

# OneNote Intro

October-25-11  
8:55 AM

## Introduction

To help you organize your work this semester, notes for this class will be distributed in a number of ways. One way will be to use this application: *OneNote 2010*.

OneNote is part of the Office 2010 suite of applications and is available as part of the Student Teacher edition. If you don't have it, and if you have access to the MSDNAA website, you can download a copy.

OneNote provides a large number of benefits for students and instructors alike. Things like searching the notes for a particular topic (even if the text is part of an image); organization using notebooks, sections, and pages; and adding tags to content so that you can sort by those tags (for example, labeling all the definitions in the notebook and then seeing a compacted list so you can pick the one you want).

The purpose of this section (*Using OneNote in Class*) is to bring you up to speed with OneNote so that you can start to use it effectively to help you through your classes.

*Using OneNote in Class* is tailored to help you get going as soon as possible. It contains material specifically relevant to class work and is a simple getting started explanation. It contains all the information to use OneNote in the classroom.

Get yourself a flash drive for use in all your classes and keep a copy of your notebooks on that drive. That way you'll always have the notebook with you.

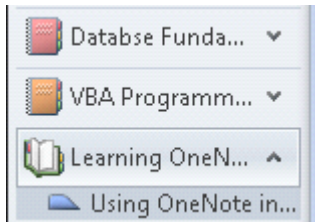
Click on the [OneNote's Organization](#) page in the pane to your right to continue.

# OneNote's Organization

October-26-11  
7:57 AM

## Notebooks

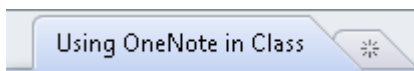
A *notebook* is the first major organization level. It acts like a binder. A notebook is used for each major subject that you will be keeping notes for. You can see the notebooks you have on the left.



You can have a notebook for every class. A notebook is simply a collection of the materials for a particular subject. Within the notebook you will compile and organize whatever materials you feel necessary for your understanding of the topic. OneNote opens, by default, all notebooks every time you start the application.

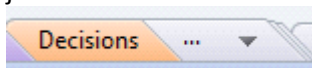
## Sections

The next layer of organization are *sections*. If the notebook is the binder, think of sections as the dividers used to subdivide the material. Sections are used to hold information regarding a single topic about a subject. Across the top of a notebook you will see a set of tabs.

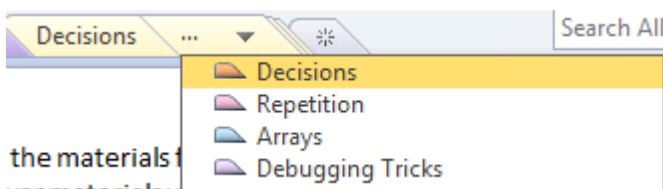


A tab represents a section. Each section has a title (the words listed on the tab itself). Each section can hold a number of pages.

Sometimes there will be far too many sections to display all of them as tabs across the top. OneNote will group them together as a single tab with three periods and a downward facing arrow. It will be just to the left of the New Section tab.



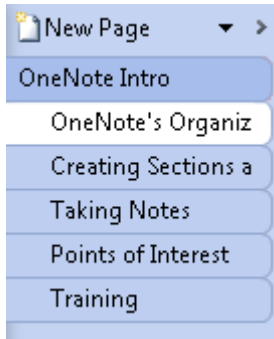
To see tabs that aren't visible, click the arrow at the end of the tab list and select the section you want to see.



## Pages

Within sections you can have as many *pages* as you like. Think of pages as the individual pieces of paper inside your binder. Pages can be used to further refine your organization. Perhaps information from each class about a particular topic can have its own page. You can see what pages are available

on the right side of the screen.



We'll look at how to create notebooks, sections, and pages on the next page.

## Moving Around

To jump to a notebook simply click on its name in the right hand pane.

Once in a notebook to see a section click on the appropriate tab.

And to see a particular page within a section click on its name on the right hand side of the screen.

Now, on the right hand side of the screen, click on the [Creating Notebooks, Sections, and Pages](#) page to continue.

# Creating Notebooks, Sections, and Pages

October-25-11  
9:32 AM

During the course you will be given notes in a OneNote workbook. You will read through the material and participate in the class(es) related to those notes. To ensure that you don't accidentally delete a portion of those notes or change them accidentally, it is wise to create your own notebooks, sections, and pages to take notes in.

★ *A note about saving notebooks:* OneNote saves automatically any time you change sections or perform other similar operations or at pre-arranged times.

## New Notebooks

At some point it might be necessary to create your own notebooks. To do so follow these steps:

1. Click File to enter Backstage view.
2. Click New.
3. My suggestion is to create the workbook on your computer so click *My Computer*.
4. Enter a name for the notebook.
5. And finally tell OneNote where to save/create the notebook. My suggestion is to put it on the desktop or directly to your flash drive.

★ *A note about closing notebooks:* when a note book is closed it is removed from the notebook pane. Try not to remove a notebook unless you are positive you don't need it anymore. Closed notebooks are not searched and you can't work with them unless re-opened.

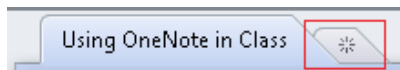
Double click the image below for a visual representation of the above steps.



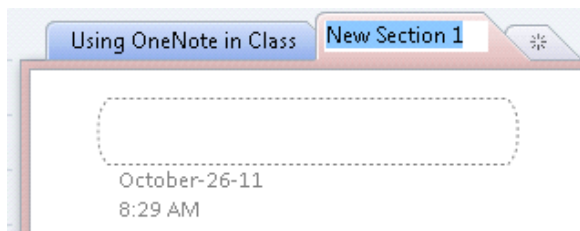
CreatingNotebooks

## New Sections

To add a new section click the *Create a New Section* tab. It's the small tab at the end of the section tabs at the top of the notebook. It's highlighted in red in the image below.



To change the name of the new tab simply double click the tab and type the new name.



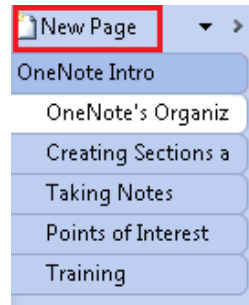
Now you have a blank section to work in. You can start taking notes here, or you can continue to organize your material.

## New Pages

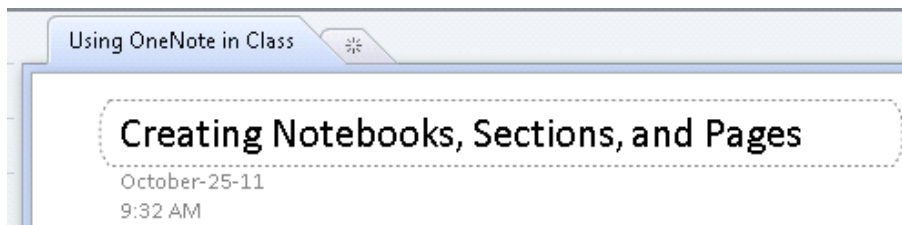
Within a section you can create new pages which can be used to sort your notes further. Each section contains a single page to start with. You can add other pages to further organize your material.

To create a page:

1. Make sure your section is selected (you can see it's content in the content window).
2. On the right of the screen, at the top of the page list pane, click the *New Page* button.
3. You could also simply right click in the pane and choose *New Page*.



At the top of any new page there is an ellipse with a dotted outline. Whatever you type into that ellipse will be used as the title for the page and appear in the name list to the right.



Click the [Taking Notes](#) page to continue.

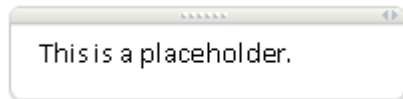
# Taking Notes

October-25-11  
9:51 AM

Now that you have a new notebook, section, and pages to work in, it's time to start taking notes.

To do so, simply click in any available empty spot on a page, and start typing. It's that simple.

You'll see a place holder appear, and your content will be inside the placeholder.



Working inside this placeholder is much like working in a Word document. You can format your work the same way.

Like the rest of the Office 2010 suite of applications, you can cut, paste, copy, take screenshots (new to 2010), add recorded video or voice, and attach or embed files. Check the *Insert* menu to see all the options.

Once on the page you can move and rearrange all the content to better help you understand and organize it as well by simply clicking and dragging the place holders.

Select the [Points of Interest](#) page to continue.

# Points of Interest

October-25-11  
9:58 AM

## Consolidating Distributed Sections

Usually you will be given a notebook at the start of classes. Rarely will that notebook be complete. Sometimes you will be given additional materials, supplements, or new chapters using OneNote. A single chapter could be distributed in its own notebook. To make things easier on yourself, you'll want to consolidate those chapters into a single notebook or a master notebook.

This is a simple process as detailed here.

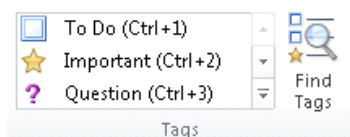
1. Make sure both notebooks (the distributed one and your main notebook) are open in OneNote.
2. In the notebook pane, expand them both so you can see the different sections.
3. Click and drag the section you want to move from the distributed work into the main workbook. You'll see a dark line appear. Where the dark line is, if you let go of the mouse, the section will be moved to that location.
4. Right click on the distributed notebook and choose *close*.

It's that simple.

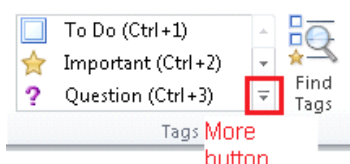
## Tagging

In OneNote you have the ability to tag certain items so that you can be reminded about them and locate them faster when you want them.

To see a list of available tags open the *Home* tab and look for the *Tags* pane.



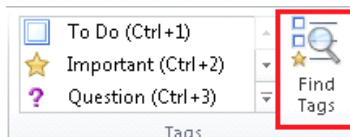
Click the More button to see the complete list.



This is a list of the tags currently available for use. You can define your own as well, but this list is fairly complete for what you might need.

Simply highlight the item (word, phrase, image, or whatever) you want to tag and then select the appropriate tag from the list. For example, the titles on this page have been tagged as *remember for later*.

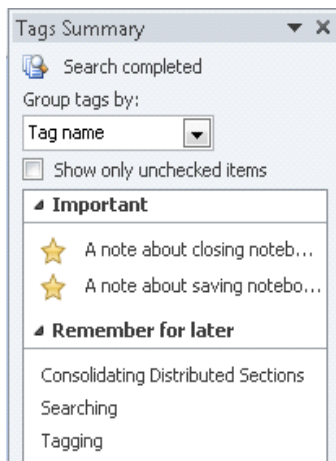
When you need to find something that you have tagged, simply click the *Home* tab and in the *Tags* pane click the *Find Tags Button*.



This opens the *Tags Summary* pane on the right. You will see a sorted list of items by the tag's name. Click on the one you want and you're there.

Scroll down for more.

Scroll up or down for more.

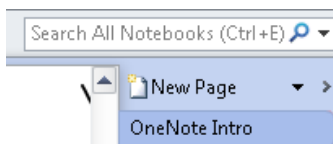


Scroll down for more.

## Searching

One of the strengths of OneNote is its ability to search through any open notebook (any notebook in the pane on the left).

The search box is located just above the page list pane.



Start typing. As you type you'll notice that OneNote actually starts to refine the list for you. Once you see what you're looking for click it and you'll be taken to that section/page.

Now, click the [Training](#) page to the right for some additional resources.



# Training

October-25-11  
10:42 AM

Here I've tried to collect some useful links to help you transition to using OneNote for this class. If you are having trouble acclimating to this application come and see me or review these training modules.

[Basics of OneNote](#) - this link details the basics of getting started with OneNote.

- What is OneNote?
- Using Templates.
- Create new workbook.
- Saving.
- Inserting a new page.
- Typing or writing notes.
- Printing

[Make the Switch](#) - this link gives you some hands on practice with One Note.  
It contains a practice session to get used to the environment.

[Keyboard Hotkeys](#) - this link provides an exhaustive list of all possible keyboard shortcuts for use in OneNote.

# Basic Programming Concepts

October-28-11  
8:36 PM

Essentially programming is listing the steps a computer must follow to complete a task, much like writing a set of instructions for anything else. As the programmer you have to decide the order those steps have to fall in. Remember that computer's are stupid. If your steps are in the wrong order you're going to get an error in your program. Always remember to plan first and code second.

## Programming Styles

Programming comes in many styles. We'll focus on the following three types:

1. Procedural.
2. Event based.
3. Object Oriented.

Procedural programming is a style of programming where the programmer lists the steps the computer needs to take in the correct order that they need to be executed. In essence the program is started, it runs, and it ends in the order it's written.

Event based programming is where the actions taken by the computer are based on actions or events that occur. For example, a program will wait until a button is pressed then act or until a sensor records a certain temperature level. In essence the program is started and it waits. It keeps running until a certain action or event happens.

Object oriented programming is based around the creation of objects (programming constructs built of code snippets). These objects provide a base set of code that can be reused from program to program as well as being added to and improved.

We will be focused on Event-based programming for the rest of this class. We'll be using Excel's built in programming language called *Visual Basic for Applications (VBA)*. We will be using VBA to allow you to create new applications in Excel and extend Excel's capabilities.

## Basic Programming Flow

Most applications follow a simple set of steps:

1. Start up -> There might be some things that we have to get ready before we can let the user use our program. So, during start up we prep our program for use.
2. Accept Input -> since we're focused on event-based programming this is the stage where an event occurs. The user enters a number or presses a button.
3. Process Data -> The computer takes that data and then does something with it.
4. Output response -> We have to show the user the results of their input or simply show them some result.
5. Stop program -> and finally we will need to clean up and shut down the application.

