Starting with Visual Basic for Applications in Excel

by Maudibe

Introduction:

It has become obvious to many that Excel, although an extremely powerful program, has limitations. To achieve routine and more difficult tasks, VBA for Excel has stimulated much interest but many do not know how or where to start. The purpose here is to show the beginner what is readily available to him/her and to get them off to a running start. I will explain in the most simplest steps how to set things in motion. We will oncentrate more on the flow rather than the actual Code. Learning to code will come later.

VB interface

While in excel, every programmer already has all the tools they need. In this section, we will review how to launch the VB interface and cover different aspects. There are nuances between different versions but to the contrary, there are much more similarities. I will cover versions 2003 and 2010. For those who have Office 2007, there is minimal difference between 2007 and 2010, so the 2010 descriptions will apply to you.

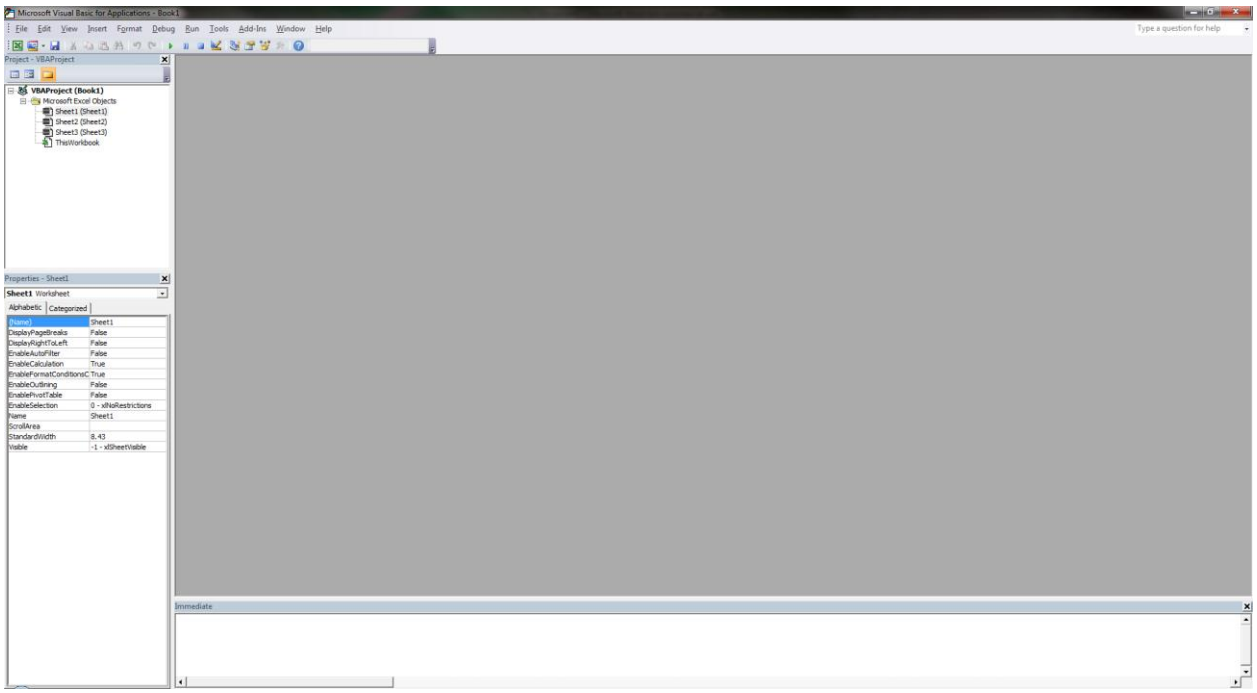
First, let's open the VB interface: (shortcut- Alt+F11)

2003: Tools> Macro> Visual Basic Editor

2010: The Developer tab must be visible on the ribbon. If it is not, Click on File> Options> Customize Ribbon. Move The Developer tab from the Left column to the right then click OK. The Developer Tab should now be visible on the Ribbon.

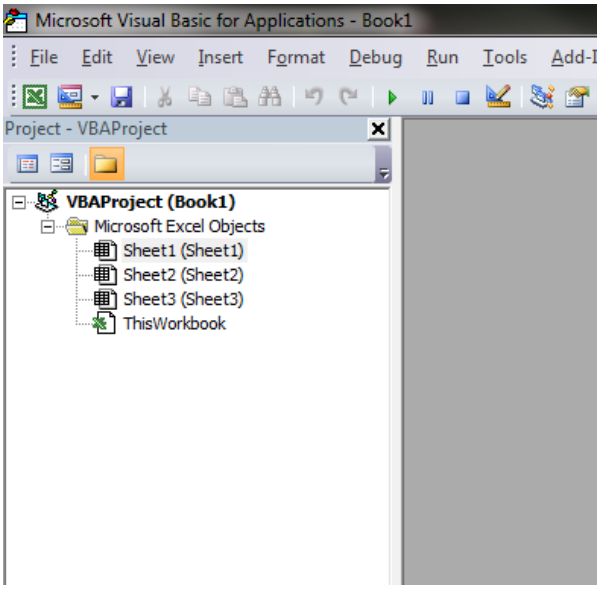
Developer tab> Visual Basic

The visual Basic editor is very similar for all versions. Below is a screen shot of the 2010 editor.

