (inputs), they would go in the parentheses. This Sub procedure contains no arguments.

' My First Macro
' Paul Nissenson

Comments are created by using an apostrophe and are automatically made green by the text editor. Everything after the apostrophe is ignored by the program. It is good programming practice to put commentary in all programs to help others who may use your code in the future. Additionally, you may wish to leave reminders to yourself describing what different sections of code do in case you need to revisit programs long after you write them.

Sheets("Sheet1").Select
Range("A3").Select
ActiveCell.Value = 5

We need to tell the program where in the workbook we want to place values. First we select the worksheet name, then the cell. The first two lines of code tells VBA that you are interested in the worksheet named

"Sheet1", and the cell "A3" within Sheet1. Cell A3 is now the **active cell**. The third line **assigns** the value of 5 to the active cell.

Note: The equal sign means the value on the right side is **assigned** to the item on the left side. It does not mean "equals." This distinction will become important in Topic 6.

```
Sheets("Sheet3").Select
Range("B2").Select
ActiveCell.Value = 62
```

Similarly, the first two lines of code tells VBA you are interested in worksheet "Sheet3", cell B2. The third line assigns the value of 62 to the active cell.

```
End Sub
```

The subroutine procedure (macro) is terminated with the **End Sub** command.

Variables

Most computer programs require the use of variables to store data temporarily. Variables can be thought of as containers for numbers or text. They are created when you assign a value to them. Here is a simple example that uses variables.

```
Sub main()
a = 5
b = 2 * a + 4
a = a + 1
b = 2 * a + 4
c = "hi all!"
d = c & "Bye!"
End Sub
```

In the first statement, the variable **a** is created and the value of 5 is assigned to it. If we use the variable **a** later in the code, the value 5 will be put in its place.

In the second statement, the quantity on the right is calculated first, then that value is assigned to the variable b. Just as in the Excel Workbook Environment, the order of operations must be obeyed. **2 * a** is 10, and this is added to 4. The value 14 is assigned to **b**.

The third statement is a little tricky. It is nonsense algebraically, but it is very common in computer programming. Remember, that the right side is evaluated first, then the resulting value is assigned to the variable on the left. **a + 1** is 6, and this value is assigned to **a**. The variable **a** now has the value of 6, not 5. The previous value of **a** is overwritten and gone forever.

When **a** is used in the fourth statement, its value is 6. The value of **2 * a + 4** is 16, which is stored in b. The previous value stored in **b** (14) is overwritten and gone forever.

The variables **a** and **b** store numbers. Strings, which are groups of one or more text characters, can be stored in variables too. In the fifth statement, the text **hi all!** is stored in the variable **c**. The text you wish to store in a string variable must be within quotation marks.

You can stitch strings together using the **concatenate (&) operator**. In the sixth statement, the characters **Bye!** are stitched to the end of the string variable **c**, and the resulting string (**hi all!Bye!**) is stored in **d**.

Here are some rules for naming variables:

- The name must begin with a letter.
- The name must contain only letters, numbers, and underscores (no other symbols).
- The name cannot be more than 255 characters long.
- The name cannot be a reserved word like *Print* or *Save*.

Now let's use some variables in another example.

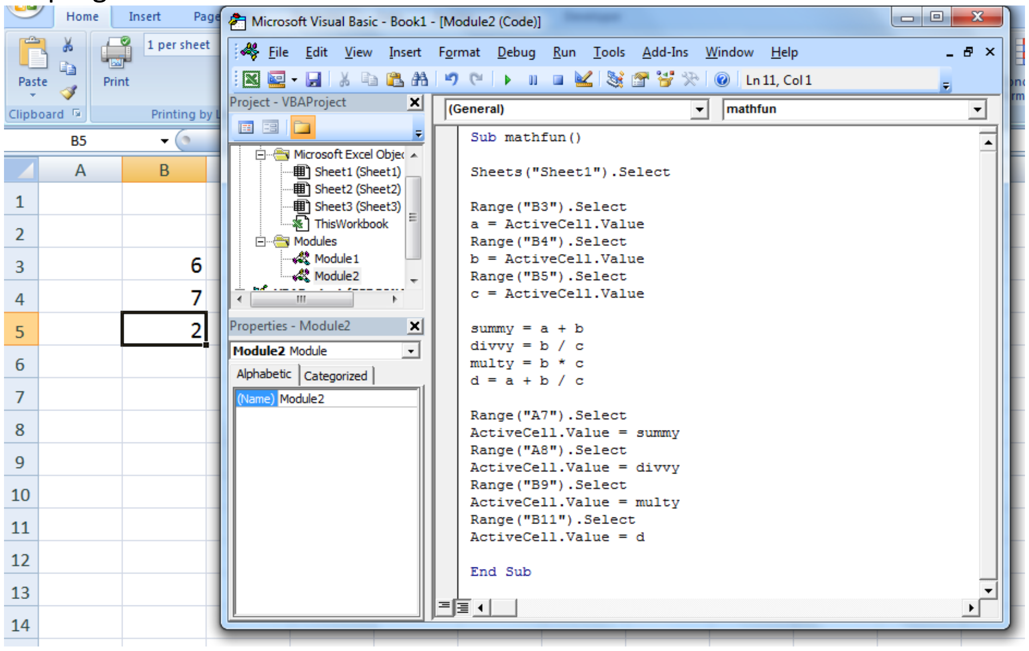
Example: Using macros to obtain data from worksheets

The previous example showed how macros can write data to the Workbook Environment. This next example shows how macros can obtain data from the Workbook Environment to perform calculations.

In worksheet Sheet1, type 6 in cell B3, 7 in cell B4, and 2 in cell B5. Then create a new module and write the following code.

```
Sub mathfun( )

Sheets("Sheet1").Select

Range("B3").Select
a = ActiveCell.Value
Range("B4").Select
b = ActiveCell.Value
Range("B5").Select
c = ActiveCell.Value

summy = a + b
d = a + b / c

Range("A7").Select
ActiveCell.Value = summy
Range("A8").Select
ActiveCell.Value = d

End Sub
```

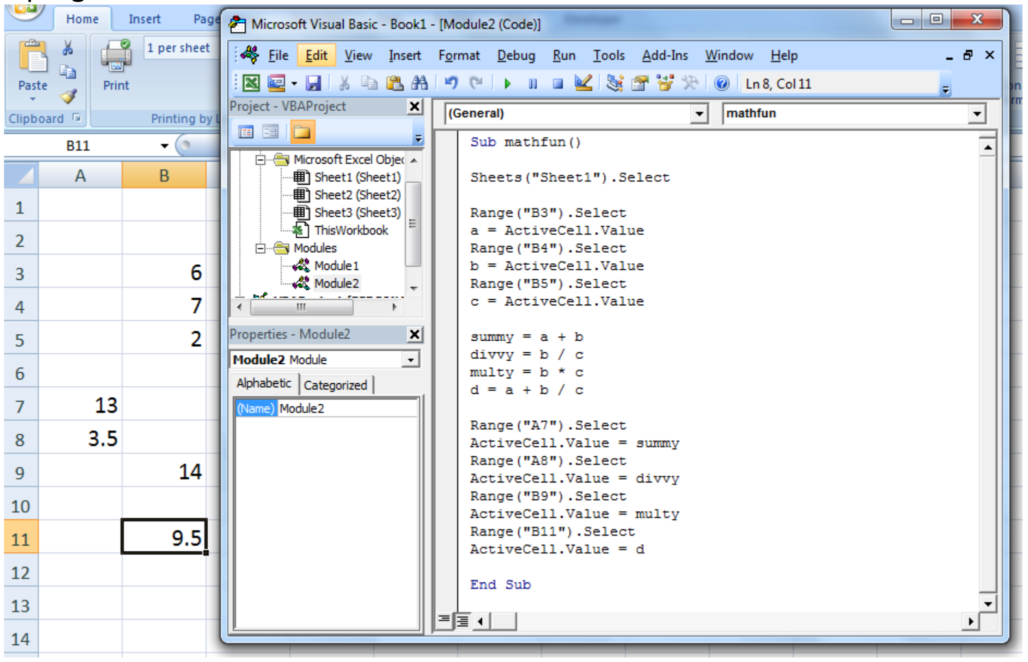
Before running the program:



	A	B
1		
2		
3		6
4		7
5		2
6		
7		
8		
9		
10		
11		
12		
13		
14		

```
Sub mathfun()

Sheets("Sheet1").Select

Range("B3").Select
a = ActiveCell.Value
Range("B4").Select
b = ActiveCell.Value
Range("B5").Select
c = ActiveCell.Value

summy = a + b
divvy = b / c
multy = b * c
d = a + b / c

Range("A7").Select
ActiveCell.Value = summy
Range("A8").Select
ActiveCell.Value = divvy
Range("B9").Select
ActiveCell.Value = multy
Range("B11").Select
ActiveCell.Value = d

End Sub
```

After running the program:



	A	B
1		
2		
3		6
4		7
5		2
6		
7	13	
8	3.5	
9		14
10		
11		9.5
12		
13		
14		

```
Sub mathfun()

Sheets("Sheet1").Select

Range("B3").Select
a = ActiveCell.Value
Range("B4").Select
b = ActiveCell.Value
Range("B5").Select
c = ActiveCell.Value

summy = a + b
divvy = b / c
multy = b * c
d = a + b / c

Range("A7").Select
ActiveCell.Value = summy
Range("A8").Select
ActiveCell.Value = divvy
Range("B9").Select
ActiveCell.Value = multy
Range("B11").Select
ActiveCell.Value = d

End Sub
```

Let’s analyze the code.

Sheets("Sheet1").Select
Range("B3").Select

The worksheet "Sheet1" is selected, then cell B3 is selected. This is now the active cell.

a = ActiveCell.Value

The value of the active cell (6) is assigned to the variable **a**. If we use **a** in the rest of the code, the value of 6 will put in its place.

Similarly, the value in cell B4 (7) is stored variable **b** and the value in cell B5 (2) is stored in variable **c**.

```
Range("B4").Select
b = ActiveCell.Value
Range("B5").Select
c = ActiveCell.Value
```

Notice we did not need to reselect the worksheet. Once we select a sheet, VBA remains on that sheet until we select a new sheet. Now the values stored in variables **a**, **b**, and **c** are used to perform some calculations.

```
summy = a + b
```

Again, the equals sign does not mean "equals." It means the expression on the right side will be assigned to the variable on the left side. The quantity **a + b** is 13, which gets stored in **summy**.

```
divvy = b / c
multy = b * c
```

b / c is 3.5, which gets stored in the variable **divvy**.
b * c is 14, which get stored in the variable **multy**.

```
d = a + b / c
```

For the variable **d**, we must calculate **a + b / c** using the order of operations. **b / c** is calculated first, then this value is added to **a**. 6 + 3.5 is 9.5, which gets stored in **d**.

Now let's write the values of **summy**, **divvy**, **multy**, and **d** to Sheet1. We already have experience with this from the previous example.

```
Range("A7").Select
ActiveCell.Value = summy
```

Our program has already selected Sheet1 earlier in the code, so we do not need to reselect it. In cell A7, we place the value of **summy**.

```
Range("A8").Select
ActiveCell.Value = divvy
Range("B9").Select
ActiveCell.Value = multy
Range("B11").Select
ActiveCell.Value = d
```

Similarly, we place the value stored in the variable **divvy** in cell A8, the value stored in the variable **multy** in cell B9, and the value stored in variable **d** in cell B11. We terminate the Sub procedure with,

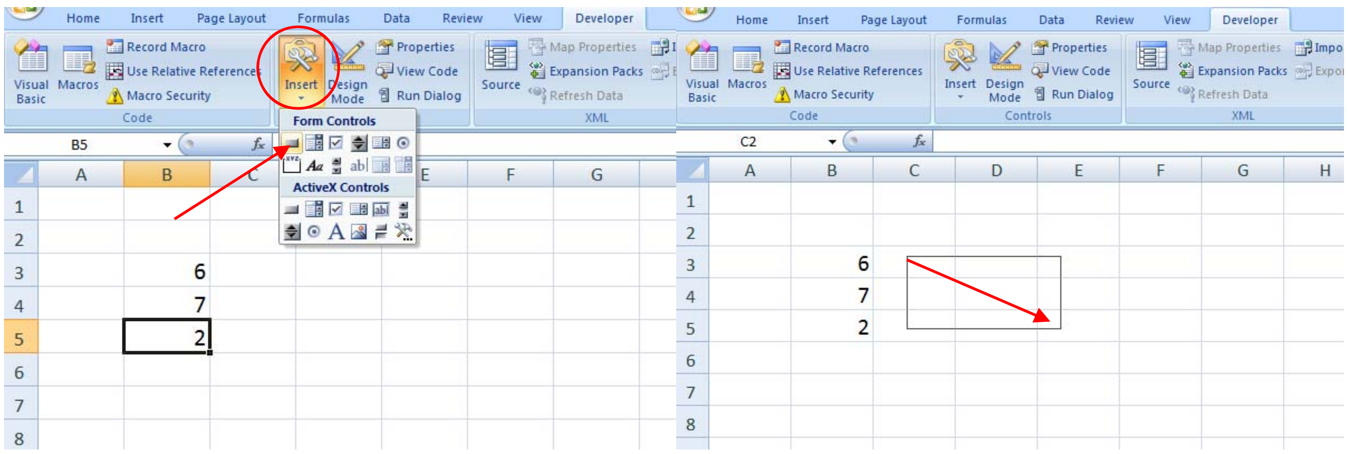
```
End Sub
```

Assigning buttons to macros: Making your programs more user-friendly

If your programs are difficult to access or understand, few people will want to use them. An important part of computer programming is making your programs as user-friendly as possible.

In the previous examples, we have run macros by clicking the run button in the VBA Environment of typing the F5 key. This requires us to open the VBA Environment, select the correct macro, then run the macro. Remember that most people do not know VBA, so it would be much nicer if a button could be created on the worksheet itself.

In the **Developer** tab in the workbook environment, click **Insert** then select **Button** under **Form Controls**. Click on the worksheet where you would like the button to be located. Drag the mouse until the button is the desired size.



An **Assign Macro** box should appear. Click on the macro you want assigned to the button. In this case, **mathfun** is the name of the macro. You can change the text on the button by right-clicking on the button and selecting **Edit Text**.

When you are finished, click on the button to run the **mathfun** macro.

Try changing the values in B3, B4, and B5 and clicking the button you just created. Now you don’t have to go into the VBA environment to run your programs.

Message Boxes: Another way to display output

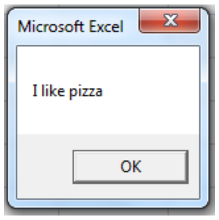
In some cases it might be more user-friendly to display answers using a pop-up message box instead of writing them to a cell in a worksheet. Message boxes are created using the **MsgBox** function. Text placed after **MsgBox** will be displayed on the screen and the program will halt until the user closes the message box.

Example: Using message boxes

Write the following macro:

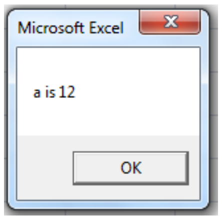
```
Sub main()
a = 12:  b = a + 1
MsgBox "I like pizza"
MsgBox "a is " & a
End Sub
```

When this macro is executed, first the variables **a** and **b** are assigned the values 12 and 13, respectively. The first Message Box then appears.



```
I like pizza
```

The program waits until you close the message box by clicking the OK button or clicking the close button (the X in the top right corner of the message box). Once you do complete this task, the second **MsgBox** function is executed.

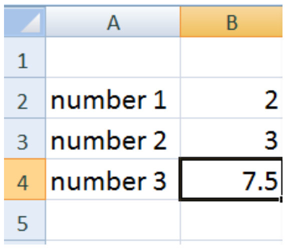


```
a is 12
```

Note: MsgBox can be used with or without parentheses.
MsgBox "a is " & a   is equivalent to   **MsgBox ("a is " & a)**

Another Example: Using message boxes

In Sheet2, type 2 into B2, 3 into B3, and 7.5 into B4.



	A	B
1		
2	number 1	2
3	number 2	3
4	number 3	7.5
5		

Then write the following macro:

```
Sub inout( )
Sheets("Sheet2").Select
Range("B2").Select
n1 = ActiveCell.Value
Range("B3").Select
n2 = ActiveCell.Value
Range("B4").Select
n3 = ActiveCell.Value
a = n1 + n2
b = n3 - n1
c = n3 * n2
MsgBox "number1 + number2 = " & a
MsgBox "number3 + number1 = " & b
MsgBox n3 & "*" & n2 & "=" & c
Range("B7").Select:  ActiveCell.Value = a + b
End Sub
```

Let’s analyze the program.

```
Sheets("Sheet2").Select
Range("B2").Select
n1 = ActiveCell.Value
Range("B3").Select
n2 = ActiveCell.Value
Range("B4").Select
n3 = ActiveCell.Value
```

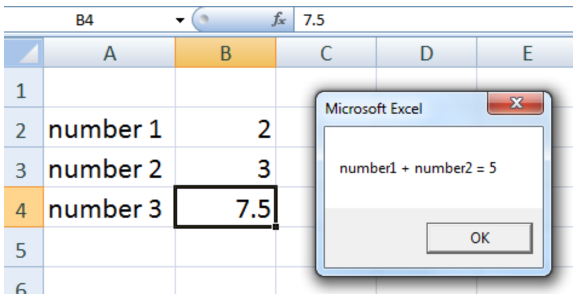
Sheet2 is selected. The value in B2 is assigned to the variable **n1**, the value in B3 is assigned to variable **n2**, and the value in B4 is assigned to the variable **n3**.

```
a = n1 + n2
b = n3 - n1
c = n3 * n2
```

The variables **a**, **b**, and **c** are calculated using the variables **n1**, **n2**, and **n3**.

```
MsgBox "number1 + number2 = " & a
```

When this line of code is executed, a message box appears.

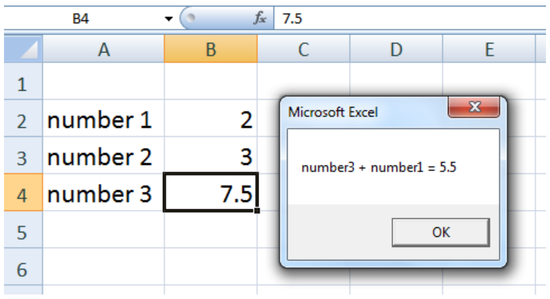


	A	B	C	D	E
1					
2	number 1	2			
3	number 2	3			
4	number 3	7.5			
5					

number1 + number2 = 5

The program halts until you click OK. Once this is done, the second **MsgBox** function is executed.

MsgBox "number3 + number1 = " & b



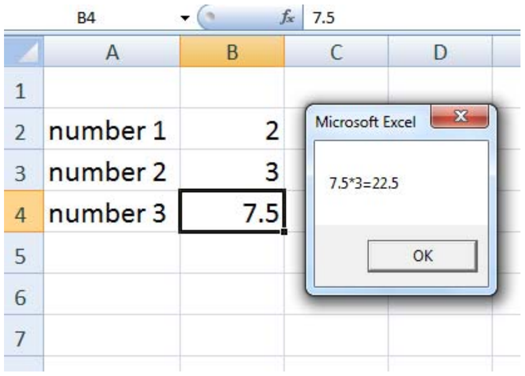
	A	B	C	D	E
1					
2	number 1	2			
3	number 2	3			
4	number 3	7.5			
5					
6					

number3 + number1 = 5.5

Something funny happened here. The value stored in the variable **b** is **n3 - n1**. But I accidentally typed **"number3 + number1"** in the MsgBox function instead of **"number3 - number1"**. Since there are no syntax errors in my program, no error messages appear. However, the program is not doing what I want it to do. Errors like this are very common in computer programming and are often difficult to catch.

Once you close the Message Box, the final MsgBox function will be executed.

MsgBox n3 & "*" & n2 & "=" & c

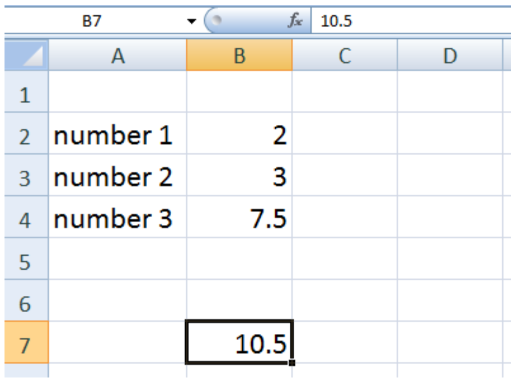


	A	B	C	D
1				
2	number 1	2		
3	number 2	3		
4	number 3	7.5		
5				
6				
7				

7.5*3=22.5

Once the last message box is closed, the final command can be executed which outputs the value of **a + b** to cell B2.

Range("B7").Select: ActiveCell.Value = a + b



B7 = 10.5

	A	B	C	D
1				
2	number 1	2		
3	number 2	3		
4	number 3	7.5		
5				
6				
7		10.5		

Input Boxes: Another way to obtain input from users

Sometimes it is more user-friendly to ask the user for input interactively using an input box. An input box can display a message to the user asking for input and obtain one piece of data from the user.