These are the general steps in any program and we will work through how to set up each stage focusing on steps 2, 3, and 4.

## Integrated Development Environments

To make their lives easier, programmers have developed programs that will assist them in writing their code. These *Integrated Development Environments (or IDEs)* give us a large number of tools that we can use to ease our programming.

The VBA editor built into Excel is one such IDE. We will be looking at and using a lot of the tools it provides to help us in our programming.

## The Compiler

High level programming languages (like VBA, C#, C++, etc.) use commands that are written in an English-like language. Unfortunately, your computer doesn't understand them. Computers only understand 0's and 1's. Therefore our high level languages then have to be translated into a low level language. Low level languages are those that the computer can understand without any further translation. Machine language is the prime example. Machine language is a programming language written completely in 1's and 0's.

The program that does the translation is called the compiler. The compiler takes our high level language and converts it into the low level language the computer can understand.

## Errors

Okay, it's safe to say that we've all had a program that didn't work. As a programmer it's your job to make sure the program works well before you let anyone use it. Errors can be dangerous, even damaging to a computer. You need to make sure that your programs are error free.

There are two types of errors that you will run into:

1. Syntax Errors
2. Logic Errors

### Syntax errors

These are the easiest to identify. If you have a syntax error your program simply won't run. All programming languages have a very specific way that all commands must be written. This is known as the command's syntax. If you don't have the correct syntax, or structure to the command, the compiler doesn't know how to translate that line of code and it simply gives up. It will tell you that it can't understand the line of code and wait for you to fix it.

### Logic Errors

This type of error, on the other hand, can be extremely difficult to identify or even realize that one exists.

If you have this type of error in your program, your program will still execute. It seems to run fine. The problem is that the results of the program are incorrect. For example, if you leave a set of brackets out of an equation therefore changing the order of operations.

Sometimes you won't realize one of these exists until a user discovers it. Planning and testing of your application will help you avoid these types of errors.

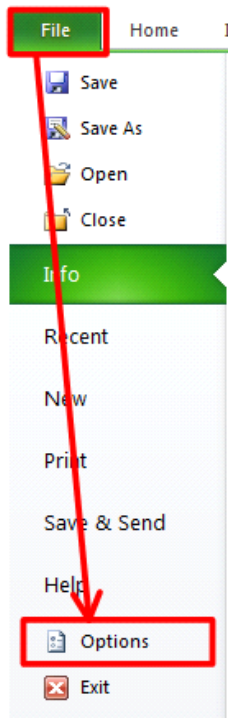
Now let's take a closer look at VBA itself. On the right click the page named [The Developer Tab](#).

# The Developer Tab

October-28-11  
8:54 PM

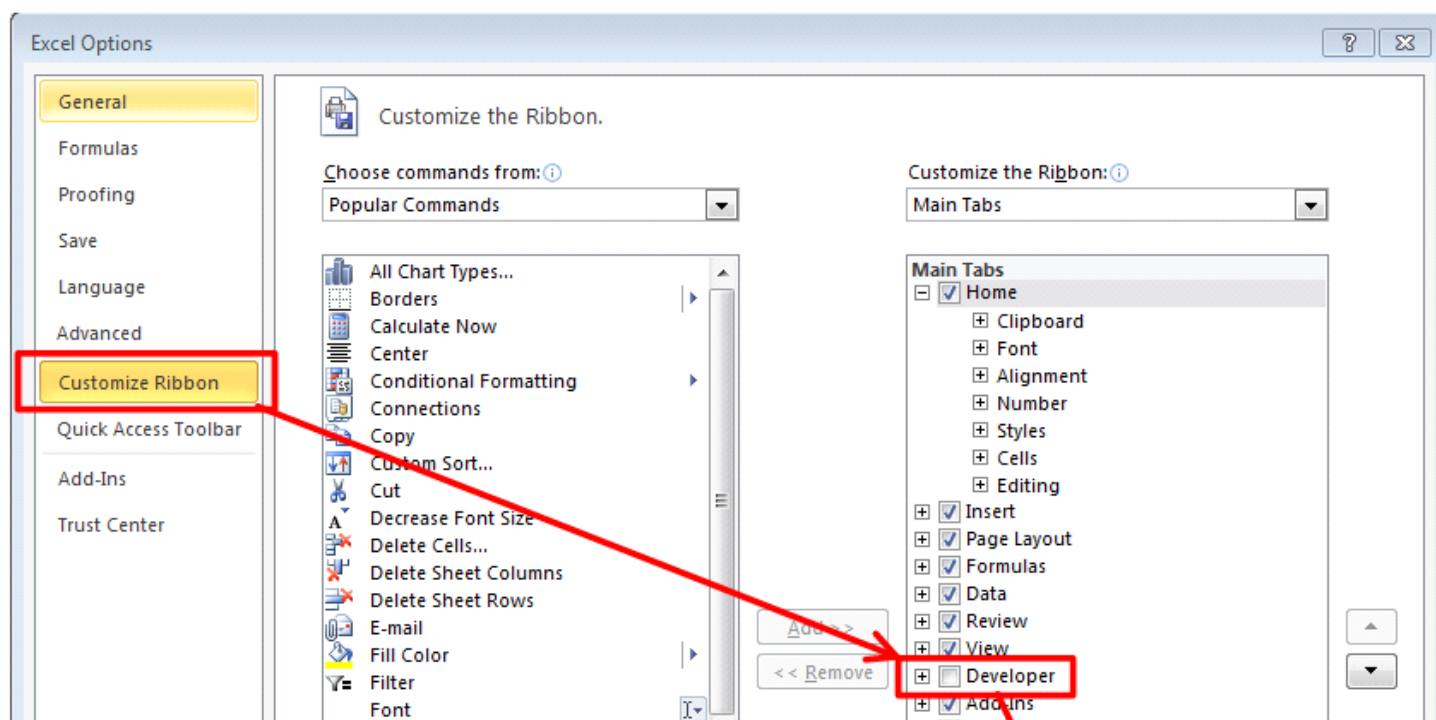
For our IDE we'll be using Excel's Visual Basic editor.

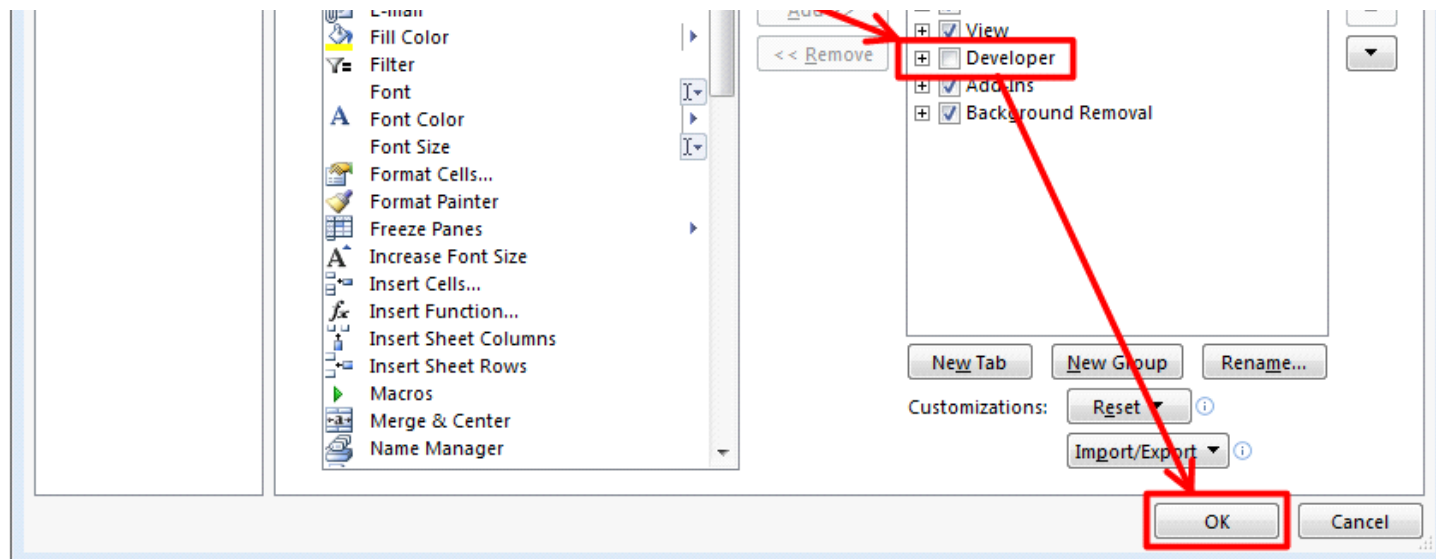
Before we can start programming we need to turn on the tab that will allow us access to the VBA options. We need to activate the Developer tab so it shows on the ribbon.



To do so:

1. Click File to enter backstage view.
2. Click Options.
3. In the dialog that opens click Customize Ribbon on the left.
4. Finally, on the right hand side you'll see a section titled Main tabs. Put a check mark beside the Developer option.
5. Click OK.
6. That should turn on the Developer tab at the top of your Excel window.



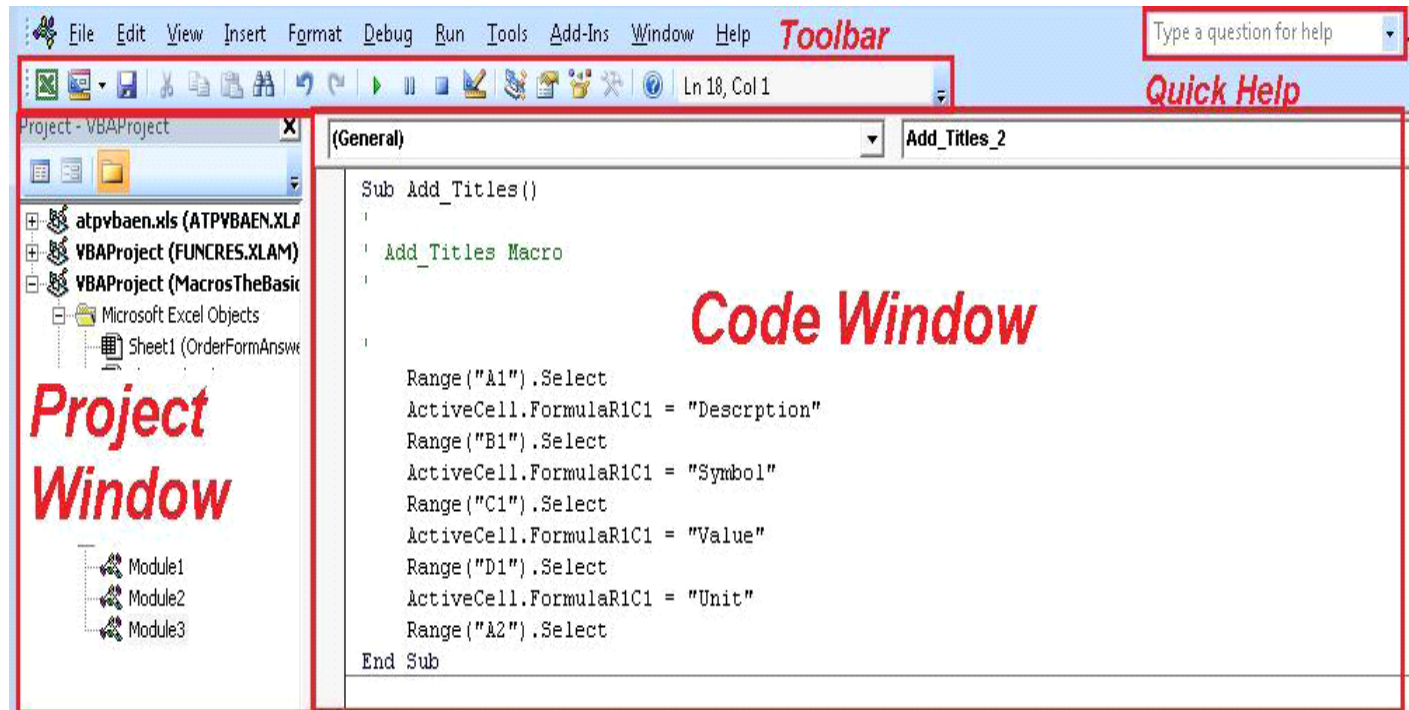


Now let's take a look at the VBA window itself. Click the [VBA Window](#) page to the right to continue.

# VBA Window

October-28-11  
8:29 PM

On the Developer tab you click the VBA button to open the VBA editor. You will see a screen similar to what's below.



There are four main areas that we'll concern ourselves with:

1. The code window – in this window you will see and write all of the code that we'll use in this class.
2. The project window – in this window you will see all the different objects that make up your project.
3. The toolbar – it provides quick access to some of the more common functions that you will use.
4. Quick help – type your question into the box and it will provide you with some feedback though be warned you may have to dig a little to find the answer you're looking for.

As stated above, the code window is going to be the one where we do most of our work. The code window does provide us with one very valuable tool: intelli-sense technology. This technology tries to anticipate what you are typing and gives you a list of items to select from to complete your code. You can simply select from the list to finish off what you are doing. If you don't see what you expect to see, then it might be an indication that something is wrong. It also provides colour coding so you can tell a bit about your code simply from looking at it. We'll discuss this more a bit later.

The project window lists all the objects that are a part of your workbook. You'll see an entry for each worksheet and a special object called a module. A module is a store house for pieces of code that can be transferred between workbooks if necessary. A module also makes code available between worksheets in the same workbook.