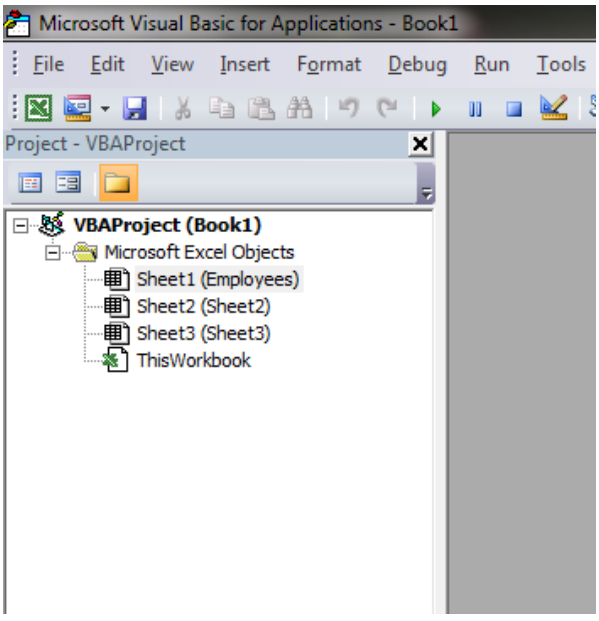


Let's take a close look at the different segments:

The Project Explorer window



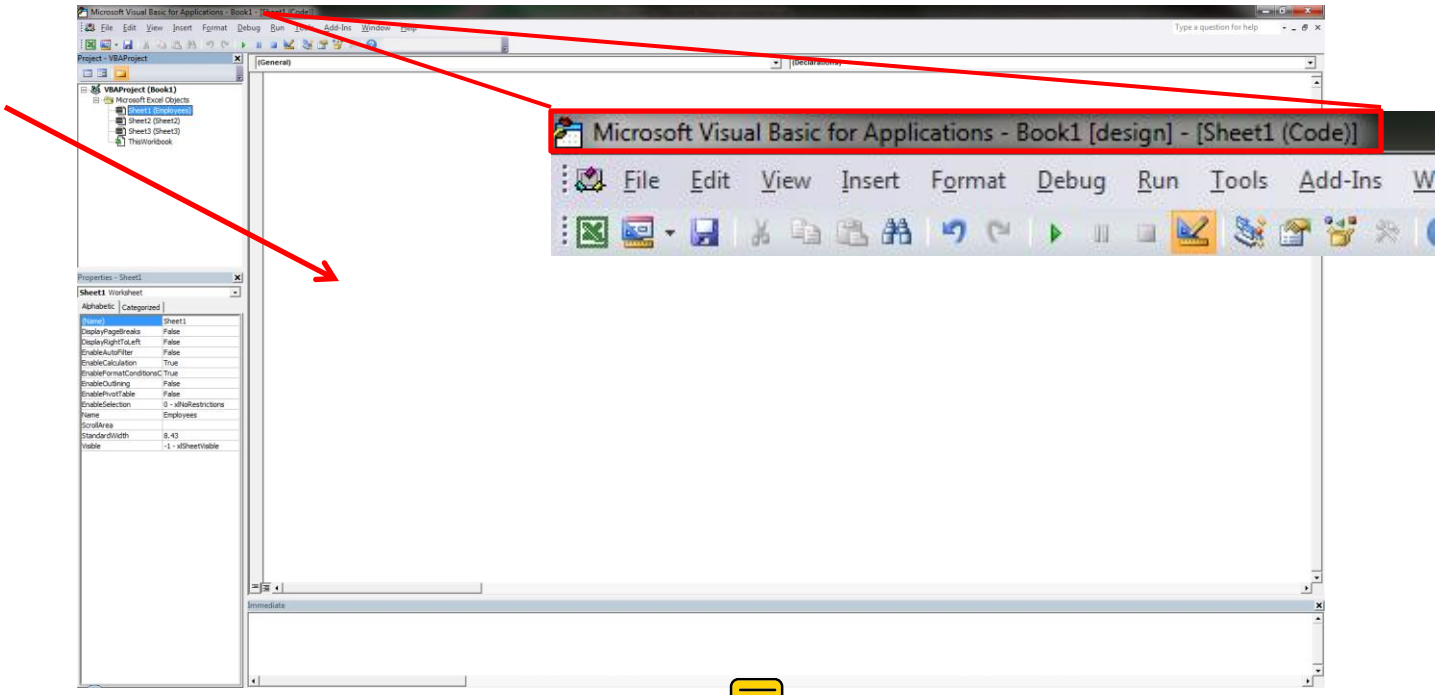
By default, when you start Excel, it starts with 3 available blank sheets. This can be changed in options. But forever how many sheets Excel opens with, there will be a corresponding sheet objects in the Project Explorer Window plus a workbook object called ThisWorkbook. The names of the sheets are in parentheses while the sheet# to the left of it just lets you know sheet index. If you were to change the name of Sheet1 (by right clicking the sheet's tab in spreadsheet view then Rename) to Employees, then in the Project Explorer Window Sheet1 would appear as:



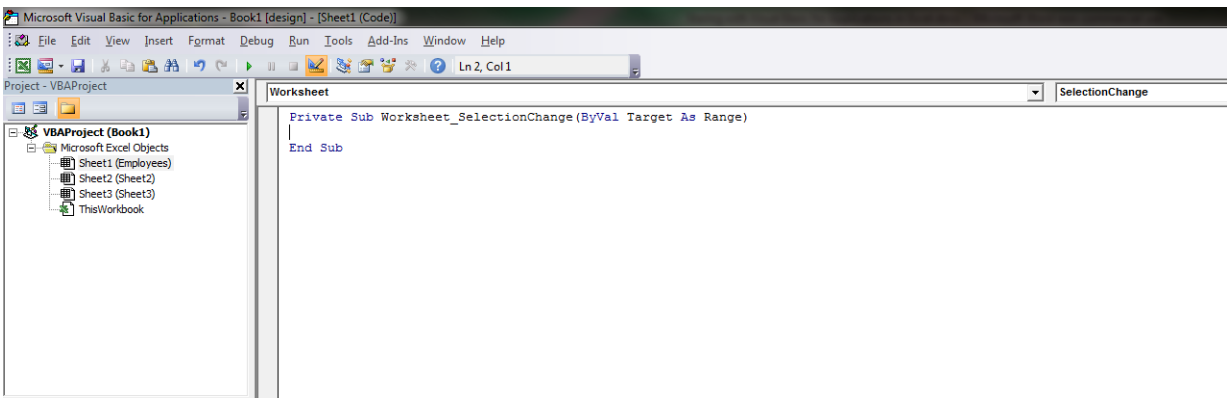
The Sheet can be reference by either the sheet# or the Sheet name.

The Code Window

Double clicking a sheet will open the Code window for that sheet. The Title Bar at top will indicate which code sheet you are viewing. Below, I have double clicked Employees and the code sheet has opened to the right of the project window.



This is where code pertaining only to that sheet will reside. If there are any objects placed on that sheet, code for the objects will be accessed or written here as well. If you click on the dropdown box at the top of the code window, you will see two headings: General and Worksheet. General will be listed initially by default when the code window opens. It is a specific section at the top of the code window where variables will be declared. I will explain in detail later what that means but for now, remember that this is not where code to do something is written. To the right of this drop box is a second drop box that tells you what you can do in the section. You will notice that only Declarations is available with General. The second heading is Worksheet. If you select it, you will notice a blank subroutine open in the code window called Worksheet_SelectionChange . This blank routine opens by default.