The general syntax for the InputBox function is the following:

```
variable = InputBox("message you want to send to user")          ' Use this for string input
```

The InputBox function will output a string (text) even if the user inputs a number. Without discussing too many technical details, I just want you to be aware that computers store the number 5 in a different manner from the character "5" so we need to be careful about how we handle input from the user. If you want the user to input a number, it is safer to put the InputBox function within the Val function.

```
variable = Val(InputBox("message you want to send to user"))    ' Use this for numerical input
```

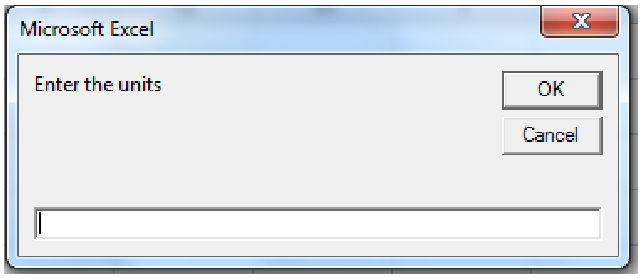
Note: VBA is very user-friendly and will allow you to omit Val in many situations without any negative consequences – it will know that you meant to store the number 5 instead of a character "5". But it is recommended to use Val when you want the user to input numerical data.

Example: Using input boxes

Write the following macro and execute it:

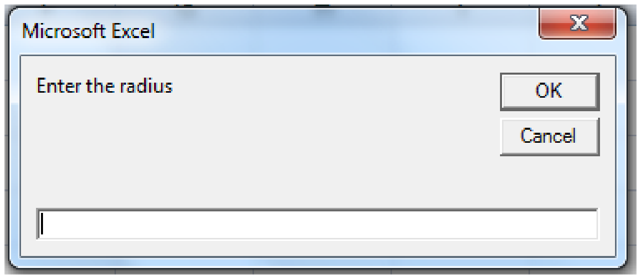
```
Sub displayRad( )
units = InputBox("Enter the units")              ' Don’t use Val when the user should enter text
rad = Val(InputBox("Enter the radius length"))   ' Use Val when the user should enter a number
MsgBox "The radius is " & rad & units
End Sub
```

When the first InputBox function is executed, an input box appears. The program is halted until the user enters a value for the units.



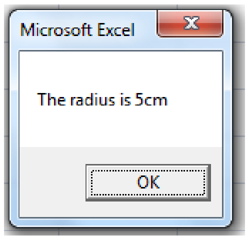
Enter the units

In this example, I typed **cm** into the text field and clicked OK. The string **cm** is now stored in the variable **units**. Next, the second InputBox function is executed.



Enter the radius

In this example, I typed **5** into the text field and clicked OK. This number is stored in the variable **rad**. Finally, we display a message to the user using a message box.



The radius is 5cm

Absolute references versus relative references in the VBA environment

In the previous examples, we have been using absolute reference when selecting cells. Here is an example of using absolute references:

```
Sub info( )
Sheets(“Sheet1”).Select
Range("A1").Select
ActiveCell.Value = "Paul Nissenson"
Range("A2").Select
ActiveCell.Value = "ID #"
Range("B3").Select
ActiveCell.Value = "=Sin( PI( )/4 )"
Range("C5").Select
ActiveCell.Value = 2.134
End Sub
```

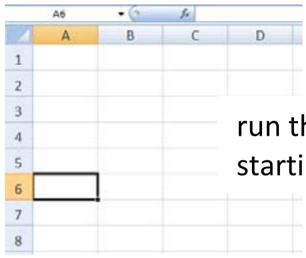
Cell A1 is selected, then the string **Paul Nissenson** is written to that cell.

Next, cell A2 is selected, then the string **ID #** is written to that cell.

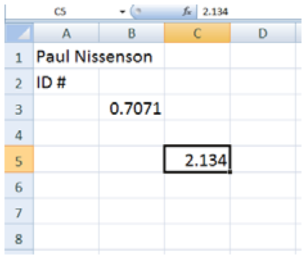
Next, cell B3 is selected, then the string **"=Sin(PI( )/4)"** is written to that cell. Note that this expression will be evaluated after it is entered into B3, just like if you typed the expression into B3 manually.

Finally, cell C5 is selected, then the number **2.134** is written to that cell.

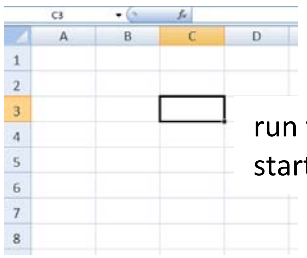
These four values are written to the exact same four cells regardless of where the cursor starts.



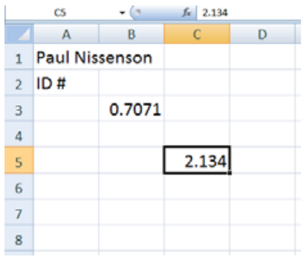
run the macro
starting in cell A6



	A	B	C	D
1	Paul Nissenson			
2	ID #			
3		0.7071		
4				
5			2.134	
6				
7				
8				



run the macro
starting in cell C3



	A	B	C	D
1	Paul Nissenson			
2	ID #			
3		0.7071		
4				
5			2.134	
6				
7				
8				

Often, we want to perform calculations and output information starting at the last cell clicked by the user, which is the current active cell. This is accomplished using **ActiveCell.Offset( ).Select**

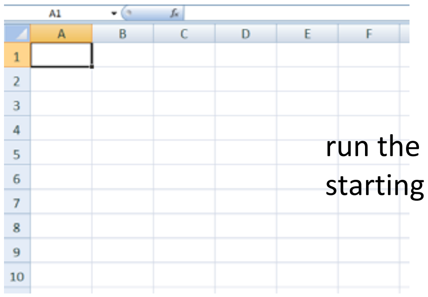
The **Offset( )** property has the general form of,

Offset(RowOffset,ColumnOffset)

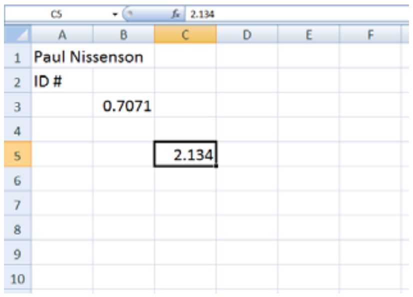
and shifts the active cell by **RowOffset** number of rows and **ColumnOffset** number of columns. For example, **ActiveCell.Offset(3 , -2).Select** shifts the active cell by +3 rows and -2 columns. If the active cell were initially at E10, it would be moved to C13 after that command were executed.

```
Sub info( )
ActiveCell.Value = "Paul Nissenson"
ActiveCell.Offset(1, 0).Select
ActiveCell.Value = "ID #"
ActiveCell.Offset(1, 1).Select
ActiveCell.Value = "=Sin( PI()/4 )"
ActiveCell.Offset(2, 1).Select
ActiveCell.Value = 2.134
End Sub
```

When you click on a cell on a worksheet, it becomes the active cell. For example, if you click on cell A1, A1 is the active cell.

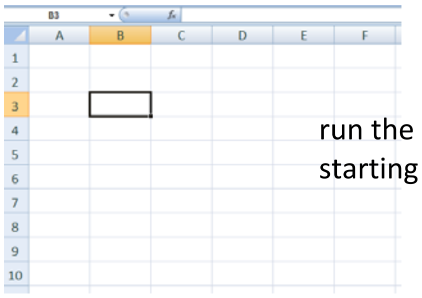


run the macro
starting in cell A1

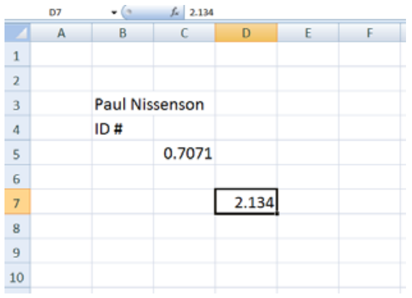

C5 2.134

	A	B	C
1	Paul Nissenson		
2	ID #		
3		0.7071	
5			2.134

If we run the macro starting in a different active cell, the four values are written to different cells. However, the distance between the cells is the same regardless of which cell is selected when the macro is run.



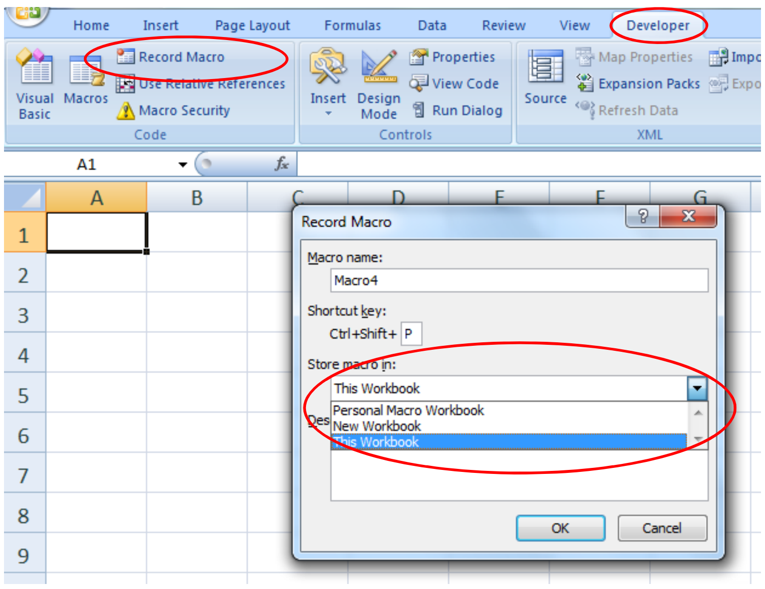
run the macro
starting in cell B3


D7 2.134

	B	C	D
3	Paul Nissenson		
4	ID #		
5		0.7071	
7			2.134

Recording macros

If you have a series of tasks that need to be completed many times, you can record the actions and assigning them to a shortcut key. In the **Developer** tab, click **Record Macro**. A pop-up window will allow you create a macro name, a shortcut key, and allow you to store your macro in either your personal macro workbook, the current workbook, or a new workbook. Choosing **Personal Macro Workbook** will allow you to use the macro in other workbooks too.



Record Macro

Macro name: Macro4

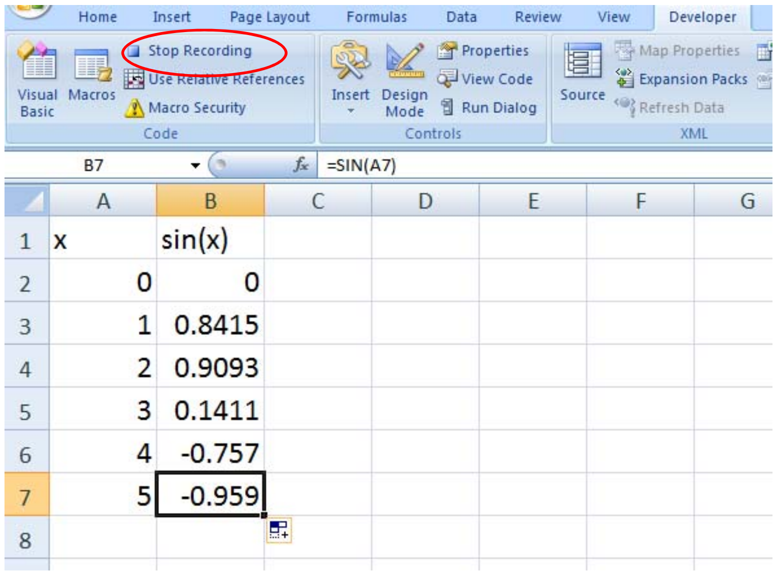
Shortcut key: Ctrl+Shift+ P

Store macro in: This Workbook

Personal Macro Workbook
New Workbook
This Workbook

In this example, I will call my macro **Macro4**, assign the shortcut key **Ctrl+Shift+P**, and store the macro in **This Workbook**. After I click on the OK button, all my actions will be recorded and converted into VBA code.

For this simple example, I will create a column of x values and the corresponding SIN(x) values. After I have completed the task, click the **Stop Recording** button.

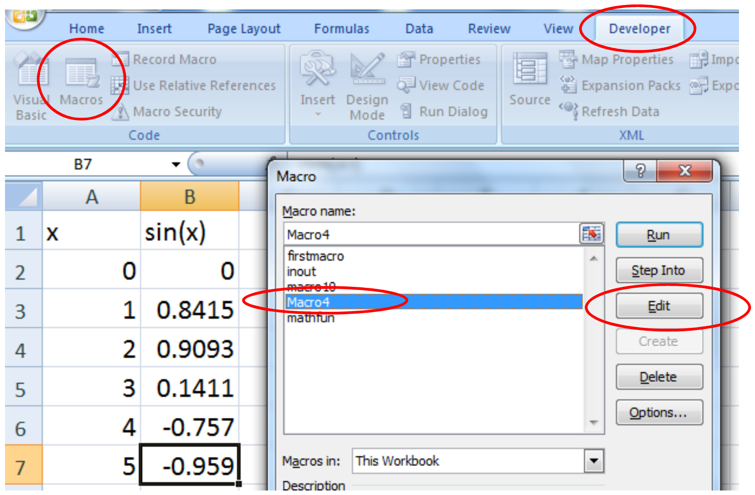


B7 =SIN(A7)

	A	B
1	x	sin(x)
2	0	0
3	1	0.8415
4	2	0.9093
5	3	0.1411
6	4	-0.757
7	5	-0.959
8		

You now have a macro that can be run by typing Ctrl+Shift+P on any worksheet in this workbook. This macro will create two columns of data containing the values of x and SIN(x). Try opening another worksheet and typing Ctrl+Shift+P.

If you would like to look at the VBA code created for this macro, click on the **Macros** button in the **Developer** tab. You should see the macro you just created. Click on the **Edit** button to examine the code.



B7

	A	B
1	x	sin(x)
2	0	0
3	1	0.8415
4	2	0.9093
5	3	0.1411
6	4	-0.757
7	5	-0.959

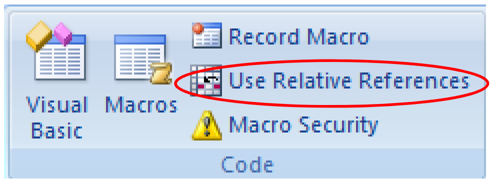
Macro name: Macro4

firstmacro
inout
macro10
Macro4
mathfun

Macros in: This Workbook

The VBA code that is automatically generated when recording macros often is more complicated than necessary. Even so, **recording macros can be a great way of learning how to code a certain task in VBA**. For example, if you want to know how to change the text color to red in a particular cell, record a macro while you perform that task and look the code that is generated.

By default, the recorded macros use absolute references. Every time you execute your macro, the actions will be performed on the exact same cells. If you want the macro actions to start acting on cells relative to a cell that you select, you can click on the **Use Relative References** button before recording your macro.



Dealing with space issues in VBA

Placing multiple statements on the same line:
Multiple statements can be placed on the same line of code if a colon (:) separates the statements. The following lines of code

```
        a = 34
        b = a + 3
        c = a * b
```

are equivalent to

```
        a = 34:  b = a + 3:  c = a * b
```

are equivalent to

```
        a = 34
        b = a + 3:  c = a * b
```

Statements on the same line are executed left to right. From now on, I will often use colons to save space in the notes.

Continuing on the next line:
If you have a long line of code, you can use an underscore to continue onto the next line. The following line of code

```
h = 10 + 20 / 2 - 6 ^ 2 * 2 + (2 + 5) ^ 2
```

is equivalent to

```
h = 10 + 20 / 2 - 6 ^ 2 * 2 _
    + (2 + 5) ^ 2
```

Note: There is a space between the 2 and underscore.

Topic 3: Data types and built-in functions in VBA

Data Types

Most interesting programs are hundreds or thousands of lines long. As your programs get longer, you increase the odds of making a silly error that could be undetected by VBA but lead to incorrect results. You can help prevent these errors by telling VBA which variables you want to exist.

Thus far, we have created variables simply by using them in an expression. For example, the statements,

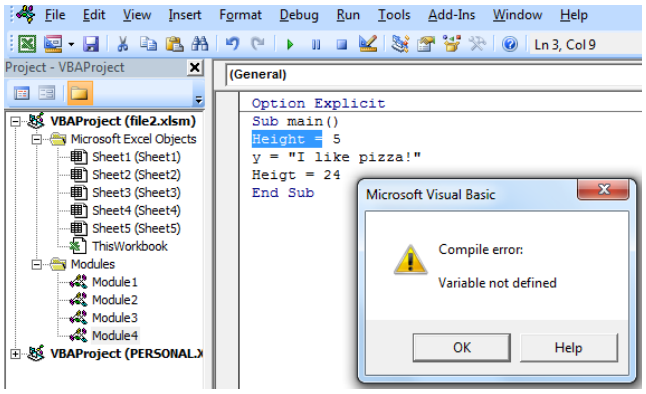
```
Height = 5
y = "I like pizza!"
```

create two variables named **Height** and **y**. The variable **Height** contains a number and the variable **y** contains a character string. Variables can contain many different types of data such as integers, numbers with decimal points, character strings, and Boolean values (true/false). When the two statements above are executed, **Height** is temporarily given an Integer data type and **y** is temporarily given a String data type automatically. By default, VBA will choose a data type for all variables you create, but this could lead to some huge problems. For example, what would happen if you wanted to change the value stored in **Height** to 24 later in the program, but accidentally typed the following statement?

```
Heigt = 24
```

VBA would create another Integer variable called **Heigt** and the variable **Height** would remain unchanged. The program would compile fine and you may never realize that you made an error in your code. We can avoid this problem by using the **Option Explicit** statement at the top of every module. Option Explicit forces you to **define** the data type of all variables manually. For example, try running the following program,

```
Option Explicit
Sub main( )
Height = 5
y = "I like pizza!"
Heigt = 24
End Sub
```



```
Option Explicit
Sub main()
Height = 5
y = "I like pizza!"
Heigt = 24
End Sub
```

Compile error:

Variable not defined

you will get an error message stating that the variable **Height** is not defined. This means that you have not yet defined (or **declared**) **Height** yet. It may seems like an unnecessary burden to declare each variable, but using

Option Explicit is considered good programming practice because it is very easy to make undetectable mistakes like in the previous example.

After using the Option Explicit statement, we must choose a data type for every variable. Here is a list of commonly used data types:

Category	Type	Memory required (bytes)	Min value	Max value
Integers	Byte	1	0	255
	Integer	2	-32,768	32,767
	Long	4	-2,147,483,648	2,147,483,647
Real	Single	4	approximately -3E38 to -1E-45	approximately 1E45 to 3E38
	Double	8	approximately -2E308 to -5E-324	approximately 5E-324 to 2E308
String	String (Fixed Length)	Variable	0 characters	~64E3 characters
	String (Variable Length)	Variable	0 characters	~2E9 characters
Boolean	Boolean	2	False (0)	True (1)

For this class, we will often be using **Integer/Long**, **Double**, **String**, and **Boolean**. Note that each data type has limits on the data that can be stored in it. For example, you cannot store integers that are larger than 32767 in a variable that has an Integer data type. These limitations are related to the amount of memory that is allocated for each data type… these concepts are beyond the scope of the class.

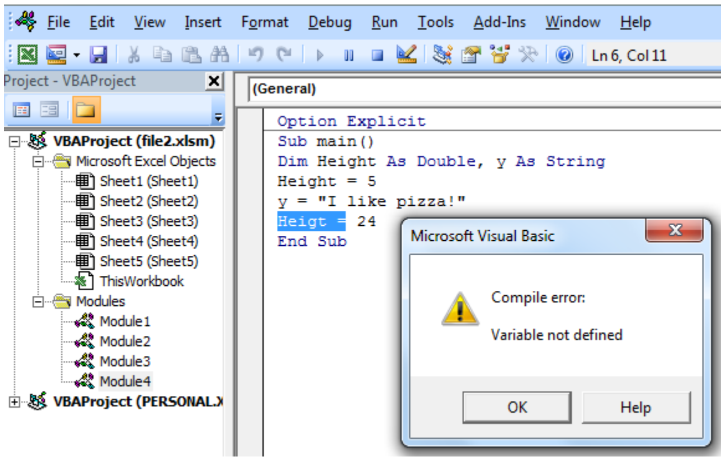
We declare variables by using the **Dim** statement,

```
Dim varname1 As type1, varname2 As type2, ...
```

Let's declare **Height** and **y** in our program,

```
Option Explicit
Sub main( )
Dim Height As Double, y As String             ' Added this line
Height = 5
y = "I like pizza!"
Heigt = 24
MsgBox Height
End Sub
```

If we run the program, we will receive an error message because **Heigt** was never declared.



```
Option Explicit
Sub main()
Dim Height As Double, y As String
Height = 5
y = "I like pizza!"
Heigt = 24
End Sub
```

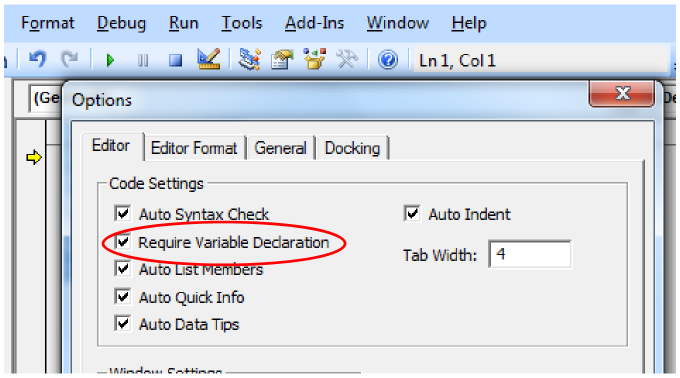
Microsoft Visual Basic

Compile error:

Variable not defined

Change **Heigt** to **Height** and rerun the code. The code should run without error.