The toolbar is nothing really that exciting. I point it out solely for the fact that you will use certain buttons on it. You should be familiar with most of the buttons and those that you aren't familiar with we'll discuss as we go.

And lastly the quick help box. If you have a problem you should get used to researching the answers to problems. I warn you now that you may have to dig to get the correct answer that you're looking for.

Now, let's take a look at how we divide up our code in Visual Basic. Click the [VBA Structural Syntax](#) page on the right.

# VBA Structural Syntax

October-28-11  
8:29 PM

Programming languages are highly structured and you must follow certain basic rules when you write code. As stated in an earlier part of this document, this is known as the syntax of that language. The code in VBA, which appears in our code window, has a very specific syntax that you need to follow.

In VBA, to make things easier for us, we break our code into smaller blocks. We give each block a name so we can identify it when we need to run it. Each block contains lines of code that perform an action. The syntax for these blocks never changes and all your code must be contained inside one of these blocks. If you do not place your code in a block, then you will receive a syntax error, and if you remember, if you have a syntax error, your program will not run.

These blocks are created so that when a user performs an action, the program knows what line(s) of code to execute. We tie these blocks of code to actions (for example buttons) that the user can perform in our programs.

Let's look at an example using a macro. This is the code that was recorded:



```
Sub Add_Titles()  
    Add_Titles Macro  
  
    Range("A1").Select  
    ActiveCell.FormulaR1C1 = "Description"  
    Range("B1").Select  
    ActiveCell.FormulaR1C1 = "Symbol"  
    Range("C1").Select  
    ActiveCell.FormulaR1C1 = "Value"  
    Range("D1").Select  
    ActiveCell.FormulaR1C1 = "Unit"  
    Range("A2").Select  
End Sub
```

All of our code must be written into blocks. These blocks are called *sub-procedures*. Each sub-procedure begins with the keyword *Sub* and ends with the words *End Sub*. We indicate the name of our block in the first line beside the keyword *Sub*. All the code must come between these lines.

In this case, when we recorded the macro, Excel wrote the sub-procedure for us. We could just as easily have typed everything in by hand. As long as you start with *Sub <a\_name>()* and end it with *End Sub* everything will be fine. Note: The brackets that follow the name are important and must be included.

Let's take a look at your first line of code next. Click [White Space and Comments](#) on the right to continue.

# White Space and Comments

October-28-11  
8:30 PM

## White Space

In VB white space, all that empty area where there is no text, has no bearing on your code. Use it to your advantage. Add empty lines to segment your code into smaller blocks. Use the tab key to indent lines of code to make things more readable. The worst thing to have to do is debug someone's code when it's all pressed up against the left hand margin. It's almost impossible.

- ! You will indent your code so that I can read it. Indent lines so that you and I can see what lines of code belong together. You will use empty lines to segment your code for readability.

## Comments

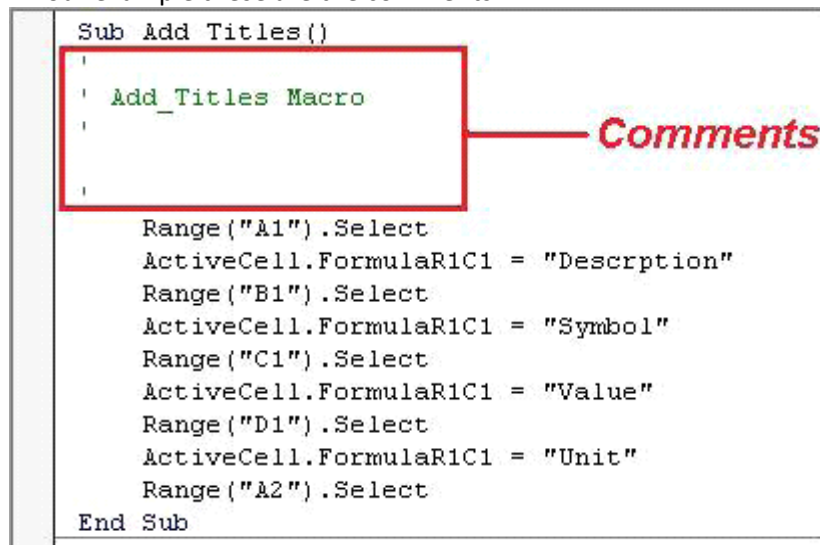
Once we have our basic structure in place we can start coding. The first command you will learn is the easiest of all; is the least used; and probably the most important, especially for beginning programmers.

A comment is a line of code that doesn't really do anything. Comments are used to provide the programmer and anyone else who is looking at the code with a reminder of what is happening. Comments are not executed when the code is run and can only be viewed in the code window. A user will have no idea that comments even exist.

So, if they don't do anything, why do we have to use them? Well, firstly, it gives you a way to add additional information to your code. Imagine you write a complex program. Two years later you're asked to modify the code. You'll probably have forgotten a lot of what you did. Comments can be used to help you remember what you did. The more likely scenario is that someone else will be the one editing your program. Or you will have to edit someone else's program. Comments would be a great help to that other person or to you.

- ! Also, as a requirement for this course you will place your name, the date, the course code, and the exercise number for everything you do in your code using comments.

To identify comments in your code you start the comment with an apostrophe – ' . The VBA editor's intelligence technology will colour all comments green by default. Each line of a comment must start with an apostrophe. In our example these are the comments:



```
Sub Add Titles()  
    ' Add_Titles Macro  
    '   
    Range("A1").Select  
    ActiveCell.FormulaR1C1 = "Description"  
    Range("B1").Select  
    ActiveCell.FormulaR1C1 = "Symbol"  
    Range("C1").Select  
    ActiveCell.FormulaR1C1 = "Value"  
    Range("D1").Select  
    ActiveCell.FormulaR1C1 = "Unit"  
    Range("A2").Select  
End Sub
```



Now, let's start programming. Click on the [Working with Data](#) tab at the top of the widow.

# Conventions

October-28-11  
9:37 PM

In every program you are going to need data to work with. We will need to look at how to get the data, how to store the data, how to process it, and finally how to return the data and results.

In this section we're going to look at a few of the commands that will allow us to accomplish these tasks.

A few notes before we move on about the formatting we'll be using in this document:

1. All commands will appear in a **dark blue** coloured font. Anything in this colour must appear in your code as it appears in this document.
2. All programmer replaceable parts or optional parts will appear in *red italics*. These portions will be replaced with values supplied by you if you need them.

All right one more thing before we start looking at actual code. Click the [Dot Notation](#) page to the right.

# Dot Notation

October-28-11  
9:39 PM

Before we move on to some more exciting code, we need to talk briefly about VB's dot- notation.

In VB we will be working with what are known as objects. An object is, in simple terms, any portion of the Excel window: a sheet, a cell, a button, etc. Every object has a series of predefined functions that affect either its characteristics or what the object can do.

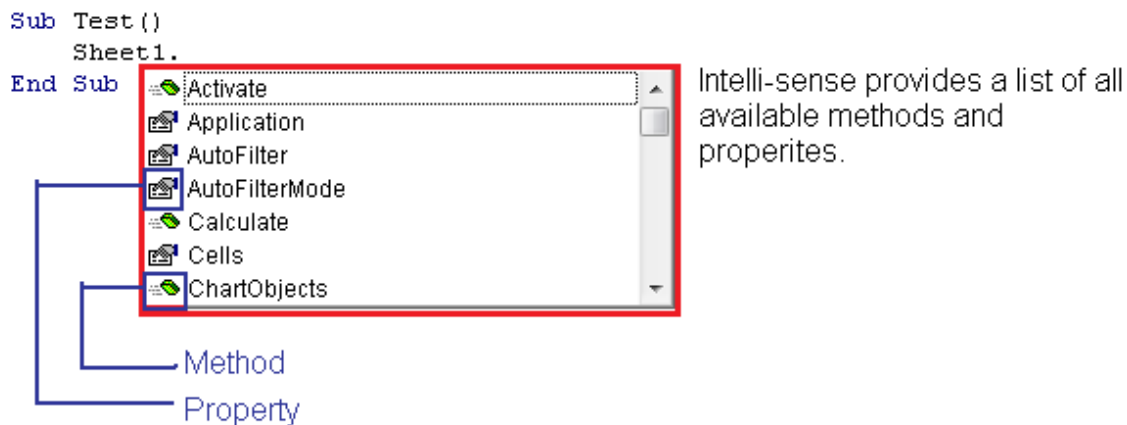
Those predefined functions affecting its characteristics such as size or colour are known as properties.

Those predefined functions that affect what an object can do are known as methods.

To access an object's properties or methods we will use the dot-notation. What that means is if we are working with an object, we will type the name of the object followed by a period, and then the name of the property or method we want to access.

For example, Sheet1.Activate means in essence for VB to go to Sheet1 and make it the active page. In this case Sheet1 is the object and Activate is its method.

You'll notice the intelli-sense works for you in this case. If you type an object followed by a period, the intelli-sense will provide you a list of all that object's properties and methods. You can identify which item in the list is a property or method based on the image in front of it. See image below.



This can also help you when you're coding. If you expect to find a method or property for an object and it doesn't appear in the list, then you may have referenced the wrong object or referenced it incorrectly.

Now let's start looking at actual code. Click the [Getting Data](#) tab on the right to continue.

# Getting Data

October-28-11  
8:27 PM

There are three places where we can get data to work with in Excel:

1. A worksheet – we can grab information from a cell in a worksheet.
2. The user – we can ask the user to input a value into a dialog box that we can then capture and use.
3. External location – we could import data from an external source as well. This process is beyond the scope of this course and is only mentioned here for completeness.

## A Worksheet

Since we are using Excel, it stands to reason that some of the data we are going to want to work with will be in one or more of the cells on a worksheet. So, we need to be able to first reference a given cell and then get at the value that it contains.

The command to reference a cell is:

**Sheet\_Name.Range("cell\_reference")**

You will provide the name of the sheet on which the value is found and inside the quotations you'll place any valid cell reference such as A1.

Now that you can point to the cell, now we need to know how to access the cell's value. *Value* is a property of the cell. So we use our dot-notation to simply add the *.Value* to the end of the previous statement:

**Sheet\_Name.Range("cell\_reference").Value**

This line of code will go to the particular sheet, the listed cell, and retrieve the value contained in that cell.

## The User

Another place we can get data from is the user. To do this we provide the user with a dialog box that allows them to enter a single value. When the user clicks OK the value can be captured and used. We'll talk about capturing the value in the next section, but to create the dialog box itself we use the following command:

**Inputbox("Prompt", "Title")**

You will provide a descriptive prompt for the user telling them what it is you expect them to enter, and you will also provide a descriptive title for the window that opens.

Now that we can get the data we'll need to store it so we can process it. Click [Storing Data](#) on the left to proceed.

# Storing Data

October-28-11  
8:27 PM

Now that we know where to get our data from, we will need temporary storage locations for that data and any intermediary steps we need to take to process that data. These temporary storage locations are known as *variables*.

A variable is simply a location in memory that holds a value. We give a descriptive name to that location. We can refer to that memory location and the value it contains simply by referencing or calling that name.

Also, we have to tell VB what type of data that memory location will contain. This is known as the variable's *data type*.

The values contained in variables are stored in different ways and used in different ways based upon their data type.

It's much like storing and using food. Depending on what type of food it is you want to store (fruit, vegetable, meat, poultry, etc.), you'll place it in either the fridge or the cupboard. Then when you go to cook it, you'll boil it, bake it, or eat it raw. It's much the same with the information you want to place in a variable. For VB to know how to handle the data, it must first know what type of data it is.

## Data Types

The first thing you as the programmer have to decide on is the type of data you want to store in your variable. For our purposes we'll look at the following data types:

1. Integer – a whole number between -32,768 and 32,767.
2. Double – double precision floating point numbers stored as IEEE 64-bit numbers in the range of -1.79769313486231E308 to -4.94065645841247E-324 for negative values and from 4.94065645841247E-324 to 1.79769313486232E308 for positive values
3. String – any combination of keyboard keystrokes can be stored in a string variable (bits of text, etc.)
4. Boolean – can be only one of two values: True or False.

These four data types will serve our purposes for this course. There are many others but, we'll stick to only these for simplicity. One note as well, placing a double value in an integer variable will result in the decimal places being dropped.

You will pick one of these based on the data you expect the variable to hold. It is always wise to pick the best fit. Even though a double can hold the value 3, it isn't wise to store that value in a double; doing so will waste memory. Since a double can hold such large numbers, they are dedicated a certain amount of space, the same amount every time one is created. To place a whole number into a memory space that can hold a double, wastes an amazing amount of memory.

! Use the most efficient data type for your variables.

## Variable Names

Once you have decided on your data type then you can decide on your variable's name. There are a few rules to keep in mind though:

1. Variable names must begin with an alphabetic character.
2. They must be unique.

3. Can't be longer than 255 characters.
4. Can't contain an embedded period or any special characters (#, ?, etc.) other than the underscore (\_).

! Also, your variable names should be prefaced with a three letter indicator as to the type of data it contains. So, if I have a variable that's supposed to contain an integer value, then you would use *int* as the preface. Others include: *str* for strings, *dbl* for doubles, and *bln* for Boolean values.

## Declaring a Variable

And finally we can declare our variable so we can use it. To do that we use a command called *dimension* (or *dim* for short). When we declare variables we always declare them at the top of our sub procedure before we start any other code. And, we must declare all variables that we are using.