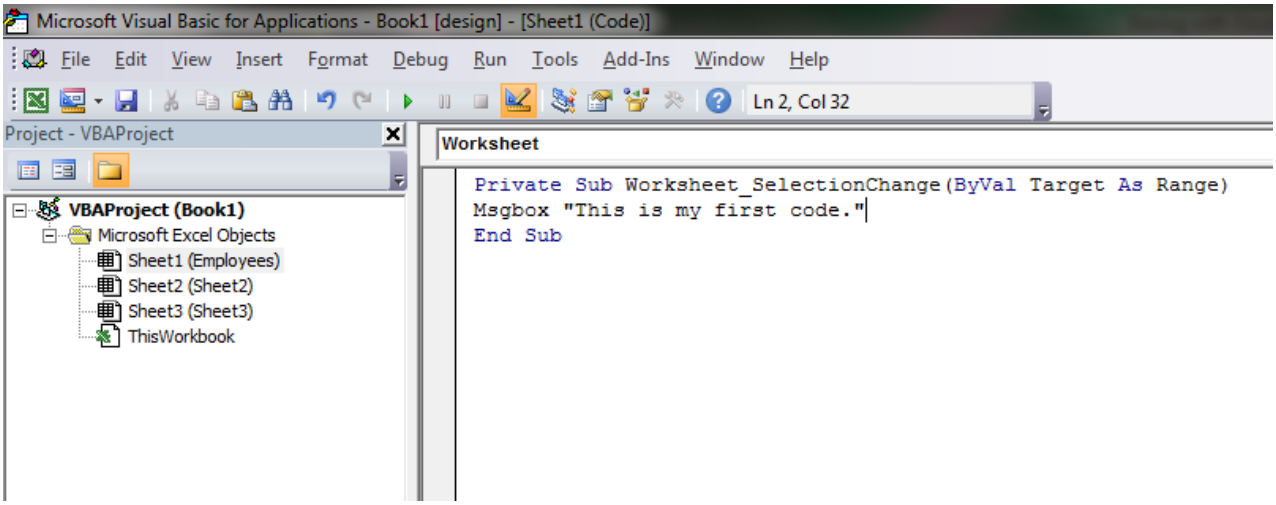


This subroutine is an event subroutine and is triggered whenever you select a new cell on that spreadsheet. An event is a specific action that initiates specific response. Examples of events are: opening or closing the sheet by clicking on the sheet tabs, switching cells, right clicking on a cell, etc.

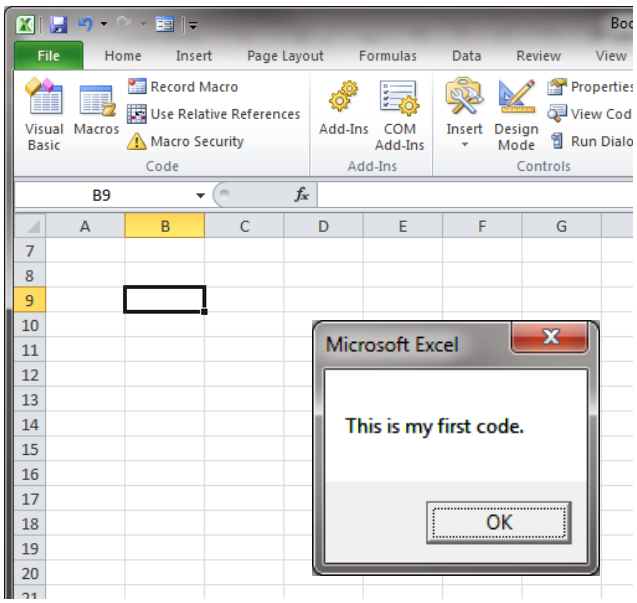
Since there is no code written in this subroutine, nothing happens when you select a new cell on the spreadsheet. If there were code written here, every time you select a new cell, this event would fire and the code would be executed. Try pasting the following line of code in between the Sub heading and the End Sub statement:

Msgbox “This is my first code.”

The result should look like this:

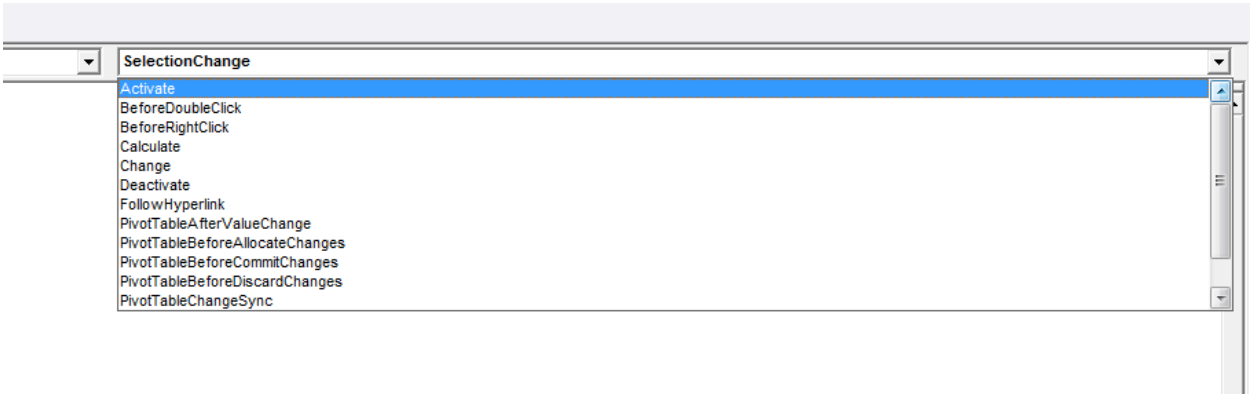


Now go back to the spreadsheet (taskbar) and click on a cell. You should see a pop up box with the message that you wrote.