We can make every module start with **Option Explicit** by clicking **Tools** > **Options**, and check the box that states **Require Variable Declaration**.



Code Settings

☑ Auto Syntax Check
☑ Require Variable Declaration
☑ Auto List Members
☑ Auto Quick Info
☑ Auto Data Tips
☑ Auto Indent
Tab Width: 4

Creating constants

Often it is desirable to create variables whose value never changes. For example, you may want to use the value of π, the gravitation acceleration, or the height of a building many times in a program. You can protect these constant variables from accidentally being overwritten using the **Const** statement when declaring variables. The general notation for **Const** is,

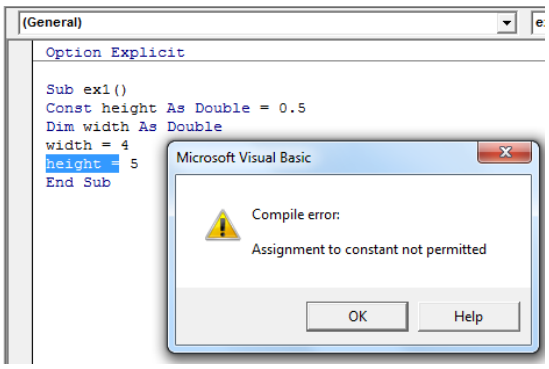
Const constantname As Type = value

Example: Creating constants

Write the following program and try to execute it.

```
Option Explicit
Sub ex1( )
Const height As Double = 0.5
Dim width As Double
width = 4
height = 5                              ' Not allowed
End Sub
```

An error message occurs stating that the constant **height** cannot be overwritten.



```
Option Explicit

Sub ex1()
Const height As Double = 0.5
Dim width As Double
width = 4
height = 5
End Sub
```

Compile error:

Assignment to constant not permitted

Built-in functions in VBA

There are many built-in functions in VBA. The names of the functions are often similar to the names of the functions used in the Excel Workbook Environment. For example,

Sqr(x) yields the square root of x
Sin(x) yields the sine of x
Log(x) yields the natural log of x
MsgBox creates a message box

To access functions that are used in the workbook environment, you can use

Application.WorksheetFunction.fname()

where **fname()** is the workbook function you want to access. Most workbook functions are accessible in VBA using this command.

Topic 4: Modular Programming using Sub procedures

Modular programming is a style of programming where a program is divided into subprograms that are each responsible for different tasks. There are many reasons for making programs modular:

- Your code will be more organized and it will be easier for others to understand your code. Textbook authors organize material into chapters for the same reason.
- Modular programming allows multiple people to work on the same program at the same time in parallel. Each person can develop a different subprogram which will be combined together in a larger program later on.
- It is easier to find mistakes and errors in a program that separates tasks into different subprograms.
- In many programs you will need to execute the exact same set of commands multiple times. With modular programming, you only need to write those set of commands once in a single subprogram. This can shorten your code significantly.
- If you write all your programs in a modular fashion, it will be easy to later reuse procedures in different programs.

You won't realize the importance of modular programming until your programs become longer. For now, just trust me that it is a critical part of programming.

We will discuss two types of subprograms, called **procedures**, in this course:
(1) **Sub** (short for subroutine) procedures
(2) **Function** procedures

A procedure is a series of statements that are grouped together to complete one or more tasks.

General syntax for a Sub procedure

```
Sub  subname ( arguments* )
statements*
Exit Sub*
statements*
End Sub
```

Note: The items with an asterisk are often used, but technically optional.

The **Exit Sub** statement terminates the Sub procedure when executed and any statements after the **Exit Sub** statement will not be executed – it is like those statements did not exist. **Exit Sub** is usually used in combination with an If structure (Topic 6) and never by itself.

In Topics 1-3, we created Sub procedures that did not contain arguments and did not use the **Exit Sub** statement.

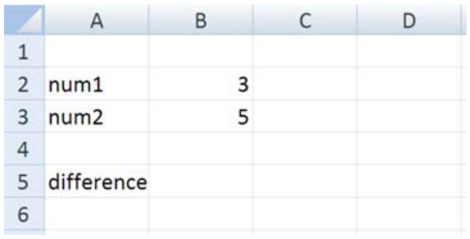
Sub procedures are invoked (or "called") using the **Call** statement.

```
Call subname(arguments)
```

If a Sub procedure is called in the middle of another program, that other program is halted until the Sub procedure is completed.

Example: Using a simple Sub procedure

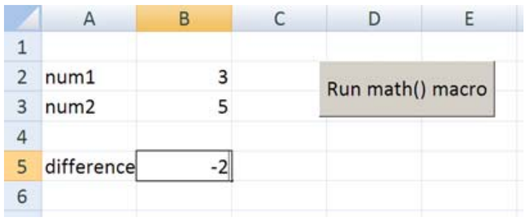
In the following example, there are numbers in cells B2 and B3. We will develop a macro that calculates the difference of these two numbers and displays the result in B5.



	A	B	C	D
1				
2	num1	3		
3	num2	5		
4				
5	difference			
6				

```
Option Explicit
Sub math( )
Dim x As Double, y As Double, ans As Double
Sheets("Sheet1").Select:
Range("B2").Select: x = ActiveCell.Value
Range("B3").Select: y = ActiveCell.Value
ans = x - y
Range("B5").Select: ActiveCell.Value = ans
End Sub
```

Create a button to run the macro by going to the Developer ribbon and clicking Insert > Button. Select cell B2 and run the macro.



	A	B	C	D	E
1					
2	num1	3			
3	num2	5			
4					
5	difference	-2			
6					

We will now change the **math()** Sub procedure so that the subtraction step will be handled by a separate Sub procedure. Of course, it is unnecessary to create a Sub procedure for this simple task, but this example is just designed to introduce you to how Sub procedures work.

```
Option Explicit
Sub math( )
Dim x As Double, y As Double, ans As Double
Sheets("Sheet1").Select:
Range("B2").Select: x = ActiveCell.Value
Range("B3").Select: y = ActiveCell.Value
Call diffy(x, y, ans)                             ' Changed line of code
Range("B5").Select: ActiveCell.Value = ans
End Sub

Sub diffy(x, y, ans)                              ' Sub procedure added to program
ans = x - y
End Sub
```

The **diffy( )** Sub procedure is written after the end of the **math( )** Sub procedure in the same module. Most of the **math( )** Sub procedure is unchanged.

```
Sub math( )
Dim x As Double, y As Double, ans As Double
Sheets("Sheet1").Select:
Range("B2").Select: x = ActiveCell.Value
Range("B3").Select: y = ActiveCell.Value
```

As before, the variables **x**, **y**, and **ans** are declared as Double variables. When a variable is given a numerical data type (Single, Double, Long, etc.), it is given a default value of 0. Then variables **x** and **y** are assigned new values. The **diffy()** Sub procedure is invoked (or "called") using the **Call** statement,

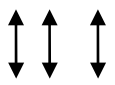
```
Call diffy(x, y, ans)
```

The **math()** Sub procedure is halted until **diffy()** is completed. The **diffy()** Sub procedure takes 3 arguments as input. Notice that only variables **x** and **y** have been given new values at the time **diffy()** is called. The variable **ans** will obtain a new value in **diffy()**.

```
Sub diffy(x, y, ans)
ans = x - y
End Sub
```

When **diffy( )** is called, the memory locations of the variables **x**, **y**, and **ans** in **math()** are passed to the variables **x**, **y**, and **ans** in **diffy()**. As a result:
The first argument in the calling statement is linked with the first argument in the **diffy()** argument list.
The second argument in the calling statement is linked with the second argument in the **diffy()** argument list
The last argument in the calling statement is linked with the last argument in the **diffy()** argument list.
It just so happens that the three arguments in both lists are the same, but they do not have to be.

Call diffy(x, y, ans)



Sub diffy(x, y, ans)

Notice how the variables **x**, **y**, and **ans** did not have to be declared in the **diffy()** Sub procedure. This is because they are associated with memory locations that have already been created in the **math()** Sub procedure. In fact, you would get an error message if you tried to declare **x**, **y**, and **ans** in **diffy()**.

Often it is useful to write down the current value of each variable in the program. Every time a variable changes its value, update the list. Here is the status of the arguments when **diffy()** is called.

In **math()**, arguments are named:	x	y	ans
	3	5	0
In **diffy()**, arguments are named:	x	y	ans

In **diffy()**, the value of **x-y** is assigned to **ans**. Then the Sub procedure **diffy()** is terminated. The new values of **x**, **y**, and **ans** are passed back up to **math()**.

Here is the status of the arguments when **diffy()** is terminated.

In **math()**, arguments are named:	x	y	ans
	3	5	-2
In **diffy()**, arguments are named:	x	y	ans

Back in **math()**, the value of **ans** is put into cell B5 of Sheet1.

```
Range("B5").Select: ActiveCell.Value = ans
```

You can use the **diffy()** Sub procedure in other modules by calling it in the same way as in the **math()** Sub procedure,

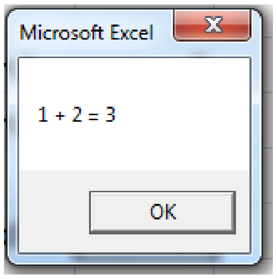
```
Call diffy(x, y, ans)
```

Example: Reusing Sub procedures

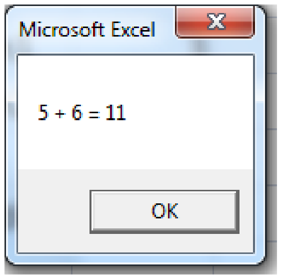
Often you will want to create Sub procedures that can be used many times with different argument names.

```
Option Explicit
Sub math2( )
Dim a As Double, b As Double, x As Double, y As Double
Dim ans As Double
a = 1:  b = 2:  x = 5:  y = 6
Call addy(a, b, ans)
MsgBox a & " + " & b & " = " & ans
Call addy(x, y, ans)
MsgBox x & " + " & y & " = " & ans
End Sub

Sub addy(n, m, result)
result = n + m
End Sub
```

Running this code will produce two message boxes.



```
1 + 2 = 3
```



```
5 + 6 = 11
```

This example introduces a couple new concepts. Let’s analyze the program.

```
Sub math2( )
Dim a As Double, b As Double, x As Double, y As Double
Dim ans As Double
a = 1:  b = 2:  x = 5:  y = 6
```

Variables **a**, **b**, **x**, **y**, and **ans** are created and given Double data types. Each variable is initially assigned a default value of 0, but **a**, **b**, **x**, and **y** are quickly assigned new values.

Here is the status of the variables in **math2()** before any Sub procedures are called.

math2()

a	b	x	y	ans
1	2	5	6	0

Next, **addy( )** is called and three arguments are passed down from **math2()**.

Call addy(a, b, ans)

Sub addy(n, m, result)

The memory location of variable **a** in **math2()** is now linked with the variable **n** in **addy()**.
The memory location of variable **b** in **math2()** is now linked with the variable **m** in **addy()**.
The memory location of variable **ans** in **math2()** is now linked with the variable **result** in **addy()**.

If the value of **n**, **m** or **result** changes in **addy()**, it will affect the value of **a**, **b**, and **ans** in **math2()**.

Here is the status of the arguments when **addy()** is first called.

In **math2()**, arguments are named:	a	b	ans
	1	2	0
In **addy()**, arguments are named:	n	m	result

In **addy()**, the value of **result** (3) is calculated and the Sub procedure is terminated. Here is the status of the arguments when **addy()** is terminated.

In **math2()**, arguments are named:	a	b	ans
	1	2	3
In **addy()**, arguments are named:	n	m	result

Notice how the value of **n** and **m** have not changed in **addy()**, so **a** and **b** retain the same value.

MsgBox a & " + " & b & " = " & ans

Back in **math2()**, the MsgBox function is invoked and a message box is displayed on the screen. Next, **addy()** is called for the second time.

Call addy(x, y, ans)

Sub addy(n, m, result)

The variable **x** in **math2()** is linked with the variable **n** in **addy()**.
The variable **y** in **math2()** is linked with the variable **m** in **addy()**.
The variable **ans** in **math2()** is linked with the variable **result** in **addy()**.

Here is the status of the arguments when **addy()** is called the second time.

In **math2()**, arguments are named:	x	y	ans
	5	6	3
In **addy()**, arguments are named:	n	m	result

In **addy()**, the value of **result** (11) is calculated and the Sub procedure is terminated. Here is the status of the arguments when **addy()** is terminated.

In **math2()**, arguments are named:

x	y	ans
5	6	11

Again, notice how the values of **x** and **y** do not change in **math2()** because **n** and **m** do not change in **addy()**.

```
MsgBox x & " + " & y & " = " & ans
```

The MsgBox function is invoked and a message box is displayed on the screen.

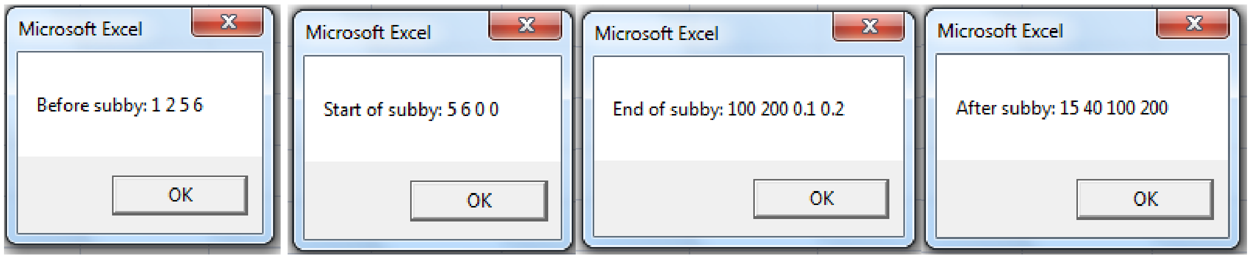
Variable scope and variable lifetime

It is easiest to illustrate these two concepts by going through an example, then discussing the results. Create the following program and execute it.

```
Option Explicit
Sub main()
Dim a As Double, b As Double, x As Double, y As Double
a = 1:  b = 2:  x = 5:  y = 6
MsgBox "Before subby: " & a & " " & b & " " & x & " " & y
Call subby(a, b, x, y)
MsgBox "After subby: " & a & " " & b & " " & x & " " & y
End Sub

Sub subby(n, m, a, b)
Dim x As Double, y As Double
MsgBox "Start of subby: " & a & " " & b & " " & x & " " & y
n = 15:  m = 40:  a = 100:  b = 200:  x = 0.1:  y = 0.2
MsgBox "End of subby: " & a & " " & b & " " & x & " " & y
End Sub
```

Note: This program does not do anything interesting. It is designed to illustrate the concepts of variable scope and variable lifetime.

When this program is run, the following Message Boxes appear.



```
Before subby: 1 2 5 6
Start of subby: 5 6 0 0
End of subby: 100 200 0.1 0.2
After subby: 15 40 100 200
```

Let's analyze the program.

```
Sub main( )
Dim a As Double, b As Double, x As Double, y As Double
a = 1:  b = 2:  x = 5:  y = 6
MsgBox "Before subby: " & a & " " & b & " " & x & " " & y
```

The variables **a**, **b**, **x**, and **y** are created and values are assigned to them. The values 1, 2, 5, and 6 are displayed in the message box.

Here is the status of the variables in **main()** before any Sub procedures are called.

Variables in **main()**:

a	b	x	y
1	2	5	6

Next, **subby()** is called.

```
Call subby(a, b, x, y)

Sub subby(n, m, a, b)
```

The variable **a** in **main()** is linked to variable **n** in **subby()**.
The variable **b** in **main()** is linked to variable **m** in **subby()**.
The variable **x** in **main()** is linked to variable **a** in **subby()**.
The variable **y** in **main()** is linked to variable **b** in **subby()**.

Here is the status of the arguments when **subby()** is called.

In **main()**, arguments are named:	a	b	x	y
	1	2	5	6
In **subby()**, arguments are named:	n	m	a	b

Now we enter **subby()**.

```
Dim x As Double, y As Double
MsgBox "Start of subby: " & a & " " & b & " " & x & " " & y
```

The values 5, 6, 0, and 0 appear in the message box. In the table above, **a** and **b** in **subby( )** are 5 and 6, but why are **x** and **y** both 0? Where **subby()** is defined, notice that **n**, **m**, **a**, and **b** are in the argument list, but **x** and **y** are not in the argument list. This means that the variables **x** and **y** are not associated with a pre-existing memory location and need to be declared using the **Dim** statement (if we did not declare them, we would get an error message). This creates locations in memory for **x** and **y**. The default value for any newly created numerical variable is 0.

```
n = 15:  m = 40:  a = 100:  b = 200:  x = 0.1:  y = 0.2
```

The values of **n**, **m**, **a**, **b**, **x**, and **y** are overwritten in **subby()**. Here is a table showing the current status of all the variables **subby()**.

Variables in **subby()**:

15	40	100	200	0.1	0.2
n	m	a	b	x	y

```
MsgBox "End of subby: " & a & " " & b & " " & x & " " & y
```

100 200 0.1 0.2 are displayed on in the message box. The subroutine is terminated and the values of the arguments in **subby()** are passed back to the arguments in **main()**.

Here is the status of the arguments when **subby( )** is terminated.

In **main()**, arguments are named:	a	b	x	y
	15	40	100	200
In **subby()**, arguments are named:	n	m	a	b

After the **subby()** is terminated, the variables **x** and **y** in **subby()** are terminated as well. Thus variables have a limited lifetime.

Back in main, we continue on from where **subby()** was called.

```
MsgBox "After subby: " & a & " " & b & " " & x & " " & y
```

15 40 100 200 are displayed in the message box.

This example shows that variables by default have **local scope**. Variables are affected only by commands within their own procedure. A variable called **x** in **main()** can be different from a variable called **x** in **subby()**. In this program, when **x** was changed in **subby()** it did not affect the variable **x** in **main()**.

It is possible to extend the scope of a variable to all procedures in a module or to an entire project by making global variables, but it is not recommended and I will not be showing your how to do this except when learning about UserForms (Topic 8). It is consider poorer programming practice to use global variables.

Passing by value vs passing by reference

In all the examples so far, the values of arguments can be changed inside Sub procedures. This is because we have been passing the variables **by reference**. However, it is good programming practice to protect variables that you do not want to change in the Sub procedure just in case you make a mistake in the Sub procedure.

You can pass variables **by value** by placing parentheses around the arguments. Let's re-examine an earlier example that used a Sub procedure named **addy()**.

```
Option Explicit
Sub math2( )
Dim a As Double, b As Double, x As Double, y As Double
Dim ans As Double
a = 1:  b = 2:  x = 5:  y = 6
Call addy((a), (b), ans)                    ' Parentheses placed around a and b
MsgBox a & " + " & b & " = " & ans
Call addy((x), (y), ans)                    ' Parentheses placed around x and y
MsgBox x & " + " & y & " = " & ans
End Sub

Sub addy(n, m, result)
result = n + m
n = 1000                                    ' new line of code
m = 2000                                    ' new line of code
End Sub
```

Although two lines of code have been introduced to **addy()** that change **n** and **m**, the values of **a**, **b**, **x**, and **y** do not change in **math2()**. The parentheses have protected those arguments from accidentally being overwritten.

Topic 5: Function procedures

Sub procedures and Function procedures have many similarities and can be used for the same task. As you get more experience with computer programming, you will find that both procedures are easier to use for certain applications.

In general, Sub procedures are useful when you want to calculate new values for one or more arguments, while Function procedures are useful when you want to calculate a single value which is returned to the program that called it.

Here is the general syntax for a Function procedure:

```
Function  fname( arguments*) As* Type*
statements*
fname = expression*
Exit Function*
statements*
fname = expression*
End Function
```

Note: The items with an asterisk are often used, but technically are optional.

The **Exit Function** statement terminates the Function procedure when executed and any statements after the Exit Function statement will not be executed. Exit Function is usually used with an If structure (Topic 6) and never used alone.

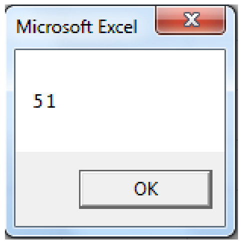
Example: Using a simple Function procedure

Write the following program and execute it.

```
Option Explicit
Sub test1( )
Dim a As Double, b As Double, c As Double, d As Double
a = 2:  b = 3:  c = -1:
d = funky(a, b, c)                    ' We do not call funky because it is not a Sub procedure
MsgBox d & " " & funky(c, a, b)
End Sub

Function funky(a, b, c) As Double
funky = a * b + c
End Function
```

Note: This Function procedure does not do anything useful. The example is designed to show how Function procedures work.

After running this program, a message box appears.