**Dim** *variable\_name* **As** *data\_type*

And lastly we need to know how to get the data back to the user. Click the [Returning Data](#) tab on the right to see how.

# Returning Data

October-28-11  
8:28 PM

When we are done with the data, we need to return it so that it can be used. We can return data to one of three locations:

1. A worksheet – we can place information into a cell on a worksheet.
2. The user – we can return the data directly to the user in a simple dialog box.
3. External location – we could export the data to an external source as well. This process is beyond the scope of this course and is only mentioned here for completeness.

## The Worksheet

Returning data to a cell on the worksheet is pretty much the same as getting the data from a worksheet. First you identify the cell you want to send a value to and then tell VB what value you want put into that cell.

***Sheet\_Name.Range("cell\_reference").Value = value***

In this case, whatever is on the right hand side of the equal sign is put into whatever is on the left. Whatever is on the right will be displayed in the cell.

## The User

You can flash the user a simple message inside a small dialog box. The dialog box will display a message which can use any value stored inside a string type variable.

To use a message box, you use the following command:

***Msgbox Prompt, Button choice, Title***

***Prompt*** is the message you want to show to the user. It must be a string. It can be a string inside quotes or it can be a string variable, or even a combination of both.

***Button choice*** is your choice of what button(s) appears on your dialog box. There are quite a few selections you can have. For now, we'll only look at adding the OK button to our message boxes. So, for the buttons you'll use vbOKOnly.

And the ***Title*** is the text displayed as the title of your dialog box.

Okay, now that we have some code under our belts, let's try an example.

Click on the [Example 1: From the Worksheet](#) page to the right and follow along.

# Example 1: From the Worksheet

October-28-11  
10:11 PM

Okay, that's a lot of explanation. Let's put it to use in an example.

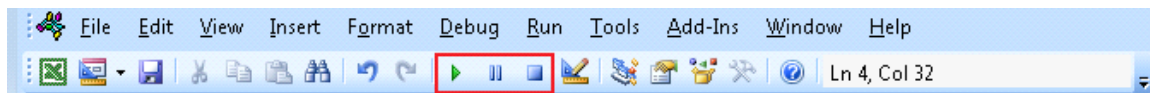
## From the Worksheet

A statement of our problem:

*We are going to take the string of characters "Hello world" from cell A1 on sheet1 and put it into cell A2 on sheet 1.*

So, let's try it:

1. Open a new workbook.
2. In cell A1 type "Hello world" without the quotes.
3. To open the VB editor, click the Developer tab, in the Code pane, click the Visual Basic button. This opens your editor.
4. In the project pane on the left hand side of the screen, if necessary, expand VBA Project(Book 1) until you see Sheet 1.
5. Double click sheet 1. This opens the code window for Sheet 1. All the code we write here will be usable only on Sheet1.
6. Create our sub procedure by adding the necessary code to the blank window. Type: `Sub Move_Text()` and then press enter. You'll notice that VB adds the closing line for you. Remember that everything we add now has to go between these two lines.
7. Now let's create our variable. Type: `Dim strText as string`. This creates our variable and now we can use it to store values.
8. Put the value from cell A1 into that variable. Type `strText = range("A1").value`.
9. Take the value from the variable and place it directly into cell A2. Type: `Range("A2").value = strValue`.
10. To test our code: at the top of the code window you should see what look like play, pause, and stop buttons. Click the play button.



11. Check sheet1 and you see the value "Hello world" printed in both cells.

```
Sub Move_Text()  
    Dim strText As String  
    strText = Range("A1").Value  
    Range("A2").Value = strText  
End Sub
```

This example should raise two questions:

1. If we're moving the text, how do we clear cell A1 so that it actually looks like we moved it and didn't just copy it?
2. And, users won't use the play button so how can I add a button so that the user can simply click a button to run the code?

## Clearing the Cell

To clear a cell we simply have to tell VB to put 'nothing' in the cell. We represent nothing by using a set of double quotes. So, if we want to assign nothing to a cell we do it this way:



`Range("cell_range").Value = ""`

In our example we would add `Range("A1").value = ""` to our code in one of two spots: between lines 3 and 4, or after line 4. See the image below.

```
Sub Move_Text()  
    Dim strText As String  
    strText = Range("A1").Value  
    Range("A1").Value = ""  
    Range("A2").Value = strText  
End Sub
```

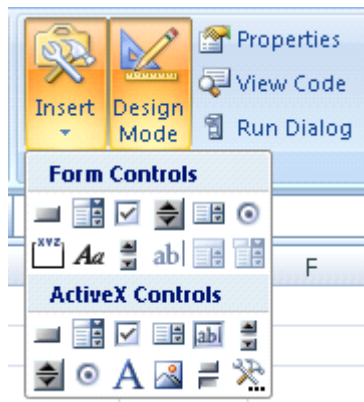
	A	B
1		
2	Hello world	
3		

## Working with Buttons

### Adding a Button

We need to add a button to our worksheet. Click on Sheet 1 so that you can see it.

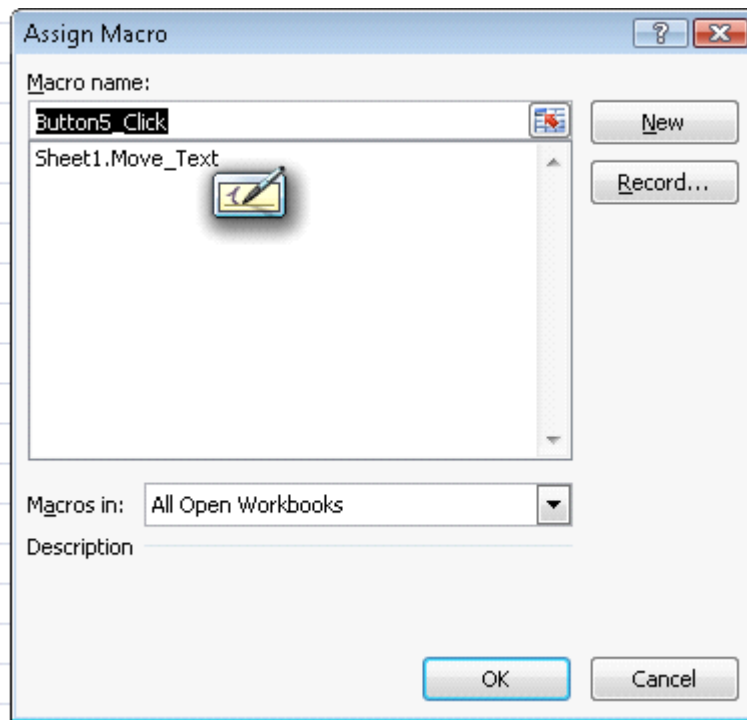
Click the Developer tab and look for a pane called *Controls*. The buttons in this pane will allow us to create our button. Click the Insert button and a small drop down appears. This drop down is divided into 2 sections: Form Controls and ActiveX controls.



To create this button we'll use the Form control version of a button. So under the Form Controls title click the image representing a button. Your cursor will change to a plus sign. Simply click and drag on sheet 1 to create the button.



Also, this opens the Assign Macro dialog box. In this box you should see the macro we created listed. Simply click our macro so it's selected and then click OK.



### Changing the Text

Simply highlight the current text and change it to Click Me.

### Using the Button

Simply click anywhere on the sheet and the sizing handles will disappear. Then when you move your mouse over the button it will appear as a hand. That means you can click on the button and it will work.

How about one more example except this time we'll take text from the user and place it in cell A1. Click [Example 2: From the User](#) on the right to give it a try.

## Example 2: From the User

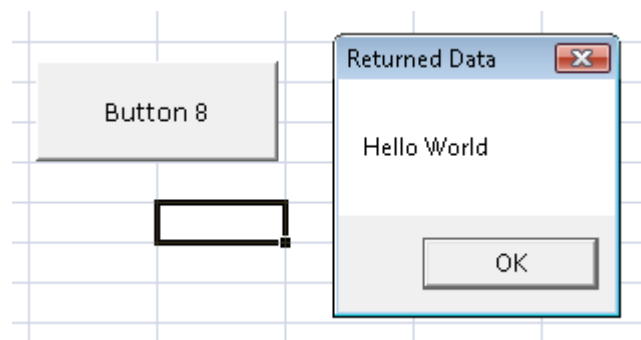
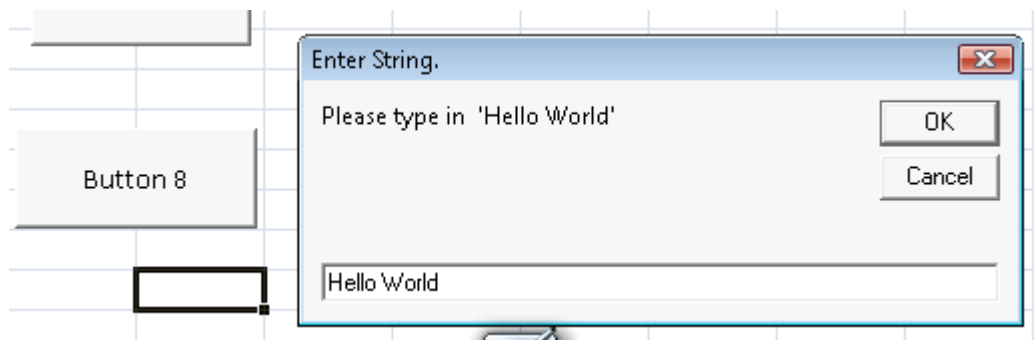
October-28-11  
10:28 PM

With your VB Editor still open click on Sheet 2 in the Project Pane.

Type in the following code:

```
Sub Move_Text2()  
    Dim strText As String  
    strText = InputBox("Please enter 'hello World'", "Enter data.")  
    MsgBox strText, vbOKOnly, "Returned Value"  
End Sub
```

Add a button to Sheet 2 that will run your code and try it. See the images below for a visual example.



A few things to note:

1. The Inputbox command is placed to the right of a variable. This is so that the value the user types in is to be placed into a variable (on the left of the equals sign) for use later.
2. And notice that what we typed in for the prompt of our message box is a string variable. We could have used "Hello world" instead, but the text was held in a variable so we used that.

So, now that we can get, store, and return data, we can move on to more interesting uses of data. Click the [Calculations](#) tab at the top of the book to continue.

# Order of Operations

October-28-11  
8:28 PM

Programming is all about the math (at least for our purposes). The symbols used for calculations are a bit different than what you might be used to.

()	Only round brackets are used.
^	Used to indicate an exponent.
/	Division.
*	Multiplication.
+	Addition.
-	Subtraction.
%	This is the mod function. It returns the remainder from division.

Since grade school you've had the order of operations drilled into you. It looks something like this:

Brackets	Any equation inside a set of brackets is calculated first starting from the inner most set and working outwards.
Exponents	Next you calculate any exponents that are present.
Division and Multiplication	Calculate any division or multiplication in the order in which they appear from left to right.
Addition and Subtraction	Calculate any addition and subtraction in the order in which they appear.

When writing equations in code, you as the programmer have to enforce the order of operations by placing extra brackets where necessary.

Now usually you'll be given a formula that looks something like this:

$$\frac{3x^{2+i}}{\sqrt{3y-1}}$$

The problem is that formulas like this all have to be written on a single line to be of use in VB. So you have to do the conversion and it should come out to look something like this:

$$(3 * x^{(2 + i)}) / (Sqr(3 * y - 1))$$

Three things of note:

1. Exponents are indicated using the ^ symbol (above the 6 on your keyboard).
2. You must include the symbol for multiplication (\*), so 3y has to be 3\*y.
3. To create the square root of a number we use the *sqr* function.

There are a number of special functions and we'll introduce a few of them as we go along.

A function is simply some pre-defined code that we can call when needed to perform a specific function.

For example, if we wanted to calculate the square root we simply use the function rather than write the code to do it every time we needed it.

There is one other thing to keep in mind. When doing calculations the data we are working with all has to be of the same type. Adding a double to an integer creates some problems for our calculations so we need to know how to convert data types so we're using all the same types in our equations.

Click on the [Conversion](#) page to the right to see how.

# Conversion

October-28-11  
8:32 PM

## Type Mismatch Errors

When working with data, if you try to do something with a piece of data that is of the wrong type (for example: trying to multiply the word 'dog' by 2) you'll get an error known as a *type mismatch error*. This means you are trying to perform an operation on a piece of data that is of the wrong data type.

To correct these problems you need to be able to convert data between types.

## Conversion

As stated before we have different types of data we'll be working with: strings, integers, and doubles (Boolean is a special data type we'll look at separately in later section). Data types don't mix very well or at all. You'll need to be able to identify a data type and then know how to convert it to one of the others.

## Integers and Doubles

When you're doing calculations you'll use one of two number data types: integers and doubles. If you remember, the key difference is that integers are whole numbers and doubles can contain decimals. Also the ranges are significantly different. [Click here to review the material.](#)

As an example, check the code snippet below and the result:

```
Sub test()  
    Dim intNumber As Integer  
    Dim dblNumber As Double  
    dblNumber = Range("A1").Value  
    intNumber = Range("A1").Value  
    Range("A2").Value = "The double variable's value:"  
    Range("A3").Value = "The integer variable's value:"  
    Range("B2").Value = dblNumber  
    Range("B3").Value = intNumber  
End Sub
```

	A	B	C
1	2.22		
2	The double variable's value:	2.22	
3	The integer variable's value:	2	
4			

Two variables are declared: an integer and a double. A value is taken from a cell and placed into each variable. Then, the value from each variable is exported back to the worksheet so we can see the value in the variable.