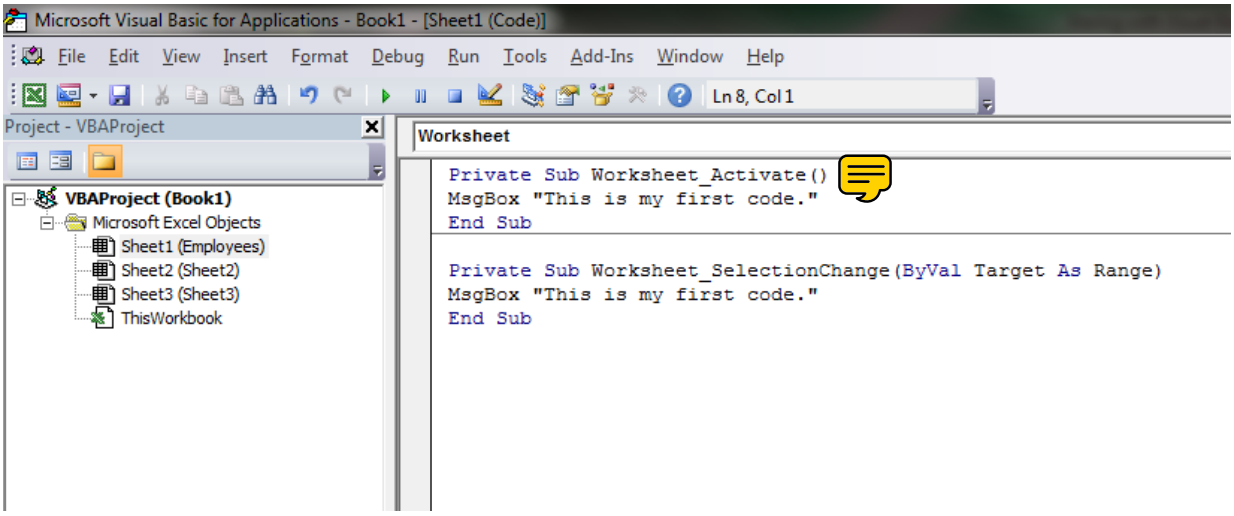


Click OK and the pop up message box disappears and the code ends. Click another cell and the same thing happens. It is obvious that you must be judicious what you write in this routine because any code you write here will be executed each time the user makes a new selection. While you are in the Excel view, click on Sheet 2. Select cells on that sheet and you will notice that the pop up message box does not appear. That's because you wrote the code on Sheet1 (Employee) code sheet and it will only be executed for that sheet only.

Switch back to the Visual basic window (taskbar). With Worksheet selected in the left drop down box, in the right drop box you will notice that you have other available events. These are all events related to the worksheet. Selecting one of these events will open a new blank subroutine for that event.



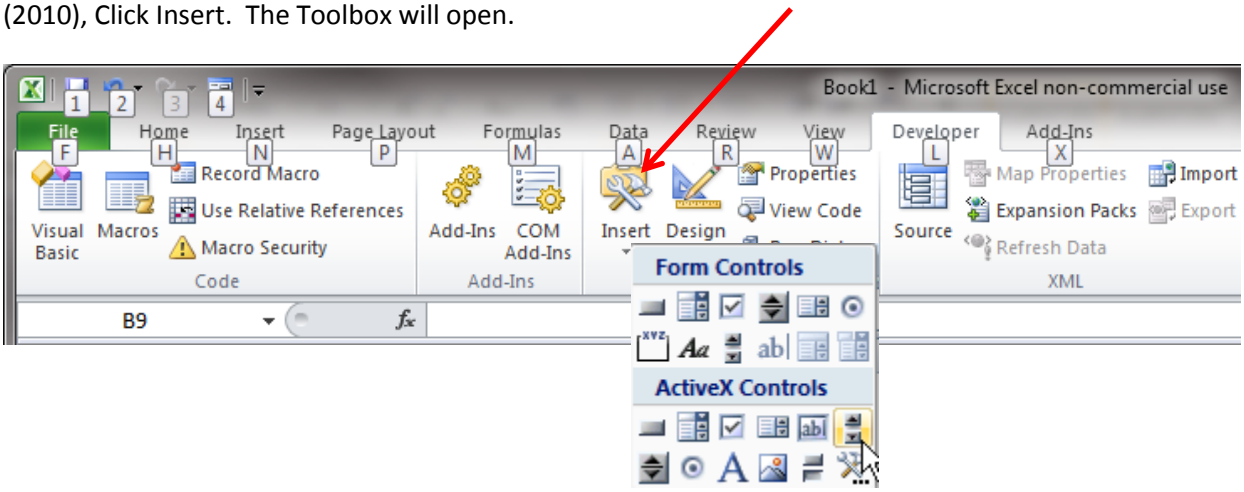
Click on Activate. A new Subroutine for Active opens. Paste your previous line of code into the routine so it looks like this:



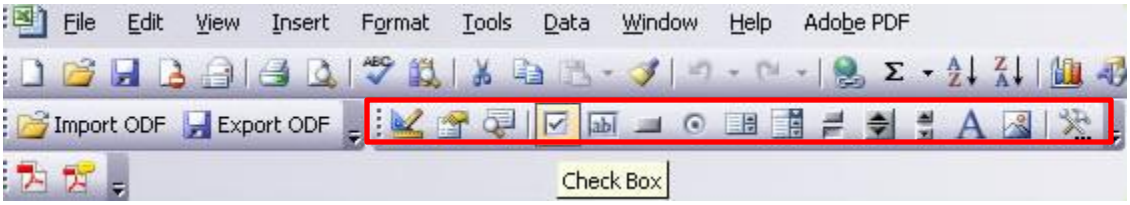
Now go back into the spreadsheet and switch sheets using the tabs. You will see the pop up message box appear every time you open Employee sheet. While you are on that sheet, switch cells. The Worksheet_SelectionChange event still fires as well. You can have only one routine for each event but the routine can have as much code (instructions) as you want.

I hope you are getting the idea of what an event is and how the event executes code written for that specific event.

Let's take the next step and add a control to the spreadsheet. A control is an object that can do many things depending on what type of control it is. From within the spreadsheet on the Developer tab (2010), Click Insert. The Toolbox will open.

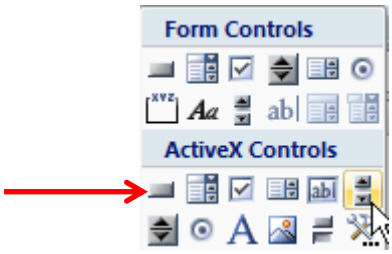


In 2003, you need to dock the Control Tool bar: View> Toolbars> Make sure Control Toolbox is checked



To add a control, select on one of the controls. In this example, we will add a command button to the sheet and apply code to that button.

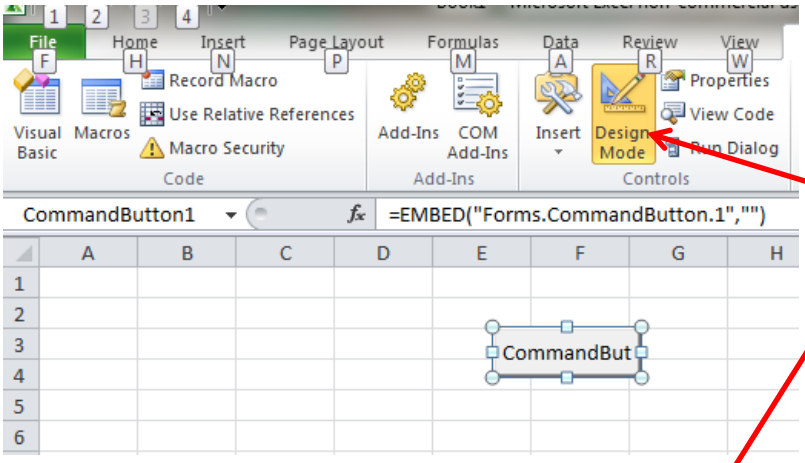
In 2010, select the ActiveX Controls Commandbutton.



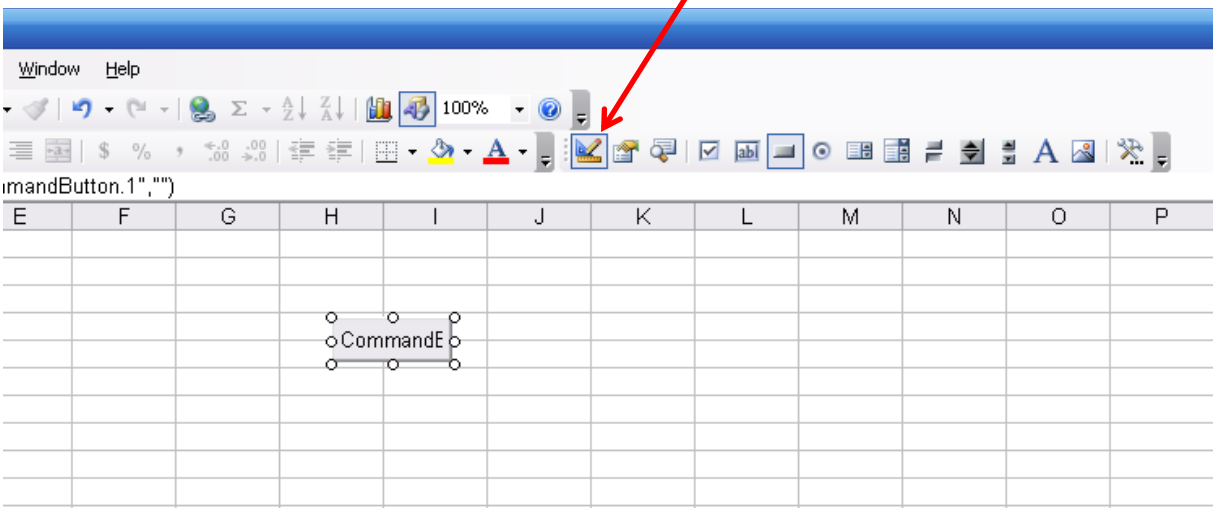
In 2003, select the Commandbutton.