Microsoft Excel

51

Let's analyze the program.

```
Sub test1( )
Dim a As Double, b As Double, c As Double, d As Double
a = 2:  b = 3:  c = -1:
d = funky(a, b, c)
```

The variables **a**, **b**, and **c** are created and assigned values. The variable **d** also is created and is assigned the value that results from calculating the function **funky()** when the arguments are **a**, **b**, and **c**. After **funky()** is terminated, a single number replaces **funky(a,b,c)**

Notice how the Function procedure **funky()** is invoked in the same manner as built-in functions, such as Sin(). **You simply place them in a statement without using the Call statement**. The Call statement is only used for invoking Sub procedures.

```
d = funky(a, b, c)

Function funky(a, b, c) As Double
```

Here is the status of the arguments when **funky()** is invoked.

In **test1()**, arguments are named:	a	b	c
	2	3	-1
In **funky()**, arguments are named:	a	b	c

The variable **a** in **test1()** is linked with the variable **a** in **funky()**.
The variable **b** in **test1()** is linked with the variable **b** in **funky()**.
The variable **c** in **test1()** is linked with the variable **c** in **funky()**.
If **a**, **b**, or **c** change values in **funky()**, these changes will be passed back to **a**, **b**, and **c** in **test1()** after **funky()** is terminated.

```
funky = a * b + c
```

The Function procedure name, **funky()**, is used like a variable that will store a number with a decimal since we gave it a Double data type. In contrast, you should never assign a value to the name of a Sub procedure.

a * b + c is 5, and this value is assigned to **funky**. When the Function procedure is terminated, the updated values of **a**, **b**, and **c** are passed back to **test1()** and the number 5 replaces "**funky(a, b, c)**" in **test1()**. In this case **a**, **b**, and **c** in **funky()** are not changed so **a**, **b**, and **c** in **test1()** are unchanged.

The number 5 is assigned to the variable **d** in **test1()**. Here is the status of the variables in **test1()** before **funky()** is invoked for the second time.

Variables in **main()**:

a	b	c	d
2	3	-1	5

Back in **main()**, **funky()** is invoked again when using the MsgBox function.

```
MsgBox d & " " & funky(c, a, b)
```

As stated earlier, wherever a Function procedure is invoked, a value will be put in its place. This means the message box will display the value of **d** and **funky(c, a, b)**.

```
Function funky(a, b, c) As Double
```

Here are the status of the arguments when **funky()** is invoked for the second time.

In **test1()**, arguments are named:	c	a	b
	-1	2	3
In **funky()**, arguments are named:	a	b	c

The variable **c** in **test1()** is linked with the variable **a** in **funky()**.
The variable **a** in **test1()** is linked with the variable **b** in **funky()**.
The variable **b** in **test1()** is linked with the variable **c** in **funky()**.
If **c**, **a**, or **b** change values in **funky()**, these changes will be passed back to **a**, **b**, and **c** in **test1()** after **funky()** is terminated. However, we already saw that the value of the arguments are not changed in **funky()**.

```
funky = a * b + c
```

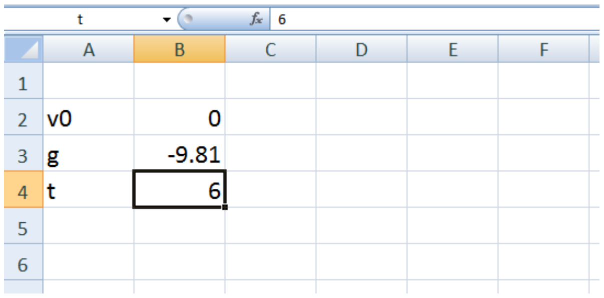
a * b + c is 1, and this value is assigned to **funky**. When the Function procedure is terminated, the number 1 replaces **funky(c,a,b)** in **test1()**.

Customized worksheet functions (User-defined functions)

All Function procedures can be invoked in the Excel worksheets too. Let's say you frequently need to use the equation for calculating the velocity of an object that is dropped from a building, **v0 + g*t** (Note: This equation assumes air friction is small).

Three inputs – **v0**, **g**, and **t** – are required to calculate this expression. We can write a Function procedure named **vel** that takes as input **v0**, **g**, and **t** from the Excel workbook, calculates the expression, and returns the value of the expression as output to the Excel workbook.

Enter the values of **v0**, **g**, and **t** on any worksheet.


t | 6

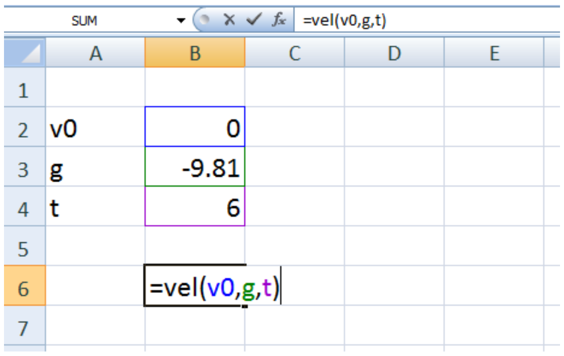
	A	B	C	D	E	F
1						
2	v0	0				
3	g	-9.81				
4	t	6				
5						
6						

Assign the name **v0** to cell B2, assign the name **g** to cell B3, and assign the name **t** to cell B4. Now open up the VBA Environment using Alt+F11, create a new module, and type the following code:

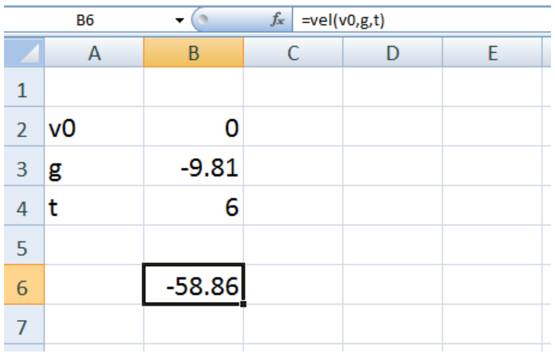
```
Function vel(v0, g, t) As Double
vel = v0 + g * t
End Function
```

This simple Function procedure, named **vel**, takes as input three arguments and calculates the value of **v0+g*t**. When the function is terminated, the value of **vel** is passed back to the cell where it is invoked.

Now we can use this Function procedure in our workbook. In some cell in the worksheet, type **=vel(v0,g,t)** and hit the Enter key. The value of **vel** is calculated based on the inputs **v0**, **g**, and **t**.

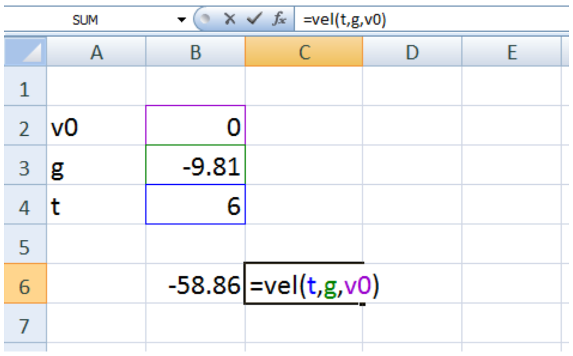

SUM | =vel(v0,g,t)

	A	B	C	D	E
1					
2	v0	0			
3	g	-9.81			
4	t	6			
5					
6		=vel(v0,g,t)			
7					

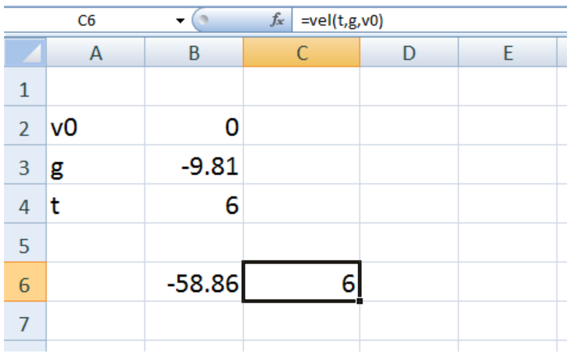

B6 | =vel(v0,g,t)

	A	B	C	D	E
1					
2	v0	0			
3	g	-9.81			
4	t	6			
5					
6		-58.86			
7					

What happens if we accidentally mix up the order of the arguments?


SUM | =vel(t,g,v0)

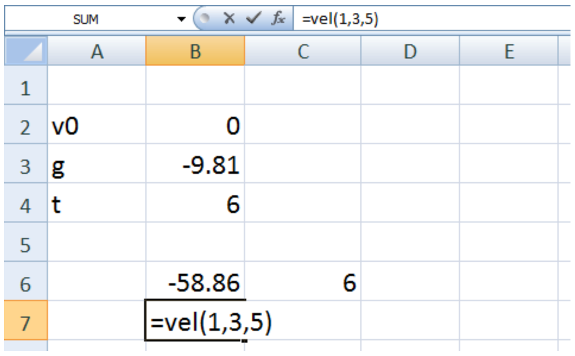
	A	B	C	D	E
1					
2	v0	0			
3	g	-9.81			
4	t	6			
5					
6		-58.86	=vel(t,g,v0)		
7					


C6 | =vel(t,g,v0)

	A	B	C	D	E
1					
2	v0	0			
3	g	-9.81			
4	t	6			
5					
6		-58.86	6		
7					

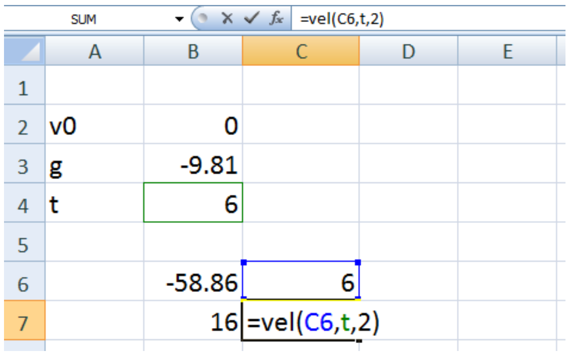
There are no error messages, but we get a different answer. When you invoke a function, the value of the first argument (**t**, in this last case) is passed to the first argument (**v0**) in the Function procedure. Similarly, the value of the second argument (**g**) in the worksheet is passed to the second argument (**g**) in the Function procedure, and the third argument (**v0**) in the worksheet is passed to the third argument (**t**) in the Function procedure. So the expression **v0 + g * t** becomes **6 + (-9.81) * 0**, or 6.

You can put numbers or cell references in the argument list too.



=vel(1,3,5)

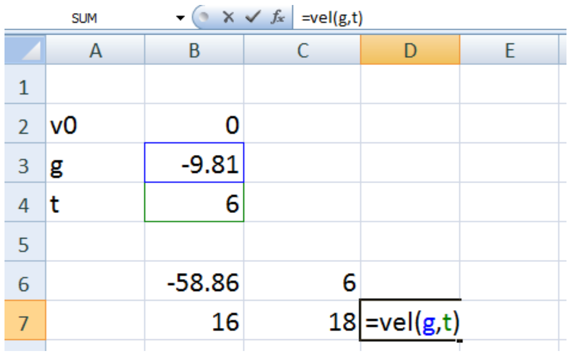
	A	B	C	D	E
1					
2	v0	0			
3	g	-9.81			
4	t	6			
5					
6		-58.86	6		
7		=vel(1,3,5)			



=vel(C6,t,2)

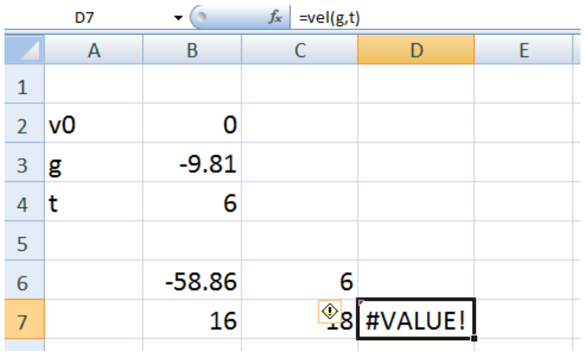
	A	B	C	D	E
1					
2	v0	0			
3	g	-9.81			
4	t	6			
5					
6		-58.86	6		
7		16	=vel(C6,t,2)		

Just make sure that you input the correct number of arguments when invoking the Function procedure.



=vel(g,t)

	A	B	C	D	E
1					
2	v0	0			
3	g	-9.81			
4	t	6			
5					
6		-58.86	6		
7		16	18	=vel(g,t)	



=vel(g,t)

	A	B	C	D	E
1					
2	v0	0			
3	g	-9.81			
4	t	6			
5					
6		-58.86	6		
7		16	18	#VALUE!	

Whoops! I only used two arguments instead of three when invoking the **vel()** Function procedure and an error message appears.

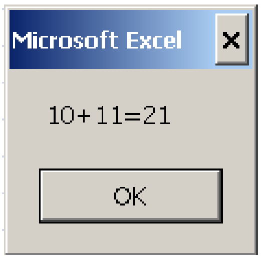
Calling procedures within procedures

Most codes used for real-world applications are lengthy and require the use of many procedures. Often procedures are called within procedures. Here is an example.

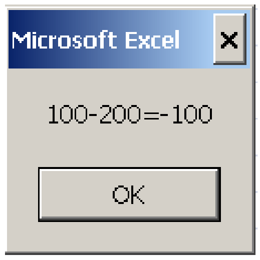
```
Option Explicit
Sub main()
Dim a As Double, b As Double, x As Double, y As Double
a = 10:  b = 11:  x = 100:  y = 200:
Call domath(a, b, x, y)
End Sub

Sub domath(a, b, x, y)
Dim diff As Double
Call calcdiff(x, y, diff)
MsgBox a & "+" & b & "=" & calcadd(a, b)
MsgBox x & "-" & y & "=" & diff
End Sub

Function calcadd(n, m) As Double
calcadd = n + m
End Function

Sub calcdiff(n1, n2, ans)
ans = n1 - n2
End Sub
```

Note: This program does not do anything useful. It is designed to show you how to use procedures within procedures.

This code produces two message boxes.



```
10+11=21
```



```
100-200=-100
```

Let’s analyze the program.

```
Sub main( )
Dim a As Double, b As Double, x As Double, y As Double
a = 10:  b = 11:  x = 100:  y = 200:
Call domath(a, b, x, y)

Sub domath(a, b, x, y)
```

The variables **a**, **b**, **x**, and **y** are created in **main()** and assigned values. The Sub procedure **domath()** is called and four arguments are passed down.

Here is the status of the arguments when **domath()** is called.

In **main()**, arguments are named:	a	b	x	y
	10	11	100	200
In **domath()**, arguments are named:	a	b	x	y

The Sub procedure **domath()** must finish before we continue in **main()**. Inside **domath()**,

```
Dim diff As Double
Call calcdiff(x, y, diff)

Sub diffy(n1, n2, ans)
```

The variable **diff** was not in the argument list when **domath()** was called, so it must be created. When **diff** is created, it is given a default value of 0. Now another Sub procedure called **calcdiff()** is called. **calcdiff()** must finish before we continue in **domath()**.

Here is the status of the arguments when **calcdiff()** is called.

In **domath()**, arguments are named:	x	y	diff
	100	200	0
In **calcdiff()**, arguments are named:	n1	n2	ans

calcdiff() must finish before we continue in **domath()**. Inside **calcdiff()**,

```
ans = n1 - n2
```

The variable **ans** is assigned a new value and **calcdiff()** is terminated. Here is the status of the arguments when **calcdiff()** is terminated.

In **domath()**, arguments are named:	x	y	diff
	100	200	-100
In **calcdiff()**, arguments are named:	n1	n2	ans

The variable **diff** now has a new value, but the variables **x** and **y** in **domath()** did not change values because the variables **n1** and **n2** in **calcdiff()** did not get assigned new values.

Here is the status of the variables in **domath()** after terminating **calcdiff()**.

Variables in domath():

a	b	x	y	diff
10	11	100	200	-100

After terminating **calcdiff()**, a message box will appear.

```
MsgBox a & "+" & b & "=" & calcadd(a, b)
```

Before we can display the message box, **calcadd(a, b)** must be evaluated.

```
Function calcadd(n, m) As Double
```

Here is status of the arguments when **calcadd()** is invoked:

In **domath()**, arguments are named:	a	b
	10	11
In **calcadd()**, arguments are named:	n	m

```
calcadd = n + m
```

The value of **calcadd** is calculated (21) and the Function procedure is terminated. The number 21 replaces the expression **calcadd(a, b)** in the **domath()** Sub procedure.

Therefore, **a & "+" & b & "=" & calcadd(a, b)** becomes **10+11=21**.

Also, the value of **a** and **b** in **domath()** do not change since **n** and **m** in **calcadd()** did not change. The last line of code in **domath()** is,

```
MsgBox x & "-" & y & "=" & diff
```

The values of **x**, **y**, and **diff** are 100, 200, and -100 respectively. Here is the status of the arguments when **domath()** is terminated.

In **main()**, arguments are named:	a	b	x	y
	10	11	100	200
In **domath()**, arguments are named:	a	b	x	y

a, **b**, **x**, and **y** in **main()** do not change because **a**, **b**, **x**, and **y** in **domath()** did not change. After **domath()** is terminated, the entire program ends.

Comparing Sub procedures and Function procedures

Here is a table comparing Sub procedures and Function procedures

Sub procedure	**Function** procedure
Can divide code into manageable pieces	Same
Can be invoked many times	Same
Communicate through arguments	Same
Can pass by reference or by value	Same
Variables locally defined by default	Same
Invoked using Call statement	Invoked using name only
Does not return a value where it is invoked	Returns a single value where it is invoked

Topic 6: Selective execution (If structures and Select Case structures)

In most real-life situations, you will perform different tasks based on your current circumstances. For example,

- You will eat only if you are hungry.
- You will shower if you just woke up.

Sometimes there are multiple options available to you. For example,

- At dinner time, you will eat at home unless your friends ask you to go eat at your favorite restaurant.

Sometimes multiple conditions must be met to perform a task. For example,

- You will drive to a particular restaurant to eat if you are hungry, you possess a driver's license, if a car is available to you, and if you have at least $50 in your bank account or a friend pays for you.

So far our programs have executed every statement (except when Exit Sub and Exit Function were used). But most of the time, we want our programs to selectively execute statements based on the current circumstances.

Before we discuss how to perform the selective execution of statements, we first must discuss relational operators.

Relational Operators

Relational operators allow us to compare two quantities. After comparing the two quantities, either a **True** or **False** will replace the relational expression.

=	**equal to**
<>	**not equal to**
<	**less than**
>	**greater than**
<=	**less than or equal to**
>=	**greater than or equal to**

Note: Unfortunately, VBA uses the same symbol (**=**) for assignment and the **equal to** relational operator. Most other languages use different symbols.

When you use a relational operator, you ask the question: **Is the left quantity _____ the right quantity?** where the blank space is the relational operator. For example,

```
Option Explicit
Sub main()
Dim a As Double, b As Double
a = 5:  b = 6
MsgBox a > b
MsgBox 2 * a <> b+50
End Sub
```

The first expression asks, is **a** greater than **b**? Is 5 greater than 6? The answer is False, which is displayed in a message box.
The second expression asks, is **2*a** not equal to **b+50**? Is 10 not equal to 56? The answer is True, which is displayed in a message box.

The answer to a relational expression can be stored in a variable of Boolean data type. The following code will accomplish the same task as the previous example.

```
Option Explicit
Sub main()
Dim a As Double, b As Double, c As Boolean, d As Boolean
a = 5:  b = 6
c = a > b
d = 2 * a <> b + 50
MsgBox c
MsgBox d
End Sub
```

Logical Operators

Relational operators allow us to compare alphanumeric quantities. If we want to compare Boolean data, we can use logical operators. We will discuss three logical operators in the course: **And**, **Or**, and **Not**.

In the following examples, assume the value of **a** is 5 and the value of variable **b** is 6.

<u>And</u>	are both quantities True?
a < b And b = 6	True And True → True
a > b And b = 6	False And True → False
a < b And b = 7	True And False → False
a > b And b = 7	False And False → False

<u>Or</u>	is either quantity True?
a < b Or b = 6	True And True → True
a > b Or b = 6	False And True → True
a < b Or b = 7	True And False → True
a > b Or b = 7	False And False → False

<u>Not</u>	what is the opposite of the quantity on the right?
Not a < b	Not True → False
Not a > b	Not False → True

If-Then structure

If you want a set of statements to be executed if a test condition is True, you can use an If-Then structure. The general form of the If-Then structure is the following:

```
If (test condition) Then
  statements executed if test condition is True
End If
```

Example: Using and If-Then structure

Write the following program and execute it.

```
Option Explicit
Sub main()
Dim x As Double, y As Double
x = 40: y = 50
If (x > 10) Then
   MsgBox "Inside If structure"
   y = 300
End If
MsgBox "y = " & y
End Sub
```

The parentheses around the test condition **x > 10** are optional, but they help make the code easier to read. The two commands inside the If structure,

```
MsgBox "Inside If structure"
y = 300
```

will get executed only if **x > 10** is True when the If-Then structure begins. In this case, **x** is 40 so those two commands will get executed. One message box will state **Inside If structure** and the second message box will state **y = 300**.

If-Then-Else structure

If you want a set of statements to be executed if a test condition is True and another set of statements to be executed if the test condition is False, you can use an If-Then-Else structure. The general form of the If-Then-Else structure is the following:

```
If (test condition) Then
  statements executed if test condition is True
Else
  statements executed if test condition is False
End If
```

Example: Using If-Then-Else structure

This program will determine whether the user should buy a soda.

```
Option Explicit
Sub buysoda()
Dim costsoda As Double, x As Double
costsoda = Val(InputBox("What is the cost of the soda?"))
If (costsoda > 0.75) Then
   MsgBox "Soda is too expensive"
Else
   MsgBox "Buy that soda"
End If
End Sub
```

A message box will state **"Soda is too expensive"** if **costsoda > 0.75** is True.
A message box will state **"Buy that soda"** if **costsoda > 0.75** is False.