Notice what happened to the decimals. They were dropped when the value was placed in the variable with the data type of integer. This type of logic error happens more than you might think. If we were to do calculations using the integer variable we would have lost the precision provided by the user.

We need to be able to convert between the different data types in order to preserve things such as precision.

To convert we use one of two commands depending on which way we're trying to go.

### CInt

To ensure that you have an integer value where required, you can use the CInt function to convert a data type to integer.

*Int\_Variable\_name* = CInt(*Variable\_name*)

### CDBl

To ensure that you have a double value where required, you can use the CDBl function to convert a data type to a double.

*DBl\_Variable\_name* = CDBl(*Variable\_name*)

### Strings

In certain circumstances it's necessary to convert a number into a string (for example, if you wanted to output the number using a message box).

### CStr

To ensure that you have a string value where required, you can use the CStr function to convert a data type to a string. A string is simply any combination of letters or numbers from the keyboard.

*Str\_Variable\_name* = CStr(*Variable\_name*)

Okay, let's try a few examples to see how this all works. Click on [Example 1: Basic Math](#) on the right to proceed.

## Example 1: Basic Math

October-29-11  
8:21 AM

We've covered quite a few things in the last few sections so let's take a step back and try putting this all together in a simple example. We now have the capability of writing a sub-procedure that can take information in, store it, process it, and then return a result. Let's try a few simple examples.

To start, a statement of the problem:

*Take the value stored in A1 and substitute it for x in the following formula:*

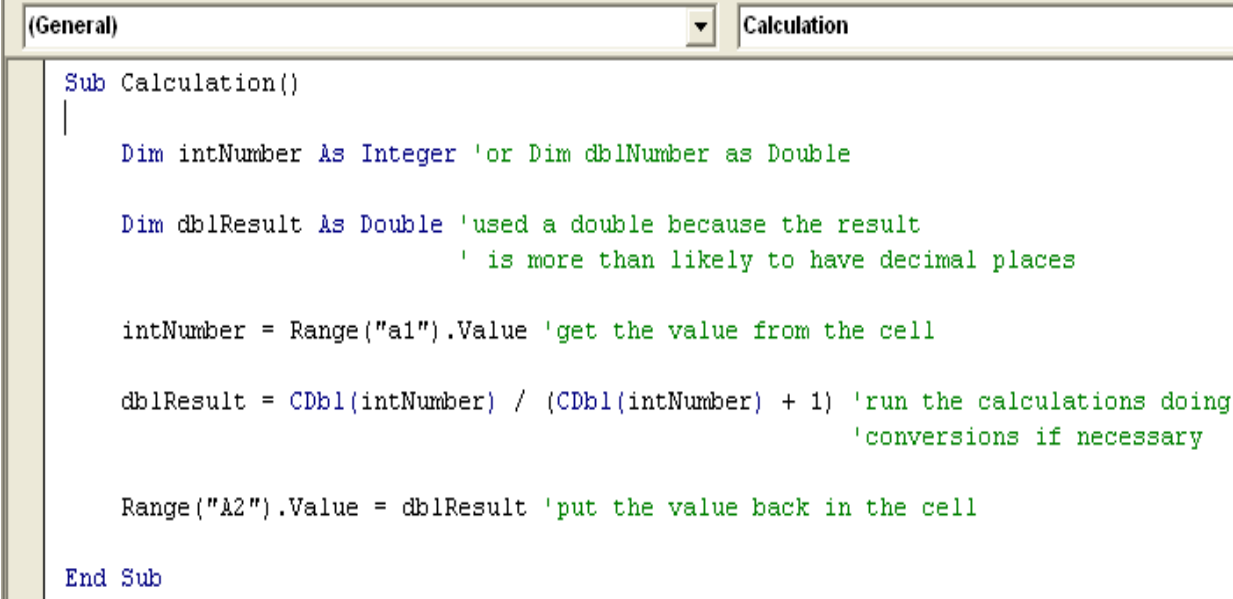
$$\frac{x}{x+1}$$

*Display the results to cell A2.*

Let's open a new workbook and try this.

1. On Sheet 1 let's enter the number 5 in cell A1.
2. On the Developer tab, in the Code pane on the left, click the Visual Basic button to open the editor.
3. On sheet 1 create a new sub procedure called *Calculation*.
4. Create a variable to hold the value from cell A1. Remember to declare all your variables at the top of the sub procedure before you use them. What type of variable are you going to use?
5. Get the value stored in A1 and place it into this new variable. Do you need to convert it? And, if so what would you do?
6. Write the code to run the calculation and place its value into a variable.
7. Put the result into cell A2.

Your code will look something like this:



```
Sub Calculation()  
    Dim intNumber As Integer 'or Dim dblNumber as Double  
  
    Dim dblResult As Double 'used a double because the result  
                            ' is more than likely to have decimal places  
  
    intNumber = Range("a1").Value 'get the value from the cell  
  
    dblResult = CDbl(intNumber) / (CDbl(intNumber) + 1) 'run the calculations doing  
                                                         'conversions if necessary  
  
    Range("A2").Value = dblResult 'put the value back in the cell  
  
End Sub
```

Since the number in the cell doesn't have decimals we use the integer data type. But, since we will more than likely end up with decimals after the calculations we need a double variable to hold the results. Which means we have to do a conversion to double so that we keep our precision.

Laos, note the use of brackets to force the order of operations. The denominator has to be surrounded by brackets so that it is calculated first before we do the division.

Now, let's try working with strings. Click on the [Example 2: Strings](#) page on the right to see.



## Example 2: Strings

October-29-11  
8:31 AM

One type of data that you may come across is strings. A string can be any combination of keyboard or special characters that can be stored in a variable of type string. This little example will show you a few things about strings.

*Using an input box, ask the user to supply their name. Store the name then display a message box that looks like the following image:*



There are a few things to note about the following code:

```
Sub Display_Name()  
    Dim strName As String  
  
    strName = InputBox("Please enter your name.", "Name entry:")  
  
    MsgBox "Hello " & strName & ". " & vbCrLf & "Have a nice day.", vbOKOnly, "Response:"  
End Sub
```

First to create the sentence containing the user's name, we actually have 4 parts. Each part is separate and has to be brought together to form one string. When you need to concatenate or put strings together, you use the & symbol between the items. In this case we could have used a numeric variable as well if we wanted to output a number instead of the user's name.

Also, note the key word vbCrLf. This special keyword can be used to cause a carriage return, and line feed. For those younger folks among us, that used to mean move the head of the printer to a new line. In this case it means pretty much the same thing. This is the command that splits one single line of text into two complete lines in the message box.

Now let's see how we can work with the Inputbox and converting strings to actual numbers. Click on the [Example 3: Converting Strings](#) page to see what that's all about.

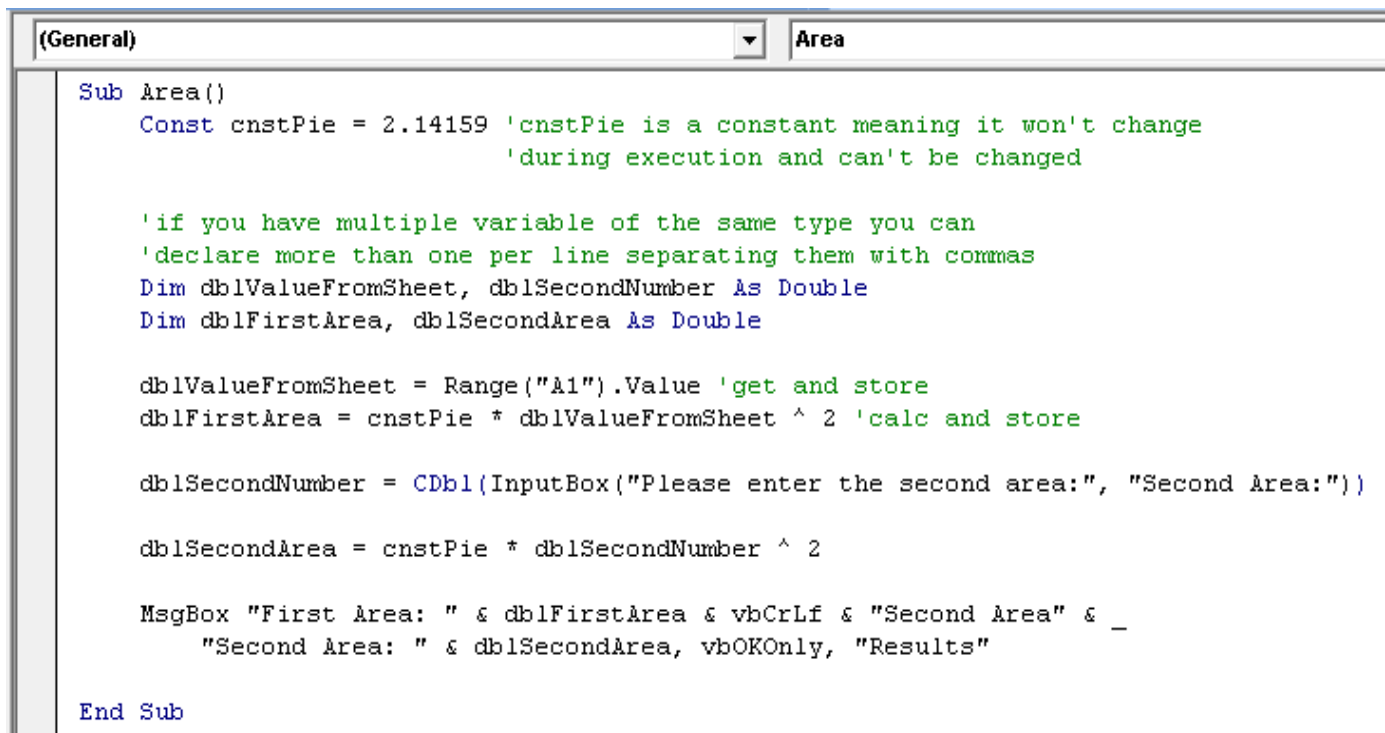
## Example 3: Converting Strings

October-29-11  
8:34 AM

One of the problems with asking the user for values is that the Inputbox command returns whatever the user enters as a string. So, this means you have to do the conversion.

*Create a worksheet that calculates the area of a circle using a value in cell A1 for the radius ( $\pi = 3.14159$ ). Then ask the user for another number and recalculate the area of the circle and display both values to the user in a message box.*

Here's what our code would look like:



```
Sub Area()  
    Const cnstPie = 2.14159 'cnstPie is a constant meaning it won't change  
                             'during execution and can't be changed  
  
    'if you have multiple variable of the same type you can  
    'declare more than one per line separating them with commas  
    Dim dblValueFromSheet, dblSecondNumber As Double  
    Dim dblFirstArea, dblSecondArea As Double  
  
    dblValueFromSheet = Range("A1").Value 'get and store  
    dblFirstArea = cnstPie * dblValueFromSheet ^ 2 'calc and store  
  
    dblSecondNumber = CDBl(InputBox("Please enter the second area:", "Second Area:"))  
  
    dblSecondArea = cnstPie * dblSecondNumber ^ 2  
  
    MsgBox "First Area: " & dblFirstArea & vbCrLf & "Second Area: " & _  
        "Second Area: " & dblSecondArea, vbOKOnly, "Results"  
  
End Sub
```

All right, so the newest thing here is the *cnstPie*. This is a new type of variable known as a constant. A constant is used to hold a value that will not change and shouldn't be allowed to change while your program is running. We use constants for values like pi and Euler's number. Also, by declaring these values at the top of our sub procedure, if necessary, we can change them in one place and we don't have to look for them.

Take a look at the code for the message box. That was going to be a pretty long line once we added in all the different parts of the output string. If you need to spread a line of code over more than one line you need to use the line continuation character which is really just an underscore. The trick is knowing where to put it. You can't just put it anywhere. You have to put it at a logical break in the command. In this case we put it just after an & symbol. Unfortunately there's no hard and fast rule. Knowing where to put it comes with some experience.

Okay, so now that we've looked at a few examples, why don't you try some? Click on the [Practice Assignments](#) page to the right and try for your self.

# Practice Assignments

October-29-11  
1:52 PM

## Practice Exercises: Working with Data

1. Which of the following are valid variable names:

2ndNumber			Second_Number
Second Number			Second.Number
_SecondNumber			Second_Number

2. Look at the following sub procedure. There are five errors. Can you spot them all?

Sub Calc area

Dim intFirst.Number as Integer

dblArea = 2.14158 \* Cdbl(intFirstNumber) ^ 2

msgbox "Your value is: " % dblArea, vbOKOnly, "Result"

End Sub

Note for the following practice exercises, even though it doesn't say so, you should plan these out before you code. Please take the time to do so.

Start a new workbook and on sheet 1:

3. Write a sub procedure that takes a number from cell A1 then uses that number to calculate the perimeter of the circle. Display the result in cell A2 ( $P = 2\pi r$ ).

4. Write a sub procedure that takes numbers from cells A1 and A2 and uses them in the following formula:  $z = \sqrt{x^2 + y^2}$ , where A1 = x and A2=y. Display the result in message box.

5. Write a sub procedure that asks (use input boxes for this question) the user for their name, asks for 2 numbers, and then uses those numbers to calculate the area of a triangle (the first number is the height and the second is the base;  $A = 1/2bh$ ). Display a personalize message for the user: Hi Dave. The area of your triangle is..."