When you select a control, you will see the cursor change to a cross hair and the Design Mode icon become highlighted. Drag and release the crosshair anywhere on the sheet to draw the commandbutton. The process is the same for drawing any of the other controls.

2010:



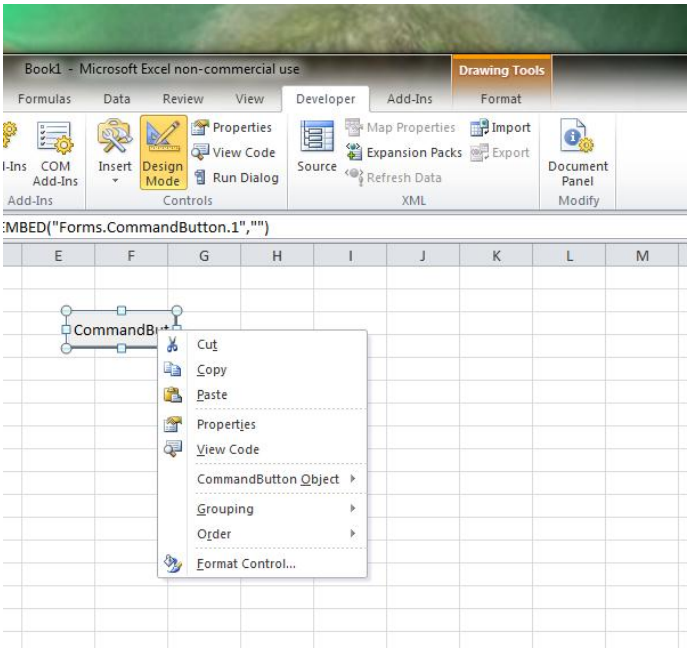
2003:

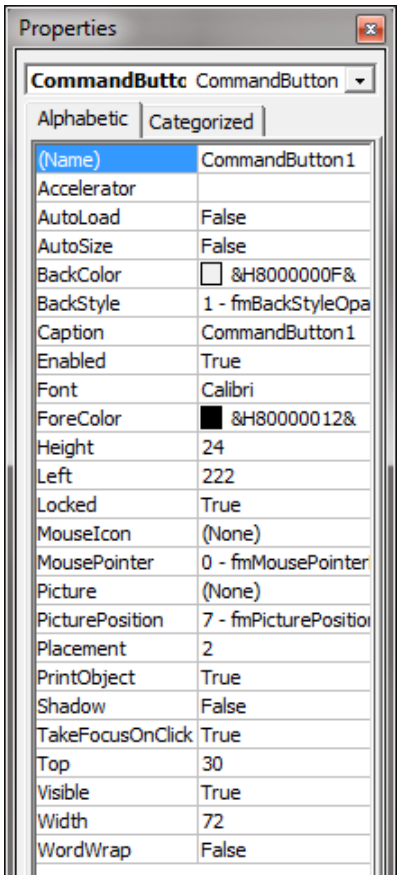


Design Mode:

While in design mode, the programmer can manipulate the control without accidentally activating the code attached to it. If this was not available and the user wanted to do something with the control, ex. move it to the right, the code would fire as soon as it was clicked upon. So while in design mode, the position can be changed. The size of the control can also be changed. Or the programmer can right click the control to view additional actions to perform on the control.

Right click on the control and select properties. A properties window will appear. A property is an attribute that the object has: height, width, color, caption, name, etc., similar to us humans or any object.