Notice that we do not know which message box will be displayed until the user inputs a value for **costsoda**.

You may have noticed that the statements inside the If structures are indented 3 or 4 spaces. Although it is not technically necessary, such indentation is considered good programming practice and I highly recommend that you do this too. It will make your code much easier to read, especially when you create longer, more complicated If structures.

If-Then-ElseIf structure

The If-Then-Else structure only allows us to test one condition. Certain statements will be executed if the test condition is True, other statements will be executed if the test condition is False. But real-life problems are more complicated and often require the ability to take multiple actions based on multiple test conditions. For situations like these, use an If-Then-ElseIf structure. The general form for this structure is the following:

```
If (test condition 1) Then
   statements if test condition 1 is True
ElseIf (test condition 2) Then
    statements if test condition 2 is True
ElseIf (test condition 3) Then
    statements if test condition 3 is True
...
Else
    statements if none of the test conditions are True
End If
```

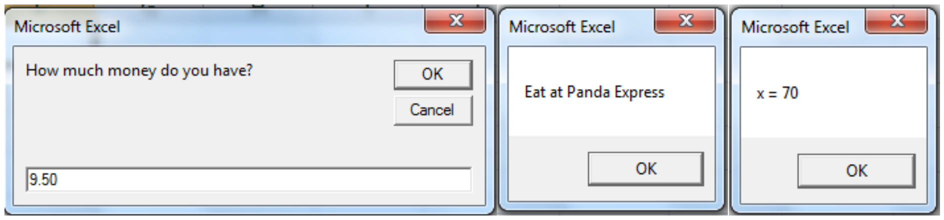
The program proceeds down the list of test conditions until a True test condition is found. Once a True test condition is found, the statements between the test condition and the next **Else If** are executed and the program jumps to the **End If**, ignoring the remaining test conditions even if some of the remaining test conditions are true.

Example: Using an If-Then-ElseIf structure

Let's say you will eat Top Ramen if you have less than $3, eat at McDonald's if you have $3-5, eat at Subway if you have $5-10, and eat at Olive Garden if you have over $10.

```
Option Explicit
Sub eat( )
Dim funds As Double
funds = Val(InputBox("How much money do you have?"))
If (funds > 10) Then
   MsgBox "Eat at Olive Garden"
ElseIf (funds >= 5 And funds < 10) Then
   MsgBox "Eat at Panda Express"
ElseIf (funds >= 3 And funds < 5) Then
   MsgBox "Eat at McDonalds"
ElseIf (funds < 10000) Then            ' This test condition is used to explain a concept below
   MsgBox "I like Pizza!"
Else
   MsgBox "Eat Top Ramen"
End If
End Sub
```

Run the program and type **9.50** when prompted to enter the amount of money you have.



```
How much money do you have?
9.50
```

```
Eat at Panda Express
```

```
x = 70
```

Let's analyze this program.

```
funds = Val(InputBox("How much money do you have?"))
If (funds > 10) Then
```

The value of **funds** is obtained from the user (assume the user typed **9.50** in the input box). Then we come to the first test condition. Is the value of **funds** greater than 10? Is 9.50 greater than 10? That is False. So the program skips the following statement,

```
   MsgBox "Eat at Olive Garden"
```

and then evaluates the second test condition.

```
ElseIf (funds >= 5 And funds < 10) Then
```

Is the value of **funds** greater than or equal to 5 **And** is the value of **funds** less than 10? Is 9.50 greater than or equal to 5 and is 9.50 less than 10? True And True yields True. So the program executes the following statement,

```
        MsgBox "Eat at Panda Express"
```

then the program jumps to the End If, ignoring the other test conditions even though the following test condition is also True,

```
    ElseIf (funds < 10000) Then
```

The statement after this second True test condition is not executed,

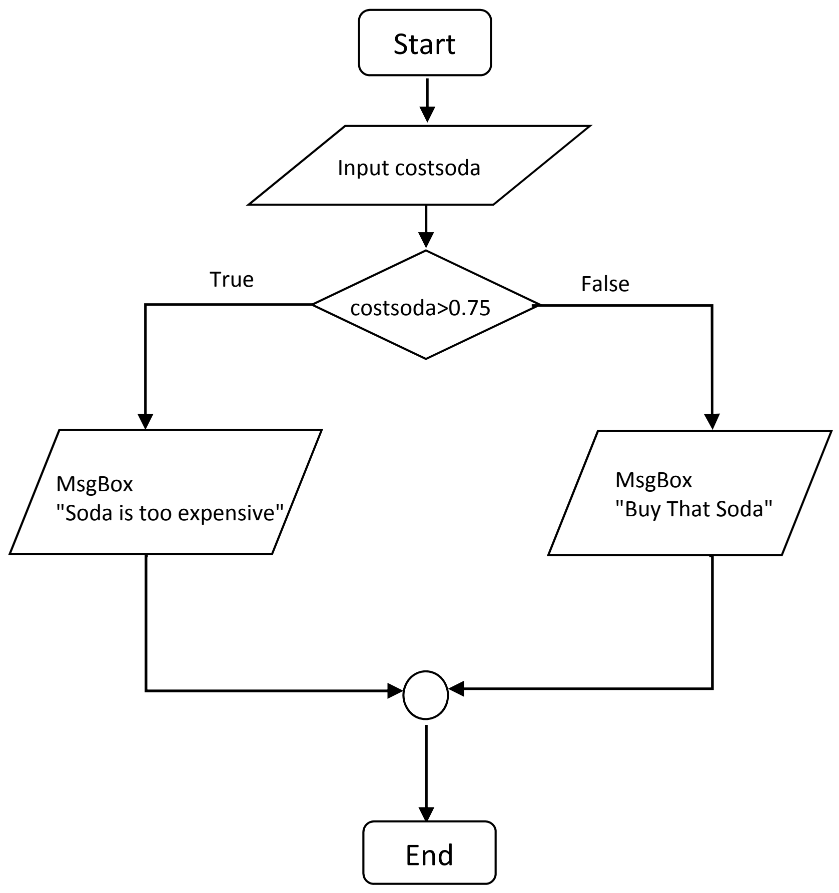
```
        MsgBox "I like Pizza!"
```

Flowcharts

Often it is helpful to visualize a program using flow charts. Here are some of the most common symbols in flow charts.

▢	Terminal: Represents the beginning or end of a program
→	Flow lines: Represents the flow of logic
▭	Process: Represents calculations or data manipulation
▱	Input/Output: Represents data input or output
◇	Decision: Represents a comparison that determines which path the program will take
○	Junction: Represents the confluence of flow lines

The example of an If-Then-Else structure involving the cost of a soda discussed earlier can be represented in the follow manner:



Select Case Structure

If you have many test conditions, but they all only require testing the value of one variable or expression, you may want to use a Select Case structure. The general form of this structure is the following:

```
Select Case  test variable
  Case valuelist1
    statements if test variable equals anything in valuelist1
  Case valuelist2
    statements if test variable equals anything in valuelist2
  ...
  Case Else
    statements if test variable does not equal anything in any of the valuelists
End Select
```

Once a valuelist is encountered that contains the value of the test variable, all subsequent valuelists are ignored and the program proceeds to the End Select statement.

Example: Using Select Case with a numerical test variable

This program asks a student for the current hour, then tells the student what he or she should do.

```
Option Explicit
Sub school( )
Dim hour As Integer, message As String
hour = Val(InputBox("What is the current hour?"))
Select Case hour
Case 8
  message = "Go to School"
Case 9, 10, 11
  message = "Pay attention in class"
Case 12 To 15
  message = "Lunch time"
Case Is > 15
  message = "Not at school"
Case Else
  message = "Not valid hour"
End Select
MsgBox message
End Sub
```

After running this code, a message box appears that states "Pay attention in class".

Notice that we can specify possible values for the test variable (**hour**) in many different ways.
- Single values (**Case 8**)
- Many possible values using commas (**Case 9, 10, 11**)
- Range of values using the **To** keyword (**Case 12 To 15**)
- Open-ended range of values using the **Is** keyword (**Case Is > 15**).

Example: Using Select Case with a String test variable

If your test variable is a string, you must put the different cases in double quotes. Here is a program that will ask the user for their favorite book, then output the author's name

```
Option Explicit
Sub favbook()
Dim book As String, author As String
book = InputBox("What is your favorite book?")
book = LCase(book)                    ' Now book contains lowercase letters only
Select Case book
  Case "moby dick"                    ' All valuelists should contain lowercase letters only
    author = "Herman Melville"
  Case "war of the worlds", "the time machine"
    author = "H. G. Wells"
  Case Else
    author = "unknown"
End Select
MsgBox "The author is " & author
End Sub
```

Let's analyze this code.

```
book = InputBox("What is your favorite book?")
```

In this example, let's assume the user input "The Time Machine". When using strings, it is important to remember that VBA distinguishes between uppercase and lowercase letters. "The Time Machine" and "THE TIME MACHINE" are treated differently.

```
book = LCase(book)
```

If you are going to be making comparisons with character strings, it is often a good idea to reduce them to all lowercase or all uppercase letters. **LCase()** is a built-in function that takes a string as input and outputs the same string, but lowercase. The output from **LCase(book)** is whatever string the user input, but in lowercase letters. So now **book** contains the string "the time machine".

```
Select Case book
  Case "moby dick"
```

The value stored in **book**, which is the string "the time machine", will be compared against different valuelists. The first case is "moby dick". Since this is different than "the time machine", we move to the next Case.

```
  Case "war of the worlds", "the time machine"
    message = "H. G. Wells"
```

The value stored in **book** is not equal to "war of the worlds", but is equal to "the time machine". The string "H. G. Wells" is stored in **author**, the program jumps to **End Select**, and the message box displays "H. G. Wells is the author."

Nested decision structures

It is common to place decision structures within decision structures.

Example: Using nested If structures

A university is trying to promote height diversity by allowing in a higher percentage of students who are relatively tall or relatively short. However, the school still cares about SAT scores. You get an overall score based on your height and SAT score. If it is above 100 points, you get into the school.

Male heights		Female heights	
Greater than 6 feet	+10	Greater than 6 feet	+25
Between 5 and 6 feet	-15	Between 5 and 6 feet	-10
Less than 5 feet	+20	Less than 5 feet	+5

```
Option Explicit
Sub school()
Dim gender As String, h As Double, sat As Integer, score As Double, msg As String
gender = InputBox("Enter gender, M or F")  :   gender = LCase(gender)
h = Val(InputBox("enter height, in feet"))
sat = Val(InputBox("Enter SAT score, 0-2400"))  :   score = sat / 20
If (gender = "m") Then          ' If the user enters "m", we enter the outer If structure here
  If (h > 6) Then
    score = score + 10
  ElseIf (h < 5) Then
    score = score + 20
  Else
    score = score - 15
  End If
ElseIf (gender = "f") Then      ' If the user enters "f", we enter the outer If structure here
  If (h > 6) Then
    score = score + 25
  ElseIf (h < 5) Then
    score = score + 5
  Else
    score = score - 10
  End If
End If
msg = "Your score is " & score & ". "
If (score >= 100) Then
  MsgBox msg & "You are in!"
Else
  MsgBox msg & "Sorry."
End If
End Sub
```

Run the program and input your gender, height, and SAT score. Can you predict whether you will get into the school?

Topic 7: Repetitive Execution (Loops)

The true power of a computer lies in its ability to execute many repetitive statements very quickly.

For example, let's say we want a computer program to add up all numbers between 1 and 20000. It would take a human many hours to accomplish this task manually, but a computer program could finish within less than second. Even if a human could use a mathematical trick to determine the sum quickly, he or she still would be far slower than a computer. Notice that adding all numbers between 1 and 20000 requires you to repeat the same set of commands 20000 times: Take a number and add it to a running total, then increase that number by 1 and repeat.

Another example: Calculate the height of a ball dropped from a building every 0.1 seconds until it hits the ground. In this case, we would perform the exact same calculations while increasing the time by 0.1 until a test condition is met.

Loops are used to execute a series of statements one or more times. There are two types of loops in VBA: (1) **For loops** and (2) **Do loops**. Both loops can be used to solve the same problem, but each is better suited for different tasks as explained below.

For loops

If you want to execute a set of commands a certain number of times, you should use a **For** loop. The general format for a **For** loop is the following:

```
For  var = startval  To  endval   Step delta
   statements
Next var
```

var = counter variable
startval = initial value of **var**
endval = end of range for **var**
delta = change in **var** each loop (sometimes called the "step size")
if **delta** > 0, loop terminates when **var > endval**
if **delta** < 0, loop proceeds when **var < endval**

By default the step size is set to 1. If you omit **Step delta**, VBA will assume you meant **Step 1**.

Before entering the loop, **var** is assigned the value of **startval**. We enter the loop if either of the following conditions are True.

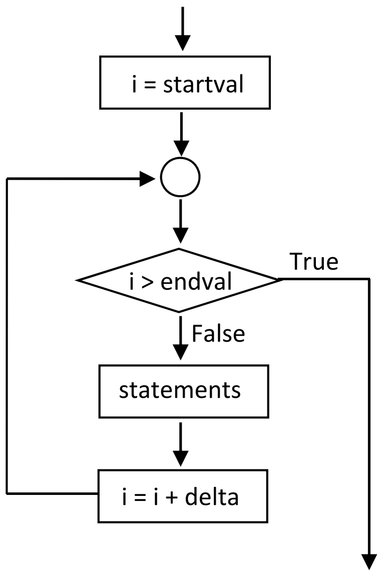
- If **delta** > 0 And **var** ≤ **endval** (most common scenario)
- If **delta** < 0 And **var** ≥ **endval** (not as common)

Assuming we have entered the loop, when the program reaches the **Next** statement, **var** is changed by **delta**. The new value of **var** will determine whether we proceed through the loop again.

- If **delta** > 0 And **var** ≤ **endval**, the loop repeats with the new value of **var**. Otherwise the loop is terminated.
- If **delta** < 0 And **var** ≥ **endval**, the loop repeats with the new value of **var**. Otherwise the loop is terminated.

Notice that the statements in the loop are indented. Although technically not necessary, it is considered good programming practice to indent 3 or 4 space because it will help keep your code neat and orderly.

Here is the flowchart for a For loop in which the counter variable is named **i** and **delta** > 0:

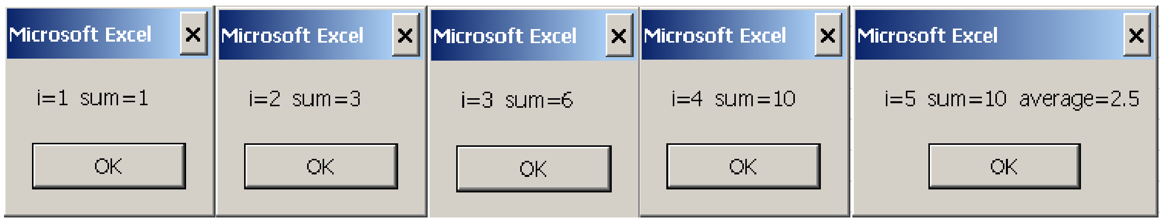


Example: Using a For loop

This program will add all integers from 1 to n.

```
Option Explicit
Sub loopy()
Dim i As Long, n As Long, sum As Long
Dim avg As Double
n = 4:  sum = 0
For i = 1 To n Step 1
   sum = sum + i
   MsgBox "i=" & i & "  sum=" & sum
Next i
avg = sum / n
MsgBox "i=" & i & "  sum=" & sum & "  average=" & avg
End Sub
```

After running the code, five message boxes will appear.



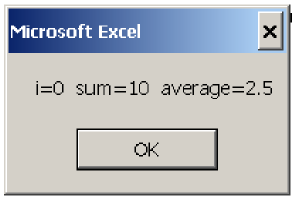
```
i=1  sum=1
i=2  sum=3
i=3  sum=6
i=4  sum=10
i=5  sum=10  average=2.5
```

The counter variable **i** changes from 1 to 2 to 3 to 4 to 5. When **i** is assigned the value of 5, which is greater than **endval** (4), the loop terminates and statements after the **Next i** are executed.

You can have negative steps as well. Let's redo the example above using a step size of -1.

```
Option Explicit
Sub loopy( )
Dim i As Long, n As Long, sum As Long
Dim avg As Double
n = 4:  sum = 0
For i = n To 1 Step -1
  sum = sum + i
Next i
avg = sum / n
MsgBox "i=" & i & "  sum=" & sum & "  average=" & avg
End Sub
```

I have removed the MsgBox function inside the loop, so only one message box will appear. The value of i changes from 4 to 3 to 2 to 1 to 0. When i is assigned the value 0, which is less than **endval** (1), the loop is terminated.


```
Microsoft Excel
i=0  sum=10  average=2.5
```

Notice in both examples I set the running total variable (**sum**) to 0 before beginning the loop. It is good programming practice to reset variables that are used for summation to 0 just in case you used them earlier in the program. For these small examples it is clear that **sum** was never used prior to the loop, but imagine if the loop was at the end of a program that is 1000 lines long. Would you be 100% confident that **sum** is never used prior to the loop? It is better to be safe than sorry.

Terminating For loops early

If you want to terminate a For loop early when a test condition is met, you can use an **Exit For.**

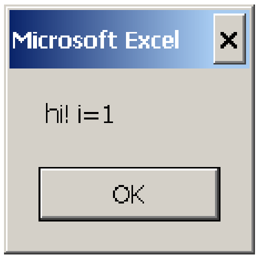
```
If (test condition) Then
  Exit For
End If
```

Or

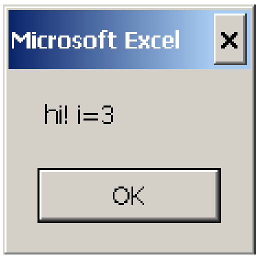
If (test condition) Then Exit For (This is another way of writing a simple If-Then structure)

Example: Using Exit For to terminate a loop early

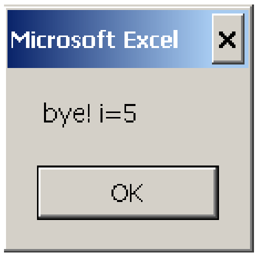
```
Option Explicit
Sub loopy()
Dim i As Integer
For i = 1 To 100 Step 2
  If (i > 4) Then Exit For                    ' Breaks loop early if i>4 is True
  MsgBox "hi! i=" & i
Next i
MsgBox "bye! i=" & i
End Sub
```



```
hi! i=1
```



```
hi! i=3
```



```
bye! i=5
```

Notice that statement **MsgBox "hi! i=" & i** was not executed when **i** had the value of 5 because the loop was terminated first.

Note: In all the examples show thus far, **var** and **delta** have been integers. However, they do not have to be integers. Here is a program that is similar to the previous example:

Example: Using non-integer counter variables in a For loops

In this example, the counter variable **t** is incremented by 0.5 after each iteration.

```
Option Explicit
Sub loopy()
Dim t As Double
For t = 0 To 100 Step 0.5
  If (t > 4.2) Then Exit For
  MsgBox "Inside For:  t = " & t
Next t
MsgBox "Outside For!  t = " & t
End Sub
```

Can you predict what the output would be?

Do loops

If you want to terminate a loop when a test condition is met, you should use a **Do** loop. There are two types of Do loops: **Do While** and **Do Until**. These loops have the following format:

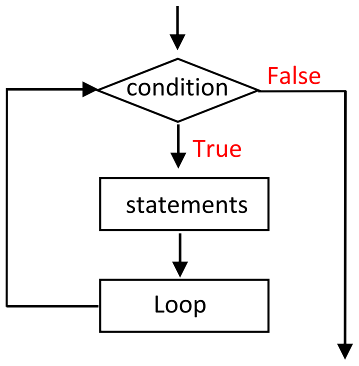
```
Do While (test condition)              Do Until (test condition)
  statements                             statements
Loop                                   Loop
```

The parentheses around **test condition** are optional, but can help make your code look neat and organized.

In a **Do While** loop, the loop is entered if the **test condition** is True.
After the statements in the loop are executed, the loop checks if the **test condition** is still True.
- If the test conditions is still True, the loop repeats
- If the test condition is False, the loop is terminated (the loop will repeat while the test condition is True)

In a **Do Until** loop, the loop is entered if the **test condition** is False.
After the statements in the loop are executed, the loop checks if the **test condition** is still False.
- If the test conditions is still False, the loop repeats
- If the test condition is True, the loop is terminated (the loop will repeat until the test condition is True)

The **Do While** loop has the following flowchart:



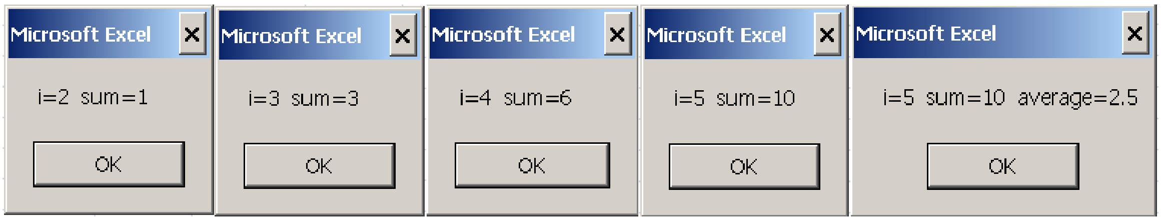
The **Do Until** loop is similar, except the red True and False are switched.