6. Design a worksheet that will allow the user to enter a radius (your choice to use a cell or input box) and then click one of three buttons to calculate either: the area, the diameter, or the circumference of a circle. You may display the result as you see fit. (Hint: when programming, each button gets its own sub procedure.)

Now, when you're ready, tackle the Final Assignment. Click on the [Final Assignment](#) page on the right.

# Final Assignment

October-29-11

1:59 PM

## Assignment: Working with Data

1. Open a new workbook and name it yourname\_WorkingWithData.xmls. Delete all worksheets except sheet 1. Rename it to CalcAge. You will write a program that satisfies the following requirements:
  - a. You will gather the user's name. The method is up to you.
  - b. Next you will gather the user's date of birth. The method is up to you. Use the Help system to find out about the new data type date. Your dates should use the format mm/dd/yyyy.
  - c. You will then calculate the user's age. Use the Help to look up a function called datediff to help you in this operation and a function called now as well.
  - d. Display the result to the user with a personalized message.

# Controlling Program Flow

October-29-11  
2:09 PM

Okay, now that the basics are out of the way, and you can write some code, we need to look at how to control the flow of our program. For example, if the user enters 'yes' at a particular prompt we will want to execute certain lines of code. If the user enters 'no' at the same prompt we want to avoid running those lines of code. The one major strength of programming is that we can decide which lines of code we want to execute based on a particular result.

When programming our choices can have only one 2 possible outcomes: yes or no/true or false. It's important to note that the decision is made and the result of the decision controls which direction our code takes. To produce this type of result it's important for you to understand *expressions*.

Expressions are simply a mathematical device that result in a yes or no/true or false answer.

For example, does  $3 = 4$ ? This simple math expression results in no. We can use these expressions to our advantage while programming.

When we write these expressions there are a only a few math operators we can use:

=	Compares to values to see if they are the same. Returns YES/TRUE if they are and NO/FALSE if they are not.
>	Compares two values to see if the one on the left is bigger than the one on the right. Returns YES/TRUE if it is and NO/FALSE if it isn't.
<	Compares two values to see if the one on the left is smaller than the one on the right. Returns YES/TRUE if it is and NO/FALSE if it isn't.
>=	Compares two values to see if the one on the left is bigger than or the same as the one on the right. Returns YES/TRUE if it is and NO/FALSE if it isn't.
<=	Compares two values to see if the one on the left is smaller than or the same as the one on the right. Returns YES/TRUE if it is and NO/FALSE if it isn't.
<>	Compares two values to see if they are unequal. Returns YES/TRUE if they are and NO/FALSE if they are not.

Every expression has three parts: something to the right of the symbol, the symbol, and something to the left.

These 'somethings' can be hard coded numbers (like 3), variable names, or equations using a mixture of both.

Now let's see how we use these expressions to our advantage. Click the [IF...END IF](#) page to the right to see how.

# IF...END IF

October-28-11  
8:30 PM

In VBA the code structure we use to evaluate our expressions is called the IF statement.

The basic IF statement has five major parts:

1. The opening key word: IF
2. The expression to be tested that will be either yes or no/true or false.
3. The keyword: Then
4. The code to execute if the expression results in yes/true.
5. And the closing keywords: END IF

For example:

```
If condition then  
    code to execute  
End if
```

When the program encounters an IF statement the expression is evaluated. If the expression results in yes/true the code between the IF and END IF lines is executed. If the result is false, then the program continues executing at the first line after the END IF. The code between the two lines isn't executed.

As an example:

```
Dim intX as integer  
Dim intY as Integer
```

```
intX = 3  
intY = 2
```

```
If intX = intY then  
    MsgBox "Yes.", "Result"  
End if
```

Would you see the message box?

In this case, no you wouldn't see the message box. The condition `intX=intY` would evaluate to false because 3 doesn't equal 2. If a condition evaluates to false we do not execute the code between the IF and END IF lines.

Let's extend the capabilities of the IF statement. Click the [A Few Other Notes](#) page to the right.

## A Few Other Notes

October-31-11  
10:14 AM

### Nesting IF Statements

Sometimes you will need to evaluate many expressions before executing a segment of code. You will evaluate one expression. If that expression is true, you'll need to evaluate a second before executing the code. Or, you may even need to evaluate further.

If you have a case like this, we can embed or nest IF statements inside each other. Like this:

```
If condition then
    If condition then
        code to execute
    End if
End if
```

If the outer IF is yes or results in true, we would then check the next expression. If the inner expression is yes or true we execute the code. If it turns out to be no or false we execute the first line after its END IF. Well that happens to be another END IF so we move to the first line of code after the last End If.

An example:

```
Dim intX as integer
Dim intY as Integer
```

```
intX = 3
intY = 2
```

```
If intX = 3 then
    If intX = intY then
        MsgBox "Yes.", "Result"
    End if
End if
```

Would we see the message box?

In this case, no we wouldn't. The first expression (*intX = 3*) evaluates to yes or true. Then the second is tested. As before this one results in a no or false result. Control jumps to the End If line and continues.

### Multiple True Conditions

Sometimes before you execute code you'll need to get positive results from more than one expression. For example: you don't want to add two numbers unless they are both divisible by 2. In other words both numbers have to be even.

We would need to evaluate both conditions at the same time (in the same opening line of an if statement) and get two true results. The trick here is to realize that we have actually three conditions we're evaluating. One on the left; one on the right; and then we need to evaluate whether both expressions (one on the left and one on the right) evaluated to yes or true. If both were true then we're going to add the numbers together. If one of them is no or false we aren't going to add them together.

Our first condition would look like this:

`intNum1 Mod 2 = 0`

`intNum1` is a variable holding our first number.

`Mod` is a VBA reserved function that divides the number on the left by the number on the right and returns only the remainder. In this case we are looking for numbers evenly divisible by 2 meaning there are no remainders or a result of 0.

The condition on the right is similar:

`intNum2 Mod 2 = 0`

So, now we have two conditions that will evaluate to either true or false. Our next step is to combine them together in the first line of an if statement so that we evaluate their results. If both return yes or true we want to execute the code. The problem is we need some way to tell VBA we want to execute our code only if both of them are true.

To do this we use another keyword called `AND`. `AND` is a special Boolean operator. Boolean logic deals with the interaction of 1's (yes or true) and 0's (no or false) within your computer. When you compare two values using a Boolean operator it's either true or false (1 or 0).

Value 1	Value 2	Result after AND
0	0	0
1	0	0
0	1	0
1	1	1

0 = no or false  
1 = yes or true

In the case of `AND`, a result of yes or true only appears when both values being compared are yes or true. Therefore, for our purposes here, we use the keyword `AND` between our two statements to create our IF statement as follows:



```

Sub Add_Even_Numbers()
    Dim intNum1, intNum2 As Integer

    intNum1 = Range("A1").Value
    intNum2 = Range("A2").Value

    If ((intNum1 Mod 2) = 0) And ((intNum2 Mod 2) = 0) Then
        Range("A3").Value = intNum1 + intNum2
    End If

End Sub

```

From the code in the image, be aware of the use of brackets.

Open a worksheet, record the code as you see it above, and test it to see what you get.

## One of Many Choices

Sometimes before you execute code you'll need to get positive results from only one expression of many. For example: you want to add two numbers as long as one is divisible by 2. In other words only one has to be even.

We would need to evaluate both conditions at the same time (in the same opening line of an if statement) and get only one true result. The trick here is to realize that we have actually three conditions we're evaluating. One on the left; one of the right; and then if one of the two expressions is true then we're going to add the numbers together.

Our first condition would look like this:

```
intNum1 Mod 2 = 0
```

intNum1 is a variable holding our first number. Mod is a VBA reserved function that divides the number on the left by the number on the right and returns only the remainder. In this case we are looking for numbers evenly divisible by 2 meaning there are no remainders.

The condition on the right is similar:

```
intNum2 Mod 2 = 0
```

So, now we have two conditions that will evaluate to either true or false. Our next step is to combine them together in the first line of an if statement so that we evaluate their results. The problem is we need some way to tell VBA we want to execute our code only if one of them is true.

To do this we use another keyword called OR. OR is a special Boolean operator. Boolean logic deals with the interaction of 1's and 0's within your computer. When you compare two values in Boolean logic it's either yes/true or no/false (1 or 0). In the case of OR, a result of true appears when at least one of the two values is true. If both values are false, then false is returned.

Value 1	Value 2	Result after OR
0	0	0
1	0	1

0 = no or false  
1 = yes or true

0	1	1
1	1	1

Therefore, for our purposes here, we use the keyword OR between our two statements to create our IF statement as follows:

```
Sub Add_Even_Numbers2 ()
    Dim intNum1, intNum2 As Integer

    intNum1 = Range("A1").Value
    intNum2 = Range("A2").Value

    If ((intNum1 Mod 2) = 0) Or ((intNum2 Mod 2) = 0) Then
        Range("A3").Value = intNum1 + intNum2
    End If

End Sub
```

From the code in the image, be aware of the use of brackets.

Open a worksheet, record the code as you see it above, and test it to see what you get.

Okay, so let's extend the capabilities of our IF statements further. Click [IF...ELSE...END IF](#) page on the right.

## IF...ELSE...END IF

October-28-11  
8:33 PM

Sometimes you'll make a decision in a program and execute code if the condition evaluates to true. But, what if it evaluates to false? There might come a time when you want to execute other code even though you got a false result.

In this case, we expand our basic IF statement to include an else statement. If your initial condition evaluates to no or false then instead of going to the End If line, the program control moves to the ELSE and whatever code follows the ELSE is executed.

The basic structure looks like this:

```
If condition then
    code to execute
else
    code to execute
End if
```

As an example, we'll modify the code from the Or statement above and flash the user a message box saying that one of the numbers must be even. Here is the modified code:

```
Sub Add_Even_Numbers3()
    Dim intNum1, intNum2 As Integer

    intNum1 = Range("A1").Value
    intNum2 = Range("A2").Value

    If ((intNum1 Mod 2) = 0) Or ((intNum2 Mod 2) = 0) Then
        Range("A3").Value = intNum1 + intNum2
    Else
        MsgBox "One number must be even.", vbOKOnly, "Result"
    End If

End Sub
```

In this case, if the user supplies us with 2 uneven numbers the initial condition evaluates to false. If it's false then the control passes to the else statement and its code is executed giving us the message box.

Note, that if any part of the initial condition is true the code above the else is executed and once that code is done, control passes outside the if statement skipping over the else and its code.

This is a simple example of data validation.

Finally, let's expand the IF structure one more time. Click the [IF...ELSE IF...ELSE...END IF](#) page to the right to see what's next.

## IF...ELSEIF...ELSE...END IF

October-28-11  
8:34 PM

You may encounter a situation where you have multiple related conditions you need to test. In this case we expand upon the IF statement further. Let's take our examples from before and expand on them a bit further.

Let's say we take two numbers. If both are even, we'll add them. If only one is even, we'll multiply them. If neither is even, we'll tell the user at least one needs to be even.

To do this we'll need to expand upon our if statement. We can add more conditions to the test using the ELSEIF statement. Each ELSEIF statement gets its own condition to test. If the condition is true then the code it contains is run and then control passes out of the IF. If it evaluates to false then the program moves down to the next part of the if statement. Here is what the modifications would look like:

```
Sub Add_Even_Numbers_Combined()  
    Dim intNum1, intNum2 As Integer  
  
    intNum1 = Range("A1").Value  
    intNum2 = Range("A2").Value  
  
    If ((intNum1 Mod 2) = 0) And ((intNum2 Mod 2) = 0) Then  
        Range("A3").Value = intNum1 + intNum2  
    ElseIf ((intNum1 Mod 2) = 0) Or ((intNum2 Mod 2) = 0) Then  
        Range("A3").Value = intNum1 * intNum2  
    Else  
        MsgBox "One number must be even.", vbOKOnly, "Result"  
    End If  
  
End Sub
```

Notice each ELSEIF (one word, no space) gets a condition and a THEN keyword as well.

Also, you can have as many ELSEIF statements as you need to test all the expressions you need.

Now let's get you to try some. Click on the [Practice Exercises](#) page on the right to try your skills.

# Practice Exercises

October-31-11  
11:24 AM

1. Ask a user to enter the fault tolerance (eg. .34) in millimeters (your choice of methods). Use the table below to create a program that will warn the user based on the fault tolerance they have entered.

Tolerance Range	To be Displayed
0.0-0.29	Warning – Outside of range
0.30-0.50	Within range
0.51 or greater	Warning – outside of range

2. Create a worksheet for a user that allows them to enter two numbers (the length and width of the sides of a rectangle). Write a program that tests the two numbers to ensure they are both positive and that there are numbers entered (test the cells to see if they are empty – don't worry about text). If they are positive, use those numbers to calculate the area of a rectangle and display the results to the user in a labeled cell. If they aren't or the cell(s) is empty, let the user know they have to fix the numbers. CHALLENGE: tell the user specifically what they did wrong.
3. Take two integer numbers from a user (use the method of your choice). Check those numbers. If the first number is between 0 and 10 and the second number is between 1 and 10, display the remainder if you divided the first number by the second. Also, use the table below to display a specific message based on the remainder:

Remainder	Message
0-3	Below target
4-6	Near target (low)
7-9	On target
10-12	Near target (high)
13 or greater	Over target

Now for the real test. Click on the [Assignment](#) page to the right to see if you really understand IF statements.

# Assignment

October-31-11  
11:30 AM

You will need to be able to do conversions between units in the future. Specifically for this assignment we'll be looking at the metric conversion for units of power. Review the chart below:

## Metric System

To convert	Multiply by	To obtain
watts	0.7376	ft-lb/sec
watts	0.001341	hp
kW	1.3410	hp
cheval-vap	0.9863	hp
ft-lb/sec	1.356	watts
hp	745.7	watts
hp	0.7457	kW
hp	1.0139	cheval-vap

You will create a user friendly application that will allow a person to enter a number and pick or enter what unit it is.

Then the user will pick what unit to convert to. You will then ensure that the user has picked valid units and entered a number (do not allow the user to enter nothing; don't worry about them entering text).

You will then perform the required calculation and show the user the answer.

You are being given freedom to design this program the way you think best. Remember though, it should be user friendly with detailed design and responses to tell a user when they go wrong.

Hand that in and then click on [Repetition](#) tab at the top of the screen.

# Repetition

October-28-11  
8:31 PM

There are going to be times when you will need to repeat certain lines of code a specific number of times.

For our purposes here, we will look at two different repetition structures (one of them has a bunch of different forms), called loops.

Every type has three basic parts to it:

1. A keyword to start the loop.
2. An expression that has to be evaluated to stop the loop.
3. And a line to mark the end of the loop.

We have to be careful, a loop that has no end is known as an endless loop. Your program will continue to run, essentially, forever. The expression is a very important part of a loop.

Click on the [DO UNTIL...LOOP](#) page on the right to get started.

# DO...LOOP

October-31-11  
11:36 AM

Our first structure will be the DO...LOOP. There are a few variations on the DO...LOOP, but they all repeat the lines of code until an expression is evaluated.

The basic structure looks like this:

```
Do Until (expression)  
    code to execute  
Loop
```

Or

```
Do While (expression)  
    code to execute  
Loop
```

The keywords UNTIL and WHILE change how our expression is evaluated.

When using UNTIL, the code will be executed UNTIL the expression evaluates to true. So, as long as it evaluates to false the loop will continue to run our code.

When using WHILE, the code will be executed WHILE the expression evaluates to true. So, as long as it evaluates to true the loop will continue to run our code.

Examples:

```
DIM X as Integer  
X=0  
DO UNTIL (X=3)  
    X = X + 1  
LOOP
```

This particular example, will run though the loop three times. After each time through, control returns to the top of the loop, and the expression is re-evaluated. If the expression is true, control moves to the line after the LOOP keyword. If the expression is false the code is executed again.

```
DIM X as Integer  
X=0  
DO WHILE (X<3)  
    X = X + 1  
LOOP
```

This example does the same as the above example, but the condition is different. In this case, as long as the condition is true the code is executed. When the condition becomes false, the loop ends and control passes to the line after the LOOP keyword.

In both these cases, X is acting as our *flag*. A flag is a variable that controls when the loop is completed. The flag is used in our expression to ensure that we have an ending to our loop. It is very important to have a flag and make sure that you are testing it. An endless loop will run as long as there is memory to



use up.

## Changing When to Test the Expression

The expression can be moved to the bottom of the loop as well. For example:

Do

*code to execute*

Loop Until (*expression*)

Or

Do

*code to execute*

Loop While (*expression*)

The basic functionality doesn't change. The loops work the same except that the expression is evaluated at the end of each pass through the loop.

This makes one significant change. This means that the code inside the loop is run once before the expression is tested.

Let's look at an example.

For this example let's write a program that will list the numbers 1-10 inclusive into the cell range A2-A11.

```
Dim intNum As Integer

intNum = 1
Range("A2").Activate 'make A1 the active cell

Do While (intNum <= 10)

    ActiveCell.Value = intNum 'add the value to the active cell

    intNum = intNum + 1 'increase variable by 1

    'Move down a one cell
    ActiveCell.Offset(1, 0).Range("A1").Select

Loop
```

There are a few things going on here.

First off, there is a set of statements that allow us to actively control which cells are currently active and that allow us to move the cell indicator (the black box around the currently selected cell) around the worksheet.

The .Activate method of the Range object allows us to move the cell indicator directly to a given cell (Range("A2").Activate).

The ActiveCell.Value statement acts like our .Value properties from before, except that it accesses whatever cell has the indicator. You don't need to know the current cell reference. You just need to know that whatever cell currently has the indicator will be the one used.

Finally, the `ActiveCell.Offset()` command allows us to move our indicator either up or down, left or right from the current cell. The first number in the first set of brackets indicates the number of rows to move down and the sheet. The second number indicates the number of columns to move left or right. The part after the decimal will remain the same for our purposes. Don't change it.

Now that we can move the cell around the sheet, we can print the number to the screen. And that is simply using the loop. Our loop will continue to run while `intNum <= 10`. To start `intNum` is 1. The first line inside the loop prints the value in `intNum` (1) to the active cell (A2). We add one to `intNum` making it 2. Then, the cell indicator is moved to the next cell down (`Offset`). Control returns to the top of the loop and tests the condition again. `intNum` is still less than 10 so we execute the code again. We print `intNum`'s value (2) to the active cell (A3). Add 1 to `intNum` (3). Move to the next cell (A4). And, we go back to the top of loop.

This process continues until we change `intNum` from 10 to 11. We add one to `intNum` to change it to 11. We move to the next cell and then return to the top of the loop to retest the condition. This time our condition evaluates to false. Once this happens control passes to the first line after the loop line.

Take some time and create a worksheet that uses this code before continuing.

Below are three variations using different looping structures. Try each one before continuing.

### Variation 1

```
intNum = 1
Range("B2").Activate 'make B1 the active cell

Do Until (intNum > 10)
    ActiveCell.Value = intNum 'add the value to the active cell

    intNum = intNum + 1 'increase variable by 1

    'Move down a one cell
    ActiveCell.Offset(1, 0).Range("A1").Select
Loop
```

### Variation 2

```
intNum = 1
Range("C2").Activate 'make B1 the active cell

Do
    ActiveCell.Value = intNum 'add the value to the active cell

    intNum = intNum + 1 'increase variable by 1

    'Move down a one cell
    ActiveCell.Offset(1, 0).Range("A1").Select
Loop Until (intNum > 10)
```

### Variation 3

```
intNum = 1
Range("D2").Activate 'make B1 the active cell

Do
    ActiveCell.Value = intNum 'add the value to the active cell

    intNum = intNum + 1 'increase variable by 1

    'Move down a one cell
    ActiveCell.Offset(1, 0).Range("A1").Select
Loop While (intNum <= 10)
```

Let's take a look at two options for use with loops. Click on the [Exit Do and Boolean Variables](#) page on the right.

# Exit Do and Boolean Variables

October-31-11  
3:48 PM

Sometimes it would be useful to stop the loop once you've done what you wanted. For example, you want to search for something, and when you find it you don't want to finish out the loop. If you want to stop a loop before the condition is met, you can use the Exit Do command. When the computer executes this command, program control passes directly to the first line after the loop line.

For example:

I have a list of numbers in column A. I'm looking for the first occurrence of the number 5. Once I find it I want to stop the loop. There's no point continuing until the end. In this case, my loop has to be set to check the whole list (for my purposes the end of the list will be the first empty cell encountered). So when I find the number I will use Exit Do to get out of the loop. I will also need to tell the user, using a message box, whether the number was found or not.

```
Sub Search()  
  
    Dim blnFlag As Boolean  
  
    Range("A1").Activate  
    blnFlag = False  
  
    Do Until (ActiveCell.Value = "")  
        If ActiveCell.Value = 5 Then  
            blnFlag = True  
            Exit Do  
        End If  
        ActiveCell.Offset(1, 0).Range("A1").Select  
    Loop  
  
    If blnFlag = True Then  
        MsgBox "Found it.", vbOKOnly, "Result..."  
    Else  
        MsgBox "Didn't find it.", vbOKOnly, "Result..."  
    End If  
  
End Sub
```

Along with the Exit Do to jump out of the loop, I've used a new variable type called Boolean. A Boolean variable can hold only values of true or false. I started the variable as false meaning that I didn't find the number. If I find the number I change it to true indicating I found it. If I get to the end of the list without finding the number it stays false. So, once outside the loop I use the value in the Boolean variable to decide whether the number was found. The use of a variable in this way is known as a flag. The variable flags which message to display.

Open a workbook and try this example before continuing on.

All right so once you've tried that let's look at another looping structure. Click the [FOR...NEXT LOOP](#) page on the right.

# FOR...NEXT LOOP

October-28-11  
8:31 PM

Another looping option allows us control our loop's number of repetitions much more precisely. This type of loop is known as a For...Next loop. It is best used when you know how many times to repeat the loop or you can use numbers to control your loop.

Just like the Do...Loop it requires a condition to know when to quit. The exception is that this condition actually tells you when it will end.

**For** *counter = value to value*  
*code to execute*  
**Next** *counter*

The counter is a variable that will hold the current value. The values are the lower and upper limits of the number of times through the loop.

For example, let's modify the code that writes the numbers 1-10 to the sheet. Instead of the do...loop let's use the for loop.

```
Sub for_loop()  
  
    Dim intX As Integer  
    Range("A2").Activate 'make A1 the active cell  
  
    For intX = 1 To 10  
  
        ActiveCell.Value = intX 'add the value to the active cell  
  
        'Move down a one cell  
        ActiveCell.Offset(1, 0).Range("A1").Select  
  
    Next intX  
End Sub
```

Notice we no longer have to manage the increasing of the variable by one each time through. The loop is taking care of that for us. Each time we return to the top of the loop we do two things: increase the counter by one, and then make sure that it is still less than the upper limit.

We can change the increment (or decrement) by adding the step command to the first line.

For intX = 2 to 20 Step 2

Means to count to 20 by 2's.

Or:

For intX = 20 to 2 Step -2

Means count down to 2 by decreasing the value by 2 each time.

And, just like the do...loop we can prematurely exit a for loop using the EXIT FOR statement. when executed, the code picks up at the first line after the Next line.

Enter this code and test it.

When done let's try some practice files.

Click on the [Practice Assignments](#) page on the right.

# Practice Assignments

October-31-11

4:22 PM

1. Open a new workbook and on sheet 1 write a program that uses a do loop to calculate the factorial of a number that is provided by the user. Output the value back to the user. Add a button to execute the code.
2. On the same worksheet write a second program that uses the for loop to calculate the factorial of a number that is provided by the user. Output the value back to the user. Add a button to execute the code.
3. On sheet 2 write a program that will use a do loop to generate the odd numbers from 1-11 and place them in cells A1-A6. Add a button to run the code.
4. Also on sheet 2 write a similar program to that in question 3 except use a for loop. Add a button to run the code.

Okay, now let's try something a bit more complicated. Click on the [Assignment](#) page to the right to challenge yourself.

# Assignment

October-31-11  
4:25 PM

In this assignment you will use a nested loop to create your own sorting application.

In this assignment you will create a list of numbers in the cell range A1:A10 and then write the code that will sort those numbers.

For this assignment you will write what is known as a bubble sort. The bubble sort is a simple sorting algorithm. It isn't the most efficient sorting algorithm, but it is effective.

The bubble sort operates simply by comparing the first two numbers in the list. If the second number is lower than the first, the two numbers are switched. You continue to the bottom of the list making switches as necessary. When you get to the bottom you check to see if you made any switches on the way down. If a switch was made, you have to return to the top of the list and run the checks again until you can get to the bottom of the list without making a switch. If no changes were made (meaning the numbers are in order), you can stop.

To see a flowchart representation of what a bubble sort looks like see the next page. The items in red are part of the outer loop and the green are part of the inner loop.

Hints:

You will notice two decisions are the same (Is the next cell empty?). The first one is an if statement inside the inner loop which uses the Exit Do option to leave the loop if the next cell is empty. The second is the condition for the inner loop.

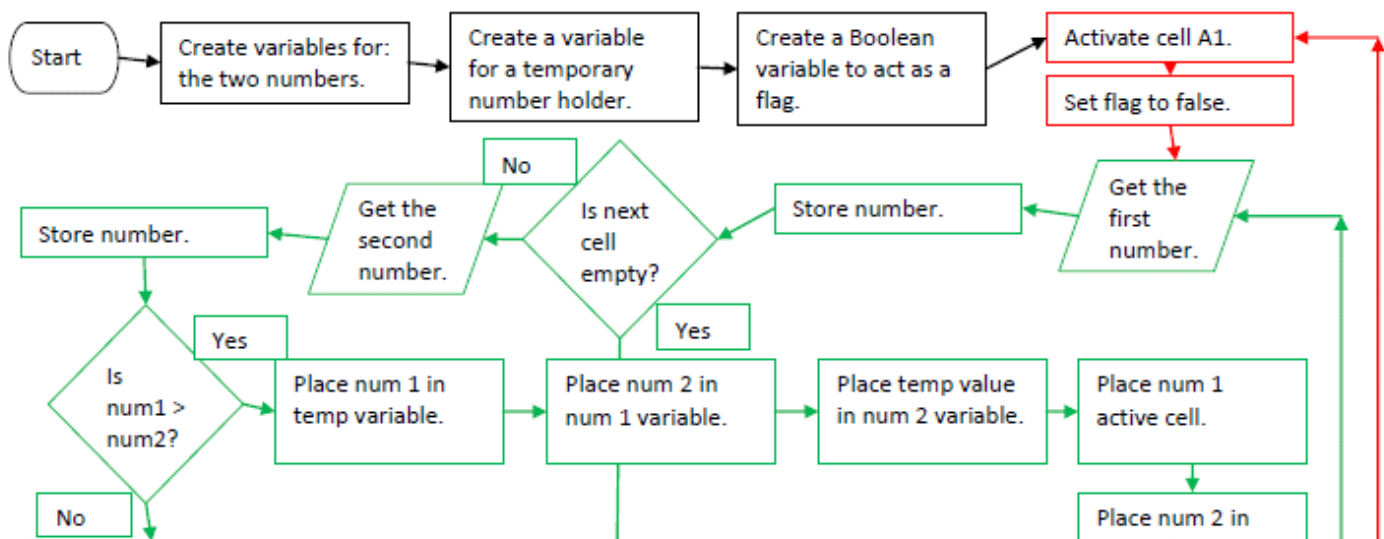
To reference the value in the active cell (the first number), you will use `ActiveCell.Value`.