In the examples that follow, I will be using Do While loops instead of Do Until loops. However, you may use whichever loop you prefer.

Both Do loops and For loops can be used to accomplish the same task. In this next example, a Do While loop is used to calculate the sum and average of integers between 1 and 4. Note: This task is better suited for a For loop, but I wanted to start with an example you are familiar with.

Example: Using Do While loop

```
Option Explicit
Sub loopy()
Dim i As Long, n As Long, sum As Long
Dim avg As Double
sum = 0:  i = 1:  n = 4
Do While (i <= n)
  sum = sum + i
  i = i + 1                    ' What happens if we switch this line with the previous line? Order matters!
  MsgBox "i=" & i & "  sum=" & sum
Loop
avg = sum / n
MsgBox "i=" & i & "  sum=" & sum & "  average=" & avg
End Sub
```



```
i=2  sum=1
```

```
i=3  sum=3
```

```
i=4  sum=6
```

```
i=5  sum=10
```

```
i=5  sum=10  average=2.5
```

If we wanted to do the same task with a **Do Until** loop, we could replace

```
Do While (i <= n)
```

with

```
Do Until (i > n)
```

Important: With **Do** loops, it is necessary to give an initial value to the variable(s) used in the **test condition**. In the previous example, the variable **i** is assigned the value 1.

Additionally, the variable(s) in the **test condition** should be updated in each iteration of the loop in order to avoid **infinite loops**. In the previous example, the variable **i** is incremented by 1 in each iteration.

Example: Infinite loop

```
Option Explicit
Sub infiniteloopy()
Dim i As Long, n As Long, sum As Long
sum = 0:  i = 1:  n = 4
Do Until (i > n)
  sum = sum + i
Loop
End Sub
```

The value of the variable **i** never changes and the loop would go on forever. You may get lucky and get an overflow error message if a variable is assigned a number that is too large, but it is possible you will not get an error message. If you get stuck in an infinite loop, type the **Esc** key many times or try **Ctrl+PauseBreak**. SAVE YOUR WORK OFTEN in case you get stuck in an infinite loop and Excel crashes.

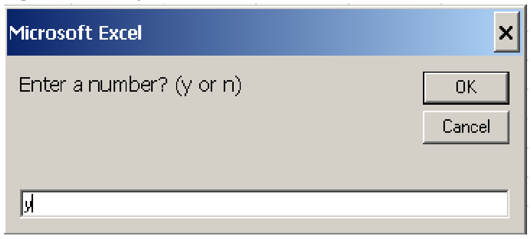
Going back to the **loopy** macro, it is easier to use **For** loops in that particular case because we need to execute a set of statements a known amount of times. Let's examine a situation where it is more natural for a test condition to terminate the loop.

Example: Using Do While loop

Allow the user to input as many numbers as he or she pleases. The program will output the numbers to the screen in column A, starting from A2. The sum and average will be displayed underneath the values.

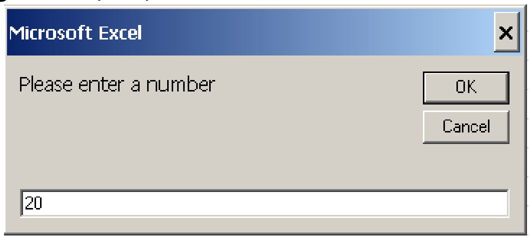
```
Option Explicit
Sub loopy()
Dim count As Integer
Dim sum As Double, num As Double, avg As Double, answer As String
sum = 0:  count = 0
Sheets("Sheet1").Select:  Range("A1").Select
answer = InputBox("Enter a number? (y or n)")      ' Need an initial value to enter the loop
Do While (answer = "y")
  num = Val(InputBox("Please enter a number"))
  sum = sum + num
  count = count + 1
  ActiveCell.Offset(1,0).Select: ActiveCell.Value = num
  answer = InputBox("Enter a number? (y or n)")
Loop
If(count > 0) Then                              ' Needed to prevent an error message if count is zero
  ActiveCell.Offset(2,0).Select: ActiveCell.Value = sum
  avg = sum / count
  ActiveCell.Offset(1,0).Select: ActiveCell.Value = avg
Else
  MsgBox "Cannot calculate avg since no values were input"
End If
End Sub
```

Here is the program output if I wanted to find the average of 20, 14, and 12.


Microsoft Excel

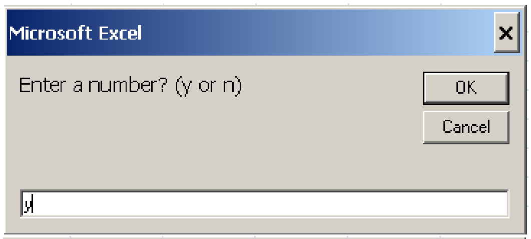
Enter a number? (y or n)

y


Microsoft Excel

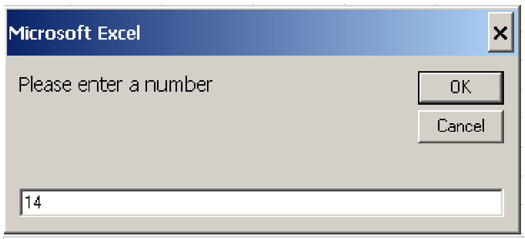
Please enter a number

20



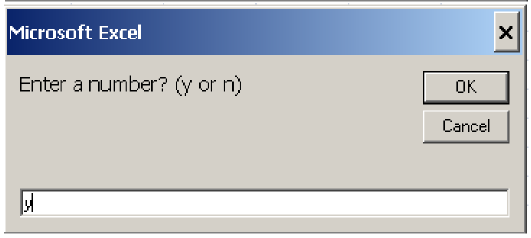
Enter a number? (y or n)

y



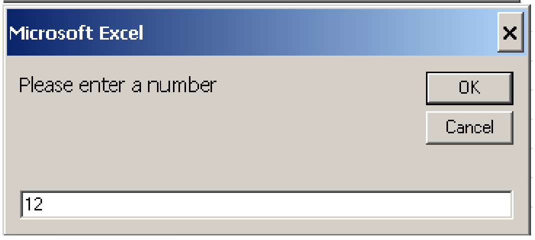
Please enter a number

14



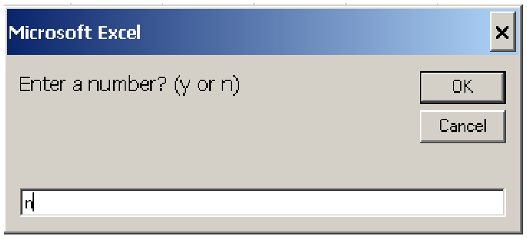
Enter a number? (y or n)

y



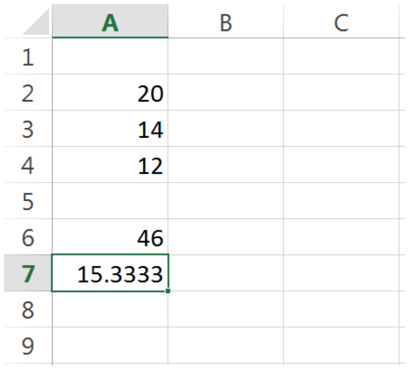
Please enter a number

12



Enter a number? (y or n)

n



	A	B	C
1			
2	20		
3	14		
4	12		
5			
6	46		
7	15.3333		
8			
9			

Terminating loops early

Do loops can be terminated early with the **Exit Do** statement,

```
If (test condition) Then
  Exit Do
End If
```

Or

```
If (test condition) Then Exit Do
```

If the **test condition** is True, the **Do** loop will be terminated immediately.
If the **test condition** is False, the loop will continue.

Nested loops

Often it is useful to place loops within loops, especially when using multidimensional arrays (Topic 9). When using nested loops, the inner loop must begin and end inside the outer loop.

The following pair of nested loops are okay because the j loop begins and ends inside the i loop.

```
For i = 1 To 2
  For j = 7 To 5 Step -2
    'statements
  Next j
Next i
```

The following pair of nested loops are illegal because the j loop ends outside the i loop.

```
For i = 1 To 2
  For j = 7 To 5 Step -2
    'statements
  Next i
Next j
```

Example: Using nested loops

```
Option Explicit
Sub loopy( )
Dim i As Integer, j As Integer
For i = 1 To 2 Step 1
  For j = 7 To 5 Step -2
    MsgBox "i = " & i & ",  j = " & j
  Next j
  MsgBox "i = " & i & ",  j = " & j
Next i
MsgBox "i = " & i & ",  j = " & j
End Sub
```

Note: This program does not do anything interesting. It is designed to demonstrate how nested loops work.

Run the program and note the values in the message boxes.
i is initially 1, j is initially 7 then 5 then 3 (which breaks the inner loop).
Then i increases to 2, j is initially 7 then 5 then 3 (which breaks the inner loop).
Then i increases to 3 (which breaks the outer loop).

So nested loops hold the outer variable (i) constant while cycling through all the values of the inner variable (j). Then the outer variable will change by the step size (+1) and the inner variable will again cycle through all its values.

Loops and Excel spreadsheets

Many VBA programs are required to access a lot of data from an Excel spreadsheet. For example, you could make a program that obtains test scores from a column in a spreadsheet and then calculates the average of those scores.

It is good programming practice to make your programs able to handle a wide variety of situations. Your program should work if 5, 50, or 500 test scores are listed in the column.

This can be accomplished with the help of the **Selection.End( ).Select** command, which moves the active cell to the edge of the current data block. There are four options that can be inserted into the parentheses.

Selection.End(xlDown).Select
- Selects the last row in the current column of the data block.
- Same as **Ctrl + down arrow** command in Excel.

Selection.End(xlToRight).Select
- Selects the last column in the current row of the data block.
- Same as **Ctrl + right arrow** command in Excel.

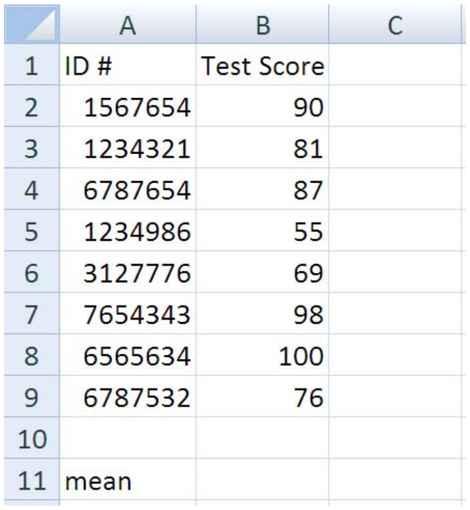
Selection.End(xlUp).Select
- Selects the first row in the current column of the data block.
- Same as **Ctrl + up arrow** command in Excel.

Selection.End(xlToLeft).Select
- Selects the first column in the current row of the data block.
- Same as **Ctrl + left arrow** command in Excel.

Other useful commands are:
ActiveCell.Row – Returns the row number of the current active cell.
ActiveCell.Column – Returns the column number of the current active cell. Column A is 1, column B is 2, etc.

Example: Obtaining data sets from spreadsheets using loops

The test scores of eight students are entered into Excel spreadsheet shown below. We want to create a VBA macro that will calculate the average (mean) score and place that value in the cell that is two rows below the last score. Although only eight scores are input, this macro should work for any number of test scores.



	A	B	C
1	ID #	Test Score	
2	1567654	90	
3	1234321	81	
4	6787654	87	
5	1234986	55	
6	3127776	69	
7	7654343	98	
8	6565634	100	
9	6787532	76	
10			
11	mean		

```
Option Explicit
Sub meanscore( )
Dim i As Long, nrow As Long, firstrow As Long
Dim lastrow As Long, totalrows As Long
Dim mean As Double
Sheets("Sheet1").Select
Range("B2").Select                  ' Start at top of list
firstrow = ActiveCell.Row           ' Determine # of first row

Selection.End(xlDown).Select        ' Jump to last row in column
lastrow = ActiveCell.Row            ' Determine # of last row
totalrows = lastrow - firstrow + 1  ' Calculate total # rows

Range("B2").Select                  ' Jump back to top of list
mean = 0
For i = 1 To totalrows              ' We will add totalrows values
  mean = mean + ActiveCell.Value
  ActiveCell.Offset(1, 0).Select
  ' mean = mean + ActiveCell.Offset(i, 0).Value      'A more efficient method than the previous two
                                                     '  lines.
Next i
mean = mean / totalrows             ' We need to jump down one more row.
ActiveCell.Offset(1, 0).Select
ActiveCell.Value = mean
End Sub
```

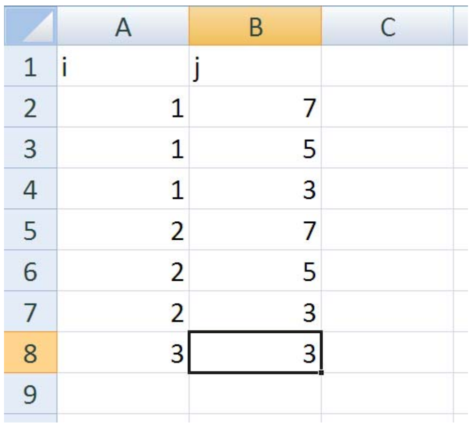
When this macro is executed, the mean (85) is placed in the cell that is two rows below the last score. Notice that the macro will work whether 8, 80, or 800 test scores are input into the Excel workbook.

Example: Outputting multiple columns of data using loops

Let’s revisit the nested loop example presented earlier in this topic. Instead of displaying the values via message boxes, we will output the information to a spreadsheet. Column A will contain the values of i, while column B will contain the values of j.

```
Option Explicit
Sub loopy( )
Dim i As Integer, j As Integer
Sheets("Sheet1").Select
Range("A1").Select: ActiveCell.Value = "i"
Range("B1").Select: ActiveCell.Value = "j"
For i = 1 To 2
  For j = 7 To 5 Step -2
    ActiveCell.Offset(1, -1).Select: ActiveCell.Value = i
    ActiveCell.Offset(0, 1).Select: ActiveCell.Value = j
  Next j
  ActiveCell.Offset(1, -1).Select: ActiveCell.Value = i
  ActiveCell.Offset(0, 1).Select: ActiveCell.Value = j
Next i
ActiveCell.Offset(1, -1).Select: ActiveCell.Value = i
ActiveCell.Offset(0, 1).Select: ActiveCell.Value = j
End Sub
```

After running the program, the following values are output to the screen.



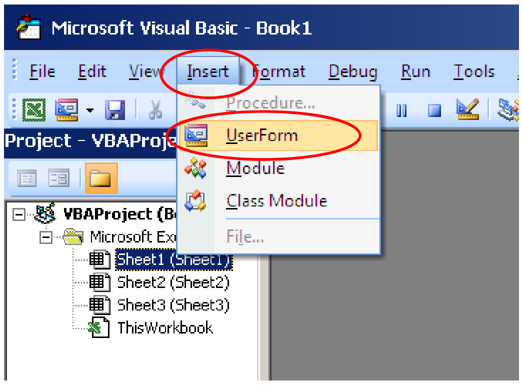
	A	B	C
1	i	j	
2	1	7	
3	1	5	
4	1	3	
5	2	7	
6	2	5	
7	2	3	
8	3	3	
9			

Topic 8: Custom Dialogue Boxes (UserForms)

Most of the previous lessons have involved concepts that are common to all computer languages. All languages have variables, Function procedures (usually called functions) and/or Sub procedures (usually called subroutines), If structures, loops, arrays, and input/output capabilities.

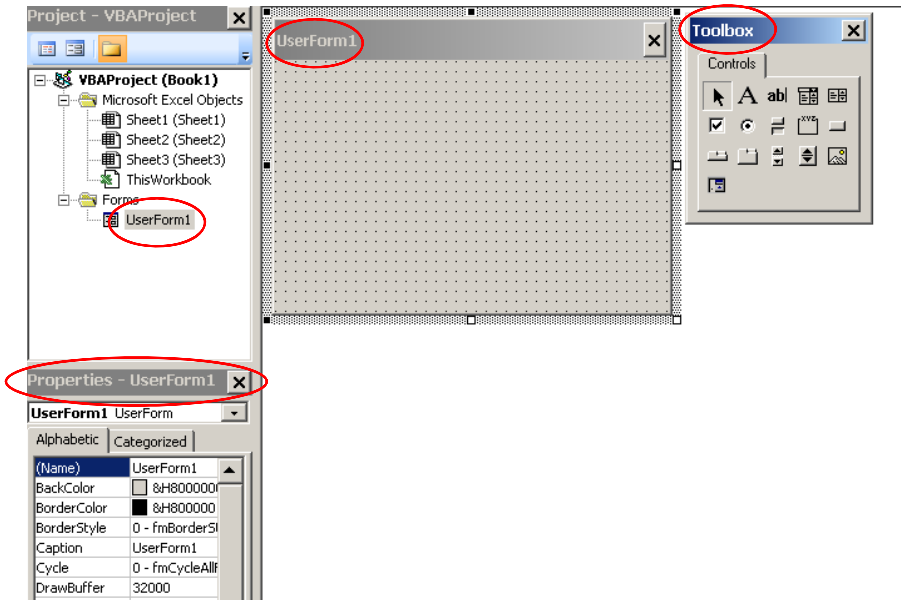
One of the advantages of using VBA is that you can create user-friendly graphical user interfaces easily. These graphical user interfaces are sometimes called custom dialogue boxes or UserForms.

First go to the Visual Basic Editor. UserForms are created by clicking on **Insert** > **UserForm**.



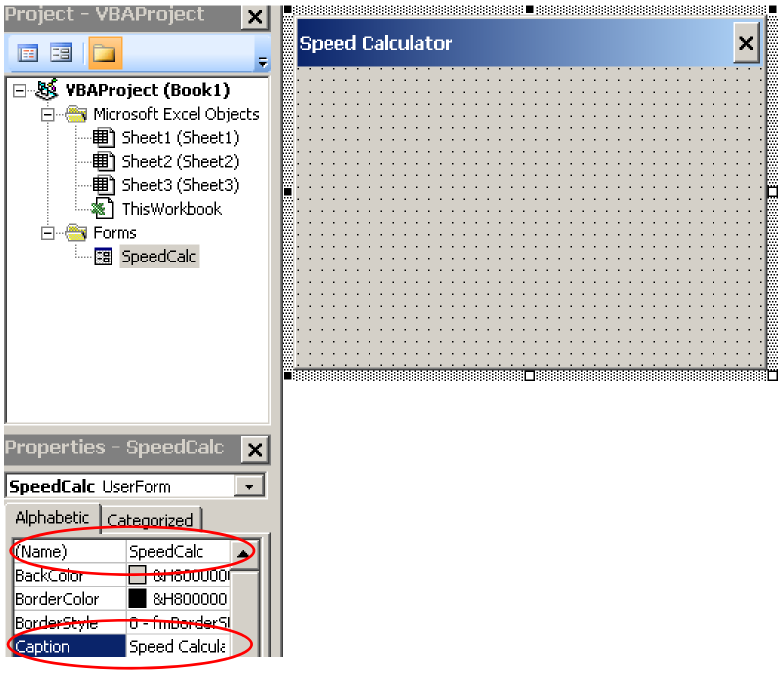
Alternatively, you also can create UserForms by right-clicking in the VBA project window and selecting **UserForm** instead of **Module**.

A blank UserForm named **UserForm1** will appear along with a Toolbox palette in the editor. The properties of the UserForm appear in the Properties Window and an icon for the UserForm appears in the Project Window.



(Name)	UserForm1
BackColor	&H8000000
BorderColor	■ &H800000
BorderStyle	0 - fmBorderSt
Caption	UserForm1
Cycle	0 - fmCycleAll
DrawBuffer	32000

We can customize the UserForm by changing its properties. This UserForm will be used to calculate the speed of a ball that is dropped from a building, so the UserForm's name and caption should reflect that. Change **(Name)** to **SpeedCalc** and **Caption** to **Speed Calculator**. Notice that the UserForm's name in the Project Window and the caption at the top of the UserForm changes.


Project - VBAProject

VBAProject (Book1)
Microsoft Excel Objects
Sheet1 (Sheet1)
Sheet2 (Sheet2)
Sheet3 (Sheet3)
ThisWorkbook
Forms
SpeedCalc

Speed Calculator

Properties - SpeedCalc

SpeedCalc UserForm

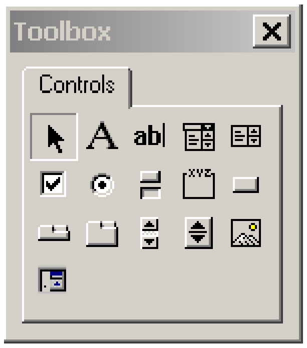
Alphabetic | Categorized

(Name)	SpeedCalc
BackColor	&H800000
BorderColor	&H800000
BorderStyle	0 - fmBorderSt
Caption	Speed Calcula

Note: No spaces are allowed for the UserForm’s (Name) field

Currently, our UserForm is blank. We need to add the following items:
- Areas where the user can enter values for the initial height of the ball, the elapsed time since the ball was released, and the gravitational acceleration.
- Labels that guide the user to where those three pieces of information should be entered.
- Buttons that will allow the user to execute the program and quit the program.