To reference the next cell (the second number) down you will use `ActiveCell.Offset(1,0).Value`.

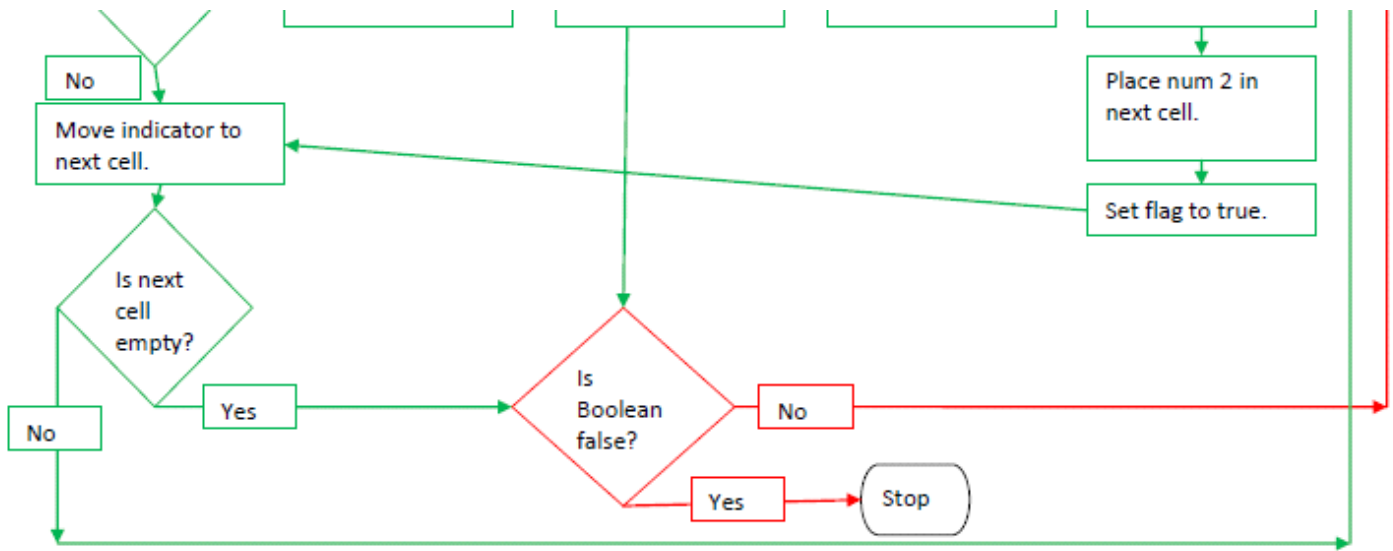
To move to the next cell you will use `ActiveCell.Offset(1,0).Range("A1").Activate`.

To test the next cell to see if it is empty, you can use the code `If ActiveCell.Offset(1,0).Value = Empty` then.

Create a sub procedure that will use this bubble sort to sort your list of numbers.







When done, click on the [Arrays](#) tab at the top of the page to continue.

# Single Dimensional Arrays

October-28-11  
8:31 PM

It is inefficient to create a large number of variables, especially if the data that you want to store is related.

Write a program that will calculate the area of a triangle of height 3.0 cm and base 3.0 cm as the base is increased by a factor of 10% ( $A = 1/2BH$ ). Run the calculations for ten increases. You will store these numbers temporarily for future use.

In a case like this you would need ten separate variables, one for each of the new values. You would have to have a system in place for knowing which was the first number generated. And, you would have to manage future code so that it would know which of the variables to use for which calculation.

VBA provides us with a construct that will handle situations like this. An *array* is a special variable. It has a name like other variables, but it also has “slots” or “pockets” that can hold a number of different values. Each “slot” is given a number (called the index) according to its position in the array. This means that knowing the array name and the slot number you want to use, you can access the values.

Defining an array is the same as defining any other variable, except you have to decide how many slots you want and tell VBA.

For example:

```
Dim array_name(number_of_slots) As data_type
```

So, as in our example, we could create a usable array such as:

```
Dim arrAreas(10) As Double
```

Now we can simply use the array like any other variable. We can assign values to it, just like we did with other variables:

```
arrAreas(1) = 10
```

This will place the value of 10 into array position 1.

That's fine but in our case that really doesn't help us much as we would have to still run the calculations on ten separate lines and do ten separate assignments. The power of the array really becomes visible when we combine them with loops. Here we'll use a for loop. We can use the for loops counter as our index number allowing use to move through the entire array one slot at a time.

Check the code below for the completed example:

```
(General) Areas
Sub Areas()
    Dim arrAreas(10) As Double 'my array to hold the values
    Dim dblCalcArea As Double 'the new area to use in the calculations
    Const cnstHeight = 3# 'the height of the circle
    Dim intX As Integer 'my for loop's counter

    dblCalcArea = 3# 'assign the starting value

    For intX = 0 To 9

        arrAreas(intX) = 0.5 * cnstHeight * dblCalcArea 'run the calculation
        dblCalcArea = dblCalcArea * 1.1 'increase the size of the base

    Next intX

    'for simplicity sake I'll output the values back to the worksheet
    Range("A1").Activate
    For intX = 0 To 9
        ActiveCell.Value = arrAreas(intX)
        ActiveCell.Offset(1, 0).Range("A1").Activate
    Next intX
End Sub
```

The first thing to notice is how I am counting the loop. My loops all start at 0 and go to 9. Well, in arrays, by default in VBA, the first position is numbered 0, not 1. And if I want 10 numbers I have to go from 0 to 9.

The line to be focused on is the following:

`arrAreas(intX) = 0.5 * cnstHeight * dblCalcArea`

The first time through the loop, the part to the left of the equals sign will look like `arrAreas(0)` which points to the first element or position in the array. The next time through it would be `arrAreas(1)` and so on. The loop helps us move through the array's elements in order.

In memory it would look like this:

Position	0	1	2	3	4	5	6	7	8	9
Value	4.5	4.95	5.445	5.9895	6.58845	7.247295	7.972025	8.769227	9.64615	10.6106

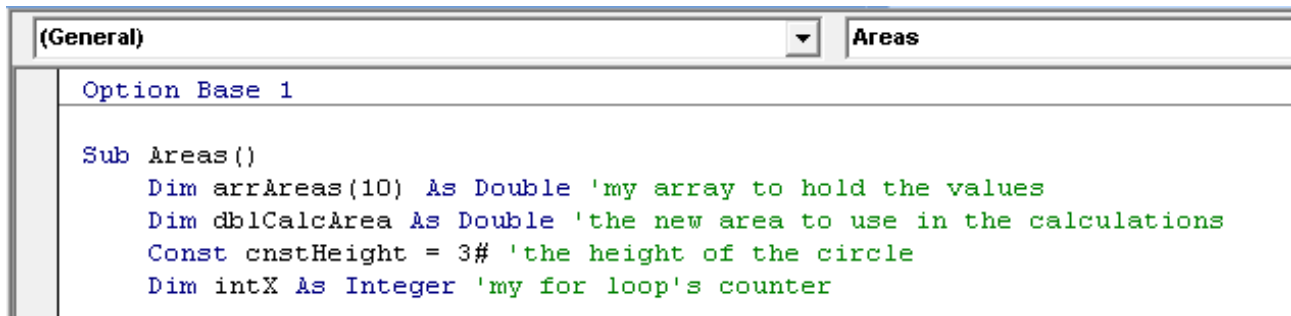
To move on, click the [Changing an Array's Starting Position](#) to see how to start at position 1 rather than 0.

# Changing an Array's Starting Position

October-31-11

4:44 PM

By default an array will start with a first position of 0. If you prefer, you can change the starting position by adding the line *Option Base 1* at the top of your code window. That will tell VBA to use 1 instead of 0 as the first position in the array.



```
(General) Areas
Option Base 1

Sub Areas()
    Dim arrAreas(10) As Double 'my array to hold the values
    Dim dblCalcArea As Double 'the new area to use in the calculations
    Const cnstHeight = 3# 'the height of the circle
    Dim intX As Integer 'my for loop's counter
```

Click on the [Subscripts Out of Range](#) to see the most common error when working with arrays.

# Subscripts Out of Range

October-31-11

4:45 PM

When you're working with arrays, this is going to be the one error you are going to see a lot. All it means is that your program is trying to reference an array position that is beyond the bounds of the current array limits.

For example you declare an array of ten items starting at position 1 (Option Base 1). You then try to access position 0. The subscript or index number is beyond the bounds of the array that has positions 1 through 10. There are no easy fixes for this and it might take some time to find out how you managed to get the subscript beyond the boundaries of the array.

Now let's see about extending our arrays. Click on the [Multi-dimensional Array](#) page on the right.

# Multi-dimensional Arrays

October-31-11  
4:52 PM

You can expand on arrays so that they will hold more than simply one list of data. They can actually hold tables of data. A multi-dimensional array sets up a table of values in memory that you can access much like you can an Excel spreadsheet. Each cell has a row number and column number. You need both the row number and the column number to access that particular array position.

For example I create a multi-dimensional array as follows:

**Dim *array\_name*(*row,column*) As *Integer***

So an array of 2 rows and 2 columns would be defined as:

Dim arrTest(2,2) as Integer

And look like:

	0	1
0	0,0	0,1
1	1,0	1,1

Now you can work with your data much like you would with any other table. Working with multi-dimensional arrays will usually require the use of a nested loop to get all the values into the proper cells.

Now let's try one practice assignment. Click [Practice Assignment](#) on the right to give it a try.

# Practice Assignment

October-31-11

4:56 PM

Write a program that will firstly accept a double value from the user (your choice of methods). Generate 10 numbers and store them in an array using the value provided by the user in the following formula:

$$3x/(1+x)$$

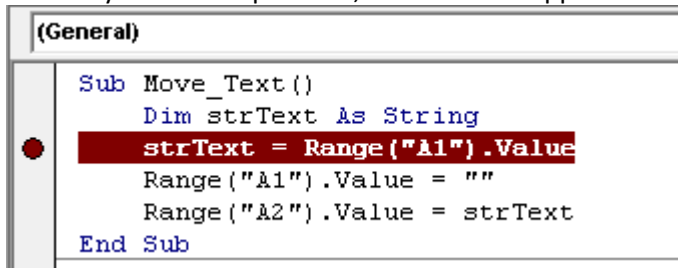
where the user's value is x. Increase the user's value by 1 each time through the loop. Display the results contained in the even numbered array positions (starting at position 0).

# Built-in Debugging Tools

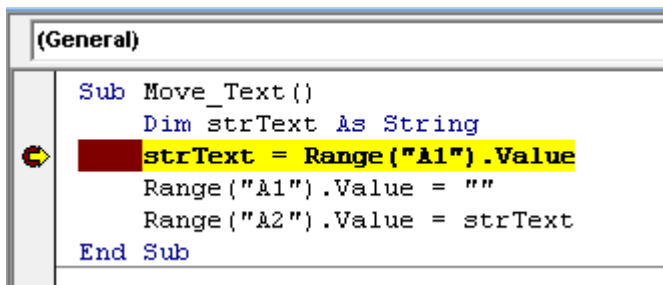
October-28-11  
8:32 PM

When you run into logic errors it is sometimes really hard to find where you went wrong. VBA provides you with a way to help debug your code. It will allow you to pause your program and then check the values contained in each variable. It will then allow you to move through your code one line at a time pausing to let you check your values along the way.

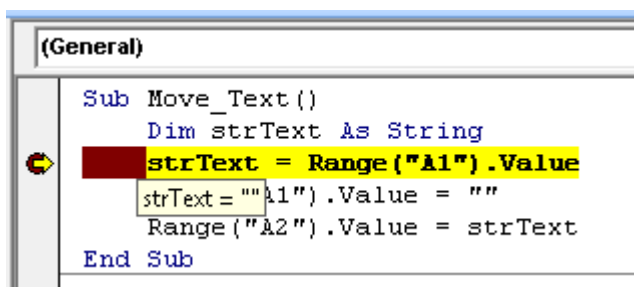
To start, you have to pick which line of code you want to pause at. Review your code and when you've decided you'll notice a grey bar along the left hand side of code window. If you click on this bar beside the line you want to pause at, a red dot will appear. See the image below:



Now when you run the program the execution will stop on this line. The line will be highlighted in yellow.



If you hover your mouse over a variable name you will see a tool tip indicating the current value contained in the variable.



This allows you to track your values through your programs.

To move to the next line press F8.

Just remember to remove the red dot by clicking on it a second time when you're done.



# Debugging Tips

October-31-11

5:00 PM

- If you're unsure of what a variable holds during program execution simply use a message box to output your values so you can check them.
- Set up a test bank of values. Have a set of values that you already know the answer to that you can input into your program and see the results.

# Answers

November-04-11  
12:29 AM



Manual