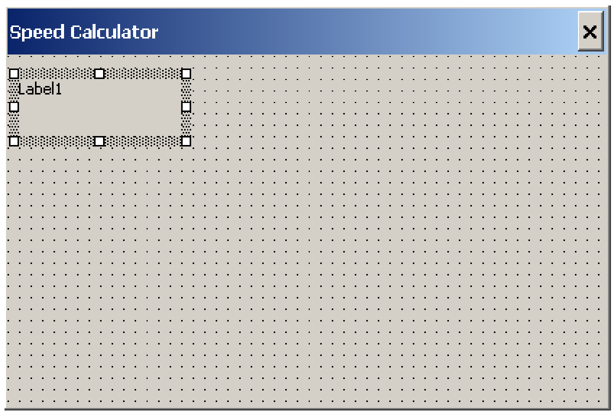
We can add these items, called **controls**, using the Toolbox palette.

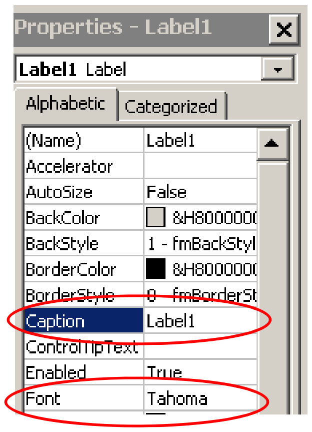


Some common controls are:
Label – Creates text labels that guide the user’s actions
TextBox – Creates a text box that allows the user to input numbers and strings
CommandButton – Creates a button that allows the user to execute commands
CheckBox – Creates a check box that allows the user to select/unselect an item
OptionButton – Creates a list of options that the user can select from. All options are displayed on the UserForm. This control usually is used when only a few options exist.
ComboBox – Creates a list of options that user can select from. Options are displayed by clicking on a drop-down menu. This control usually is used when many options exist.

Example: Using TextBoxes, CommandButtons, and Labels in UserForms

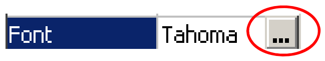
In this example, we will make a UserForm that will calculate the velocity of a ball that is dropped from a building. We first place a **Label** and **TextBox** to allow the user to enter the initial height of the ball. Choose Label from the Toolbox palette, then click-and-drag on the UserForm to create the Label.





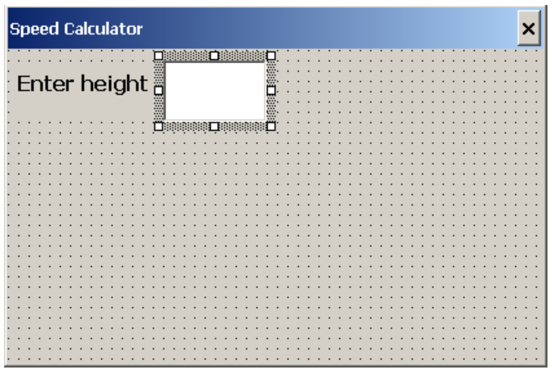
(Name)	Label1
Accelerator	
AutoSize	False
BackColor	☐ &H8000000
BackStyle	1 - fmBackStyl
BorderColor	■ &H8000000
BorderStyle	0 - fmBorderSt
Caption	Label1
ControlTipText	
Enabled	True
Font	Tahoma

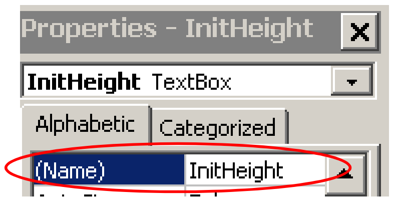
The properties of the Label are listed in the Properties Window. You can change the Label's text by clicking the Label or by clicking on the Caption property. The font can be changed by clicking on the **Font** property, then clicking on the button next to it.



Font	Tahoma	...

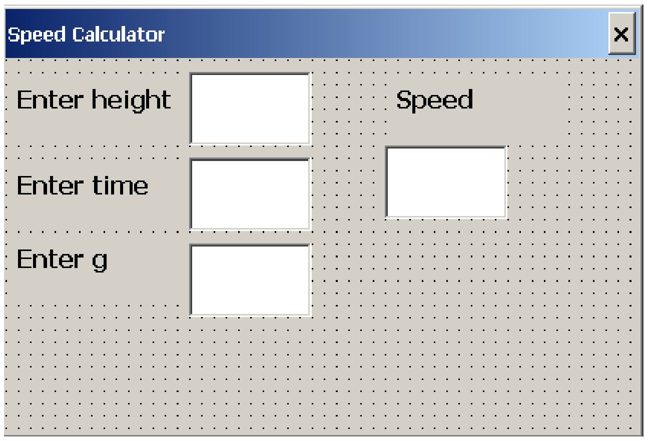
Create a TextBox next to the Label by selecting TextBox from the Toolbox palette, then click-and-drag on the UserForm. We will need to easily differentiate between these TextBox later, so enter a new, descriptive name for the TextBox in the (Name) field of the Properties Window.

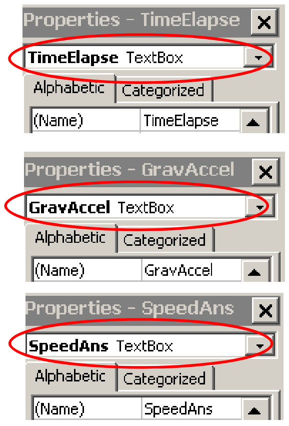




(Name)	InitHeight

The Labels and TextBoxes for the other two variables (time elapsed and gravitational acceleration) and the answer (the speed) will be similar. So we can copy and paste the Label and TextBox that are created three times using the Ctrl+c and Ctrl+v shortcut. Give the three new TextBoxes new, descriptive names.





Properties - TimeElapse
TimeElapse TextBox

(Name)	TimeElapse

Properties - GravAccel
GravAccel TextBox

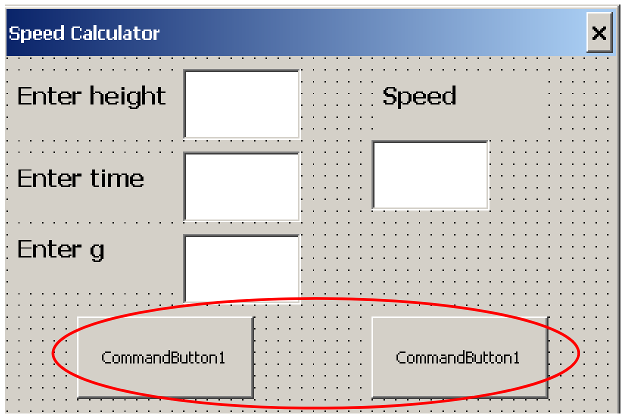
(Name)	GravAccel

Properties - SpeedAns
SpeedAns TextBox

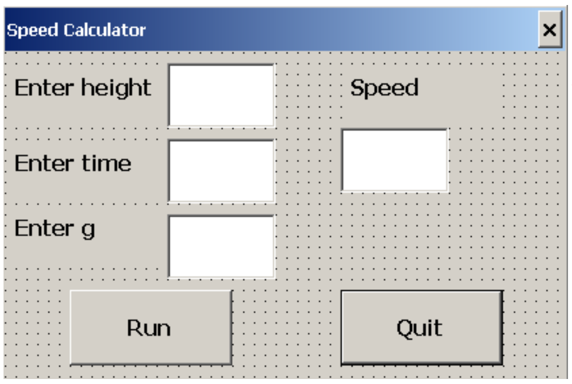
(Name)	SpeedAns

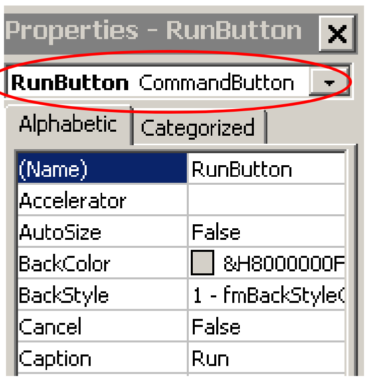
The TextBox next to **Enter time** is renamed **TimeElapse**, the TextBox next to **Enter g** is renamed **GravAccel**, and the TextBox next to **Speed** is renamed **SpeedAns**.

Next we add buttons to calculate the velocity and to close the UserForm when we are finished using it. Select **CommandButton** from the ToolBox palette and create two command buttons by clicking-and-dragging.



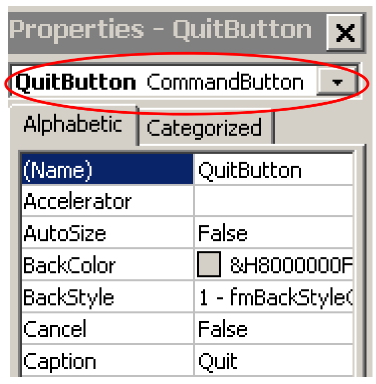
Change the Caption property of the left button to **Run** and the caption property of the right button to **Quit**. Also, change the (Name) property of the left button to **RunButton** and the (Name) property of the right button to **QuitButton**.





Properties - RunButton
RunButton CommandButton

(Name)	RunButton
Accelerator	
AutoSize	False
BackColor	☐ &H8000000F
BackStyle	1 - fmBackStyle(
Cancel	False
Caption	Run

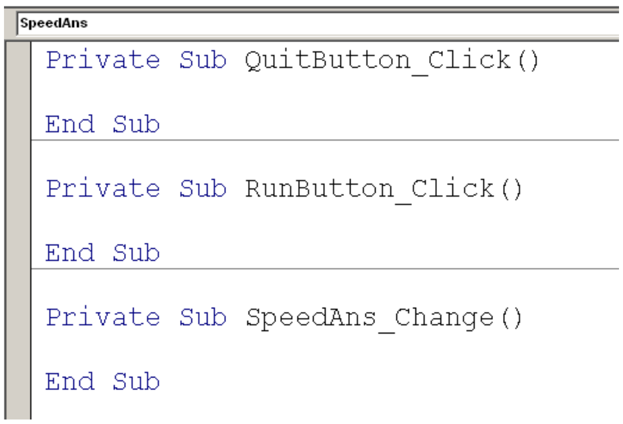


Properties - QuitButton
QuitButton CommandButton

(Name)	QuitButton
Accelerator	
AutoSize	False
BackColor	☐ &H8000000F
BackStyle	1 - fmBackStyle(
Cancel	False
Caption	Quit

Let's try running the UserForm by typing F5 to run the UserForm. Try clicking the Run and Quit buttons and enter text into the TextBoxes. Nothing happens… we have not told the UserForm what to do when these actions are taken. We need to provide code that will be executed when the CommandButtons are clicked.

Close the UserForm (click the X in the upper right corner) and go back to the UserForm editor. If you double-click on an item on the UserForm (Labels, TextBoxes, or command buttons) a code window will appear that will list a blank Sub procedure for each of item you clicked on. For example, if you click on the controls named QuitButton, RunButton, and SpeedAns.


SpeedAns

```
Private Sub QuitButton_Click()

End Sub

Private Sub RunButton_Click()

End Sub

Private Sub SpeedAns_Change()

End Sub
```

In the past, we have created Sub procedures that could be used in other modules if you call them (they were **Public** by default). A **Private Sub procedure** is limited to the module where it is written. Just leave the "Private" alone and you will be fine.

The **RunButton_Click( )** Sub procedure contains the code that will be executed when RunButton is clicked.

```
Private Sub RunButton_Click()
Dim vel As Double, g As Double, h0 As Double, t As Double, v0 as Double
h0 = InitHeight.Value
t = TimeElapse.Value
g = GravAccel.Value
v0 = 0                          ' The ball has no initial velocity since it is dropped from a building
' The equation for the height of a ball that is dropped from a building at time t is h0+v0*t+0.5*g*t^2
' The equation for the velocity of a ball that is dropped from a building at time t is v0+g*t

If (h0 + v0 * t + 0.5 * g * t ^ 2 > 0) Then     ' If the ball is still above ground at time t
  vel = v0 + g * t
Else                                            ' The ball has hit the ground at time t
  vel = 0
End If
SpeedAns.Value = vel                            ' Update the SpeedAns TextBox
End Sub
```

The values of **h**, **t**, and **g** are obtained from the text boxes with names **InitHeight**, **TimeElapse**, and **GravAccel**. Next, the code checks to see if the ball would have hit the ground by the time **t**. If the ball would not have hit the ground, the velocity is **v0 + g * t**. If the ball would have hit the ground, the velocity is zero (this program assumes that the ball does not bounce). Finally, the velocity is output to the **SpeedAns** TextBox.

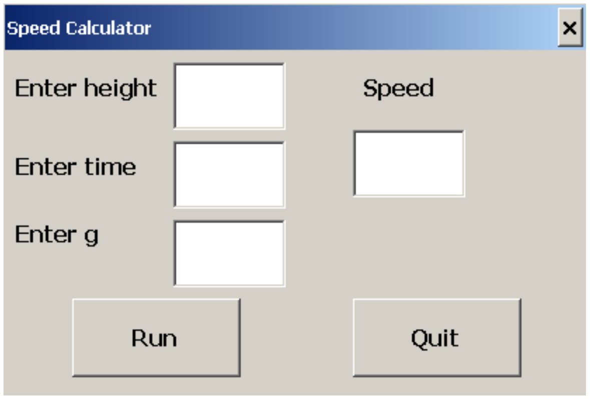
When **QuitButton** is clicked, we want the UserForm (named **SpeedCalc**) to disappear from the screen using the **Unload** statement.

```
Private Sub QuitButton_Click()
Unload SpeedCalc
End Sub
```

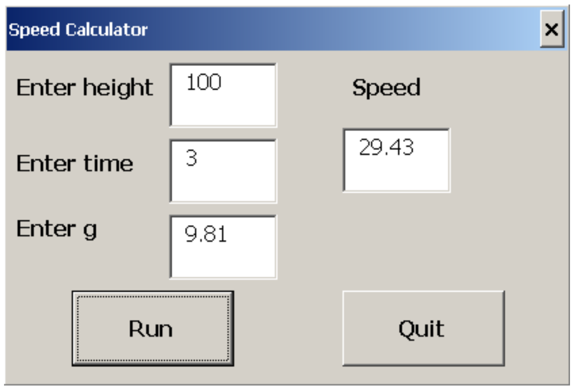
Now that we have created the UserForm, we need to create a macro that will show the UserForm to the user. Insert a new module and insert the following code,

```
Option Explicit
Sub velocity()
SpeedCalc.Show
End Sub
```

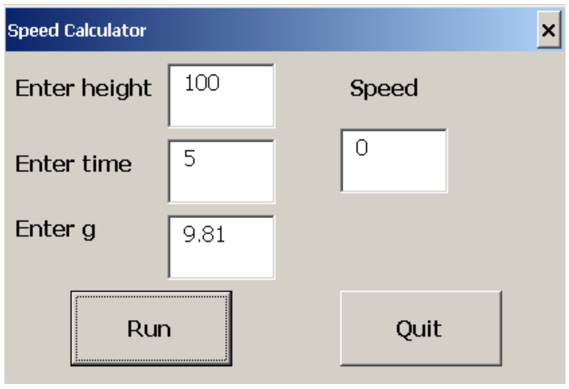
When the **velocity** macro is executed, the UserForm named SpeedCalc will appear.



Try plugging in values for **height**, **time**, and **g**, the click on the **Run** button.



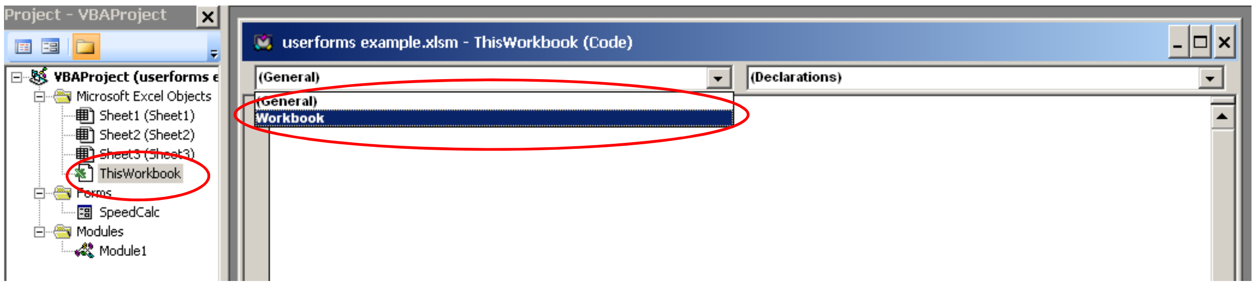
Enter height	100
Enter time	3
Enter g	9.81
Speed	29.43



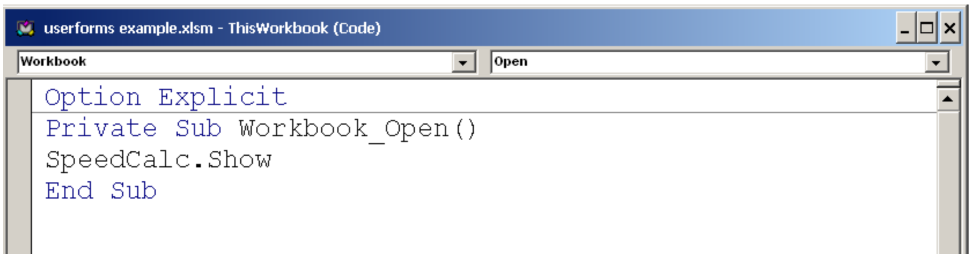
Enter height	100
Enter time	5
Enter g	9.81
Speed	0

After you are finished click on the Quit button and the UserForm disappears.

If you want the UserForm to automatically appear when you open the workbook, you can add an **event handler**. Click on **ThisWorkbook** item in the Project Window. Choose the **Workbook** option from the drop-down menu.



The **Workbook_Open( )** Sub procedure is executed every time the workbook is opened. We want the **SpeedCalc** UserForm to open when the workbook is opened, so type **SpeedCalc.Show** in the Sub procedure.



```
Option Explicit
Private Sub Workbook_Open()
SpeedCalc.Show
End Sub
```

Close and reopen the Excel workbook. The UserForm should open right away.

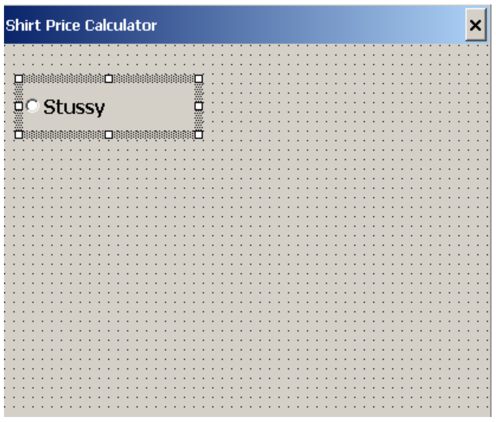
Example: Using OptionButtons and ComboBoxes in UserForms

The **OptionButton** and **ComboBox** controls are used when we want the user to select one option from a set of options.

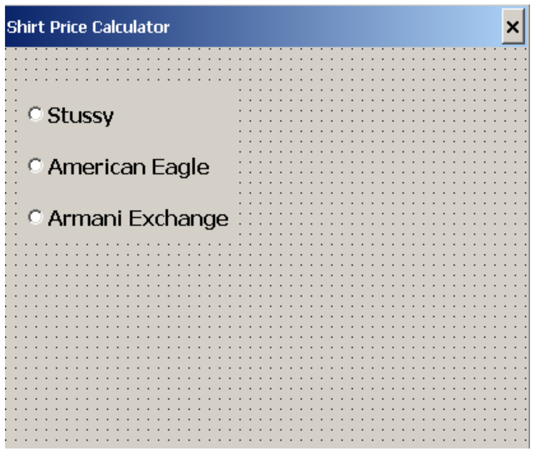
Let's create a UserForm that tells the user the cost of a shirt based on the shirt brand and size. There will be three brands and five sizes to choose from. The user will choose a shirt brand using an OptionButton control and a shirt size using a ComboBox control.

Create a new UserForm. Change the (Name) field to **ShirtPrice** and its Caption field to **Shirt Price Calculator**.

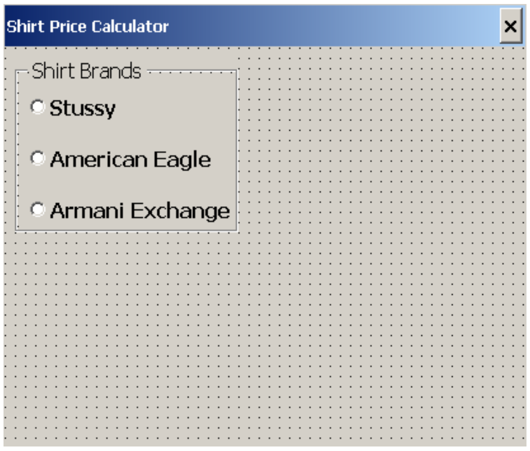
Add an OptionButton control to the UserForm. Change its (Name) field to **Option_Stussy** and its caption field to **Stussy**.



Add two more OptionButton controls to the UserForm by click on the first OptionButton, then typing Ctrl+c and Ctrl+v.
Change the (Name) and Caption fields of the second OptionButton to **Option_AE** and **American Eagle**.
Change the (Name) and Caption fields of the third OptionButton to **Option_AX** and **Armani Exchange**.

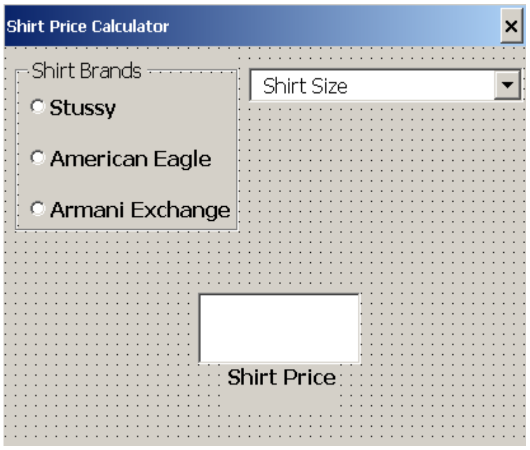


Frames allow you to combine similar Controls into a single container. This makes the UserForm look nicer and allows you to move many controls on the UserForm easily. You can place a frame around the three OptionButton controls by inserting a Frame control into the UserForm and dragging the OptionButton controls into the Frame control.



Now we want to add a **ComboBox** control for the shirt sizes. Click on the ComboBox icon in the Toolbox palette and drag out a ComboBox control on the UserForm. Change the (Name) field of the ComboBox control to **cbo_ShirtSize**.

Finally, we will need a TextBox control to display the cost of the shirt. Click on the TextBox icon in the Toolbox palette and drag out a TextBox control on the UserForm. Change the (Name) field of the TextBox control to **TotalCost**. Also, add a Label to tell the user that the TextBox contains the cost of the shirt.



Now that we have all the controls in place, we need to give instructions to the controls.

The cost of the shirt will depend on both the brand and the shirt size. There will be a cost associated with the brand and there will be an extra cost associated with the shirt size since larger shirts require more material. For UserForms that require input from more than one control to calculate the quantity of interest (the cost of the shirt) it often is convenient to make **Public** variables that are accessible to all procedures the code associated with a UserForm. Note: In general, I strongly discourage the use of Public variables except when creating UserForms. It is considered poor programming practice and is a bit risky for programs with multiple Sub and Function procedures because it is easy to lose track of which procedure last updated a variable.

Double-click on the UserForm and insert the following VBA code at the very top of the UserForm module.

```
Option Explicit
Public brandcost As Double, sizecost As Double, totcost As Double
```

Public variables are declared after Option Explicit outside of a Sub procedure. The Public variable **brandcost** stores the cost of shirt due to the brand, **sizecost** stores the cost of the shirt due to the size, and **totcost** stores the total cost of the shirt.

Now we will give options for the shirt size when the user clicks on the ComboBox control. Double-click on the UserForm and insert the following VBA code.

```
Private Sub cbo_ShirtSize_DropButtonClick( )
cbo_ShirtSize.List = Array("XS", "S", "M", "L", "XL")
End Sub
```

This Sub procedure is executed whenever the drop down menu is clicked by the user. Inside this Sub procedure the built-in function named **Array()** is used to populate the list of options for the user. There are five shirt sizes.

Add the following Sub procedure which will be executed whenever a new value for the ComboBox is selected. The new shirt size is obtained and the total cost of the shirt is updated.

```
Private Sub cbo_ShirtSize_Change()
Select Case (cbo_ShirtSize.Text)       'Access the shirt size
Case "XS"
  sizecost = 0
Case "S"
  sizecost = 2
Case "M"
  sizecost = 4
Case "L"
  sizecost = 5
Case "XL"
  sizecost = 6
End Select
totcost = sizecost + brandcost         'Update the total cost
TotalCost.Value = totcost              'Output the new total cost to the TotalCost TextBox
End Sub
```

Add the following three Sub procedures to the VBA code that will be executed whenever a new OptionButton is selected. The new **brandcost** will be updated, as well as the total cost of the shirt. If you double click on the OptionButtons, the first and last line of each Sub procedure will be created automatically for you.

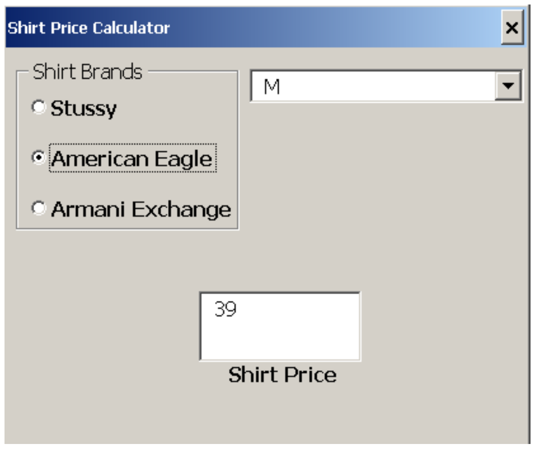
```
Private Sub Option_Stussy_Click()
brandcost = 25
totcost = sizecost + brandcost
TotalCost.Value = totcost
End Sub

Private Sub Option_AE_Click()
brandcost = 35
totcost = sizecost + brandcost
TotalCost.Value = totcost
End Sub

Private Sub Option_AX_Click()
brandcost = 65
totcost = sizecost + brandcost
TotalCost.Value = totcost
End Sub
```

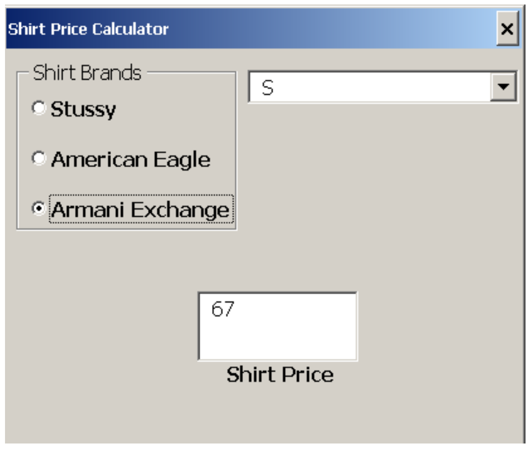
By using Public variables, we didn't have to worry about passing the values of **sizecost**, **brandcost**, and **totcost** to the various Sub procedures. Each Sub procedure knows the value of all three variables at all times. However, I want to emphasize (for the millionth time) that you usually should avoid using Public variables.

Our UserForm is now ready. Insert a new module and add the following code which loads the UserForm.

```
Option Explicit
Sub shirt( )
ShirtPrice.Show
End Sub
```

Create a button on a worksheet that runs the **shirt()** Sub procedure. When the **ShirtPrice** UserForm appears, choose some brand and size combinations in the custom dialogue box.





If you want the UserForm to be loaded when the workbook is opened, click on **ThisWorkbook** in the **Project Window**, select **Workbook** from the drop down menu, and type **ShirtPrice.Show** in the **Workbook_Open( )** Sub procedure.

```
Option Explicit
Private Sub Workbook_Open()
ShirtPrice.Show
End Sub
```

Close and reopen the Excel workbook. The **ShirtPrice** UserForm should open right away.

Topic 9: Arrays

Many programs require you to store and access a lot of similar data.

For example, say you are required to develop a program that will calculate the x- and y-position of a cannonball from 0 seconds to 1 second using a time step of 0.1 seconds. If you want to store this data, you would need to create 22 variables (11 times × 2 coordinates = 22 variables). This is a bit messy. Now imagine you wanted to store the cannonball's position every 0.0001 seconds. That would require a lot of variables.

Alternatively, we can create one container, called an array, to store the data. An array is similar to an Excel worksheet, with each value in an array (called an element) given a specific address.

You can only store values that have the same data type in an array. For example, you may not store both Strings and numerical data in the same array.

1-D arrays (vectors)

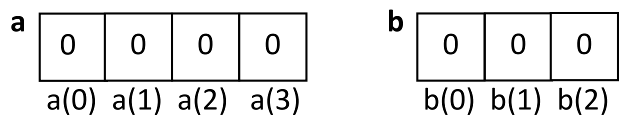
The simplest types of arrays are one-dimensional. You can picture them as one row of boxes with each box able to store a single value.

You set the number of elements in an array when declaring the array. For example,

Dim a(3) As Double, b(2) As Double

This declaration creates two arrays, **a** and **b**. By default, the element numbering system starts at 0 and ends at the value listed in the array declaration statement. Note: Many other languages also start the element numbering at 0, while others start the numbering at 1.

Here is how to picture arrays **a** and **b**.

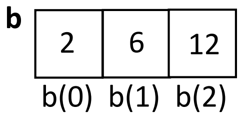


The numbers in parentheses are called the element's index or subscript number. By default, all elements in a numerical array are assigned the value 0.

The first array, called **a**, contains 4 elements. Four locations are reserved in the computer's memory for the array **a**.
The second array, called **b**, contains 3 elements. Three locations are reserved in the computer's memory for the array **b**.

1-D arrays require one number to refer to a specific element. You assign values to the array elements in the same manner as variables.

```
b(0) = 2                ' b(0) is assigned the value 2
b(1) = b(0) + 3         ' b(1) is assigned 2 + 3, or 5
b(1) = b(1) + 1         ' b(1) is assigned 5 + 1, or 6
i = 1                   ' i will be used as an index number
b(2) = b(i)*b(i-1)      ' b(2) is assigned b(1)*b(0), which is 6*2, or 12
                        ' It is common practice to use a variable for the index number when using loops
```



If you try to store a value in a non-existent array element,

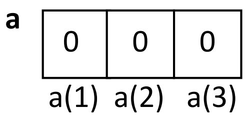
```
b(3) = 100
```

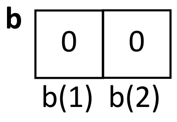
you would get a "Subscript out of range" error message.

If you wish, you can force the lowest array element to be 1 instead of 0 by placing the **Option Base 1** statement before the first Sub procedure in a module. I will be including **Option Base 1** in many examples.

```
Option Explicit
Option Base 1
Sub arrayfun( )
Dim a(3) As Double, b(2) As Double
```

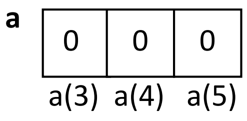
Now arrays **a** and **b** start with an index of 1 instead of 0. Notice that the size of each array has shrunk by one element. You can only store three values in **a** and two values in **b**.

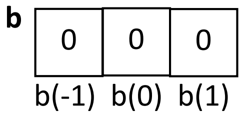




Alternatively, you can specify the lower and upper bound of the element numbers using the **To** keyword. This is not done often in practice.

```
Option Explicit
Sub arrayfun( )
Dim a(3 To 5) As Double, b(-1 To 1) As Double
```



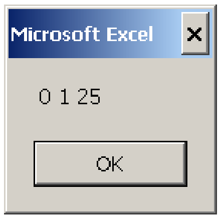


Loops and arrays naturally go together since loops will allow you to access all of an array's elements easily.

Example: Using loops and arrays

In this example, we will store the value of **i**^2 in each array element, where **i** is the subscript number. For example, in the 4th element we would store 4 ^ 2, or 16.

```
Option Explicit
Sub arrayfun()
Const N As Integer = 5
Dim a(N) As Long, i As Long              ' The array has 6 elements with subscripts 0, 1, 2, 3, 4, 5.
For i = 0 To N
  a(i) = i ^ 2              ' The loop will access a(0), then a(1), then a(2), …, and finally a(N)
Next i
MsgBox a(0) & " " & a(1) & " " & a(N)
End Sub
```



```
0 1 25
```

Notice how many times the size of the array (**N**) appears in the program. It is convenient to create a constant (such as **N**) for the upper bound of an array as it will allow you to update your entire code by making one small change.

Multidimensional Arrays (Matrices)

Although in practice 1-D and 2-D arrays will be adequate for 99%+ of everyday applications, VBA arrays can contain up to 60 dimensions. If you need to go use anything higher than a 3-D array, you likely are not coding in an efficient manner or are working on a very computationally taxing problem (in which case you probably should be using a language that is better suited for number-crunching such as Fortran or C).

2-D arrays can be created in the following manner,

```
Dim c(1, 2) As Integer
```

This statement creates a 2x3 (2 rows, 3 columns) array named **c** which stores integers. All elements are initialized to the value 0. The (row, column) address of each element is shown below.

c

0,0	0,1	0,2
1,0	1,1	1,2

We can assign values to each array element by specifying a row number and column number.

```
c(0,1) = 7                        ' 7 is assigned to element (0, 1)
c(1,2) = 10 – c(0,1)              ' 10 – 7 is assigned to element (1, 2)
```

The previous two statements assign values to elements **c(0,1)** and **c(1,2)**. Here is the updated array.

c

0	7	0
0	0	3

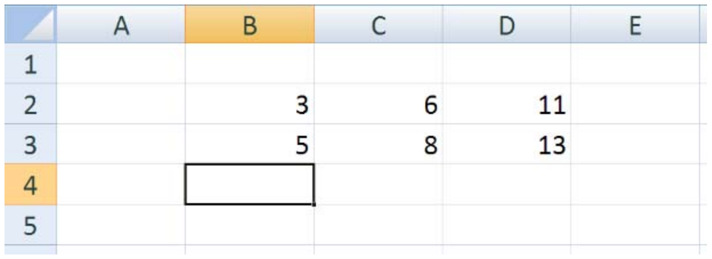
If you want to access each element in a 2-D array, you will need to have two nested loops – one loop for the row index and one loop for the column index. In general, if you have an N-dimensional array, you will need N loops.

Example: Accessing all elements in a 2-D array

In this example, we will first populate all elements in the array **c** with values, then we will output those values to a worksheet.

```
Option Explicit
Option Base 1
Sub arrayfun()
Const nrow As Integer = 2, ncol As Integer = 3
Dim c(nrow, ncol) As Integer
Dim i As Integer, j As Integer

' Populating all elements with values
For i = 1 To nrow                 ' For each row
  For j = 1 To ncol               ' we proceed through each column in that row
    c(i, j) = i * 2 + j ^ 2       ' This is some random equation I thought of
  Next j
Next i

' Outputting the values to a worksheet
Sheets("Sheet1").Select
Range("B2").Select
For i = 1 To nrow
  For j = 1 To ncol
    ActiveCell.Value = c(i, j)
    ActiveCell.Offset(0, 1).Select        ' We offset one column
  Next j
  ActiveCell.Offset(1, -ncol).Select    ' After the row is finished, go to the beginning of the next row
Next i
End Sub
```

After running the program, the following values should appear in Sheet1.



	A	B	C	D	E
1					
2		3	6	11	
3		5	8	13	
4					
5					

Passing arrays to Sub and Function procedures

Arrays can be passed to Sub procedures and Function procedures as arguments. When passing arrays, you only list the array name in the argument list.

As with variables passed down to procedures, we do not re-declare the arrays in the procedure. The array is assumed to have the same data type as in the calling procedure.

Example: Passing a 1-D array as an argument

In the following program a 1-D array is assigned values, then the average of those values is calculated.

```
Option Explicit
Option Base 1
Sub subby( )
Const n As Integer = 3
Dim A(n) As Double
A(1) = 3: A(2) = 10: A(3) = 2                       ' Some random values are assigned to A
MsgBox "The average of A is " & calcavg((A), (n))   ' A and n are passed by value to protect them
End Sub

Function calcavg(A, n) As Double
Dim i As Integer
calcavg = 0                 ' It is good programming practice to set summation entity to 0
For i = 1 To n
  calcavg = calcavg + A(i)  ' We access each element in A and add it to a running total
Next i
calcavg = calcavg / n       ' Calculating the average
End Function
```

As with variables passed down to procedures, we do not redeclare the arrays in the procedure. The array **A** is assumed to have the same data type as in the calling procedure (**subby**), which is Double.

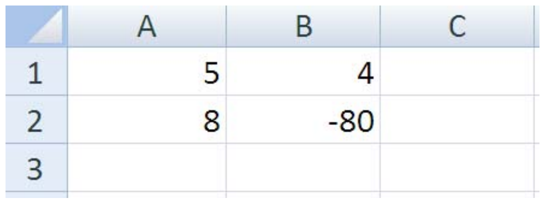
It is common practice to pass down the array size (**n**) along with the array.

Example: Passing 2-D arrays as arguments

The following program uses a Sub procedure that takes as input two 2-D arrays and calculates the sum of those two arrays. When adding two arrays, we add each element separately.

```
Option Explicit
Option Base 1
Sub subby()
Const nrow As Integer = 2, ncol As Integer = 2
Dim i As Integer, j As Integer
Dim A(nrow, ncol) As Double, B(nrow, ncol) As Double
Dim C(nrow, ncol) As Double
Dim avgval As Double, num As Double

' Assigning some random values to arrays A and B
A(1, 1) = 2: A(1, 2) = 3: A(2, 1) = 10: A(2, 2) = 20
B(1, 1) = 3: B(1, 2) = 1: B(2, 1) = -2: B(2, 2) = -100

Call addarrays((A), (B), (nrow), (ncol), C)     ' A, B, nrow, ncol are passed by value; C is passed by ref.
                                                ' The array C is now calculated
Sheets("Sheet1").Select                         ' Outputting C to Sheet1. Two loops are needed
Range("A1").Select
For i = 1 To nrow
  For j = 1 To ncol
    ActiveCell.Value = C(i, j)
    ActiveCell.Offset(0, 1).Select
  Next j
  ActiveCell.Offset(1, -ncol).Select
Next i
End Sub

Sub addarrays(X, Y, nr, nc, Z)                  ' Notice that we are using different names for the arrays
Dim i As Integer, j As Integer
For i = 1 To nr
  For j = 1 To nc
    Z(i, j) = X(i, j) + Y(i, j)                 ' The arrays are added element-by-element
  Next j
Next i
End Sub
```

After the macro is executed, the following information is output to the Excel workbook:



	A	B	C
1	5	4	
2	8	-80	
3			

Dynamic memory allocation: ReDim statement

Previously we assumed that arrays contain a fixed number of elements. But it is possible to create arrays with a variable number of elements as well (sometimes called "dynamic" arrays). When creating dynamic arrays, leave the number of elements blank. For example, the following code will create a dynamic array named **pig**,

```
Option Explicit
Option Base 1
Sub subby( )
Dim pig( ) As Double
```

The array **pig** is given a Double data type, but the number of elements is not determined yet. This is accomplished using the **ReDim** statement.

```
ReDim pig(3) As Double
```

Now, **pig** has three elements which can store information. Let's assign values to those three elements.

```
pig(1) = 5: pig(2) = 3: pig(3) = -1
```

pig	5	3	-1
	(1)	(2)	(3)

We can change the dimensions of pig by using the ReDim statement again,

```
ReDim Preserve pig(5) As Double
```

pig	5	3	-1	0	0
	(1)	(2)	(3)	(4)	(5)

Now **pig** has five elements. The new elements (4 and 5) are given a default value of 0. The **Preserve** keyword keeps the previous values in the array intact. If **Preserve** were omitted, elements 1, 2, and 3 would be reset to 0.

If **pig** is redimensionalized to a smaller number of elements,

```
ReDim Preserve pig(2) As Double
```

the values stored in elements 3, 4 and 5 are lost forever, even if **pig** is redimensionalized back to 5 elements later in the program.

Another way to create a dynamic array is to declare the array using ReDim.

```
Option Explicit
Option Base 1
Sub subby( )
ReDim f(2) As Double                ' Creates a dynamic array with 2 elements
```

We can change the size later using another ReDim statement.

```
ReDim Preserve f(10) As Double      ' Redimensionalizes f to 10 elements
```

Records

Arrays only allow you to store data with the same data type (only **Double**, only **String**, etc.). Records allow you to store data with different data types in the same container.

I won't be showing you how to make Records, but I wanted you to be aware of them.

Final thoughts for the course

In the past 60 years, computers have become smaller, faster, cheaper, more numerous, and more user-friendly. In this relatively short period of human history, we have gone from a hand-full of people using a few giant metal behemoths that weighed many tons, to almost everyone having powerful cell phones that fit comfortably in their pocket. The trend of smaller, faster, cheaper, more numerous, and more user-friendly computers likely will continue well into the future.

HOWEVER, the basic building blocks of computer programming – variables, arrays, data types, If structures, loops, Functions procedures, and other topics you learned about in this course – have existed throughout the age of computers and will continue to remain a vital part of programming. Every high-level computer language that is used today (Fortran, C, C++, Java, MATLAB, Python, Mathematica, VBA, etc.) still contains these basic building blocks.

This 100 page booklet was designed to give you a solid foundation in computer programming. It certainly does not cover every topic in VBA, but if you understand all the concepts discussed in the booklet, you definitely have the programming tools necessary to create some really interesting and amazing programs. Good luck!

Additional resources

If you would like to purchase more advanced books on VBA, here are two suggestions:
(1) **Excel 2013 Power Programming with VBA** by John Walkenbach
  This book is quite thick and packed with information.
(2) **Excel VBA Programming for Dummies (2nd edition)** by John Walkenbach
  This book is not as thick and more approachable to the beginner.

Note: I have no affiliation with the author or publisher. I just like his books.

Detailed information about all VBA commands can be found on the Microsoft Developer Network (MSDN) website. Here is the MSDN page for VBA in Office 2013:
https://msdn.microsoft.com/en-us/library/office/gg264383.aspx

Most of the concepts on that also apply to other versions of Office as well.