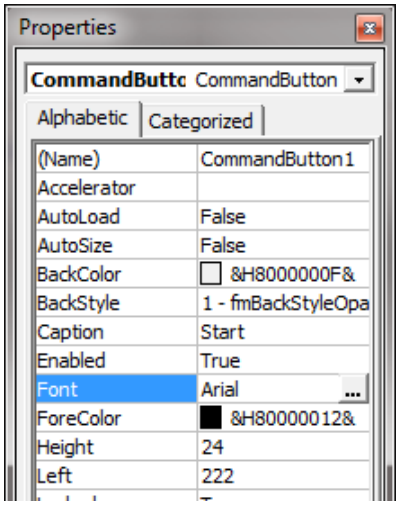


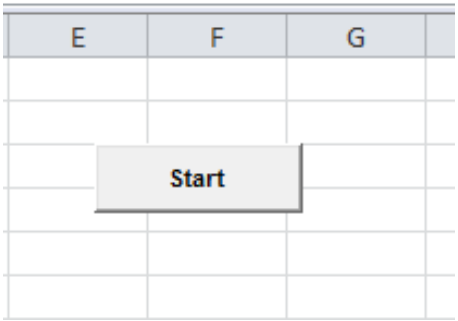


Here is listed all the properties of the control that can be changed. Let's make some changes.

1. Click on the slot to the right of Caption and enter "Start" (without the quotes)
2. Click on the slot to the right of font. And ellipsis will appear. Click on it and change the font to Arial 8 bold.

It should look like this:



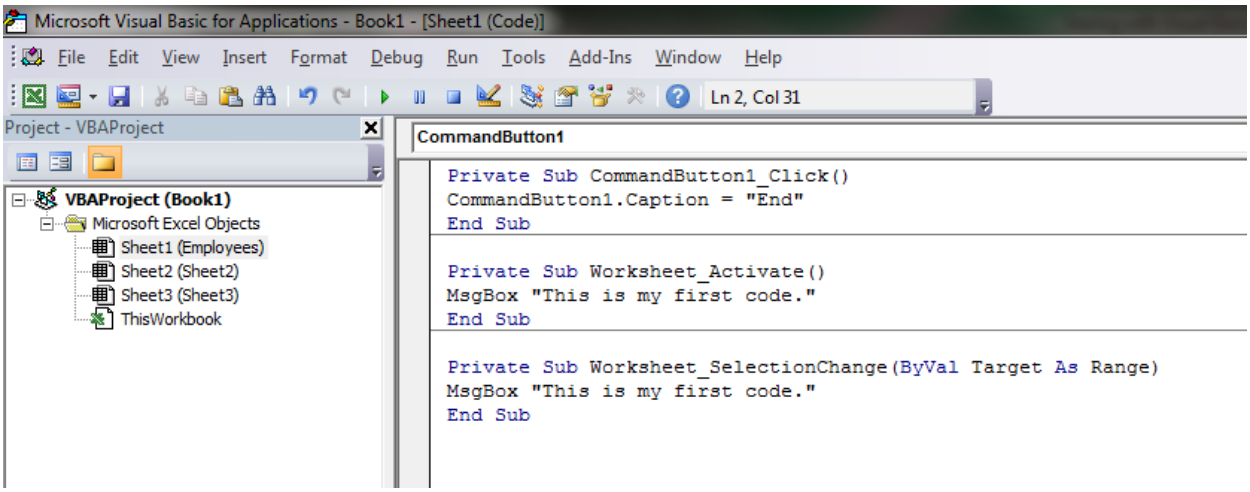


You can experiment with changing different properties to see the effect. If you highlight a property and press F1, a help screen will appear to give you more detail information of the property. Most of the properties can be changed by writing code as well as through the Properties Window, and this is the additional information that the Help screen gives. For example, you click a button and its caption toggles to a different caption. Let's write your second piece of code.

Assuming you placed the commandbutton on the sheet, open the VB Editor window (taskbar). At the top of the code window for the first sheet Employees, look at the dropdown box on the left side and you will notice something new. It's the commandbutton that you created. Why is it there? We said earlier that the code window for a sheet is: a place for declaration of variables, code for the sheet, and code for any controls placed on the sheet. Click on the commandbutton1 and the default event code will open for a commandbutton. It is a blank event subroutine called CommandButton1_Click(). If you click on the right drop down box, you will see all the other events associated with the control. Each of those events when fired can have its own separate code that makes something happen. Let's write your third code. Copy/paste the following line of code into the subroutine:

CommandButton1.caption = "End"

It should look like this:



Go back to the excel spreadsheet and make sure we are no longer in design mode by deselecting the design mode icon (no longer highlighted). Click the button and see what happens. The start caption changes from "Start" to "End"

Can I change the name of the button to something that will clue me to its function? Sure can and advisably so. Go back to the spreadsheet, go into design mode, and right click the control, select properties. Change the name (not caption) to something like StartButton. There can be no spaces in the names for objects or you will receive an error message if you attempt to do so

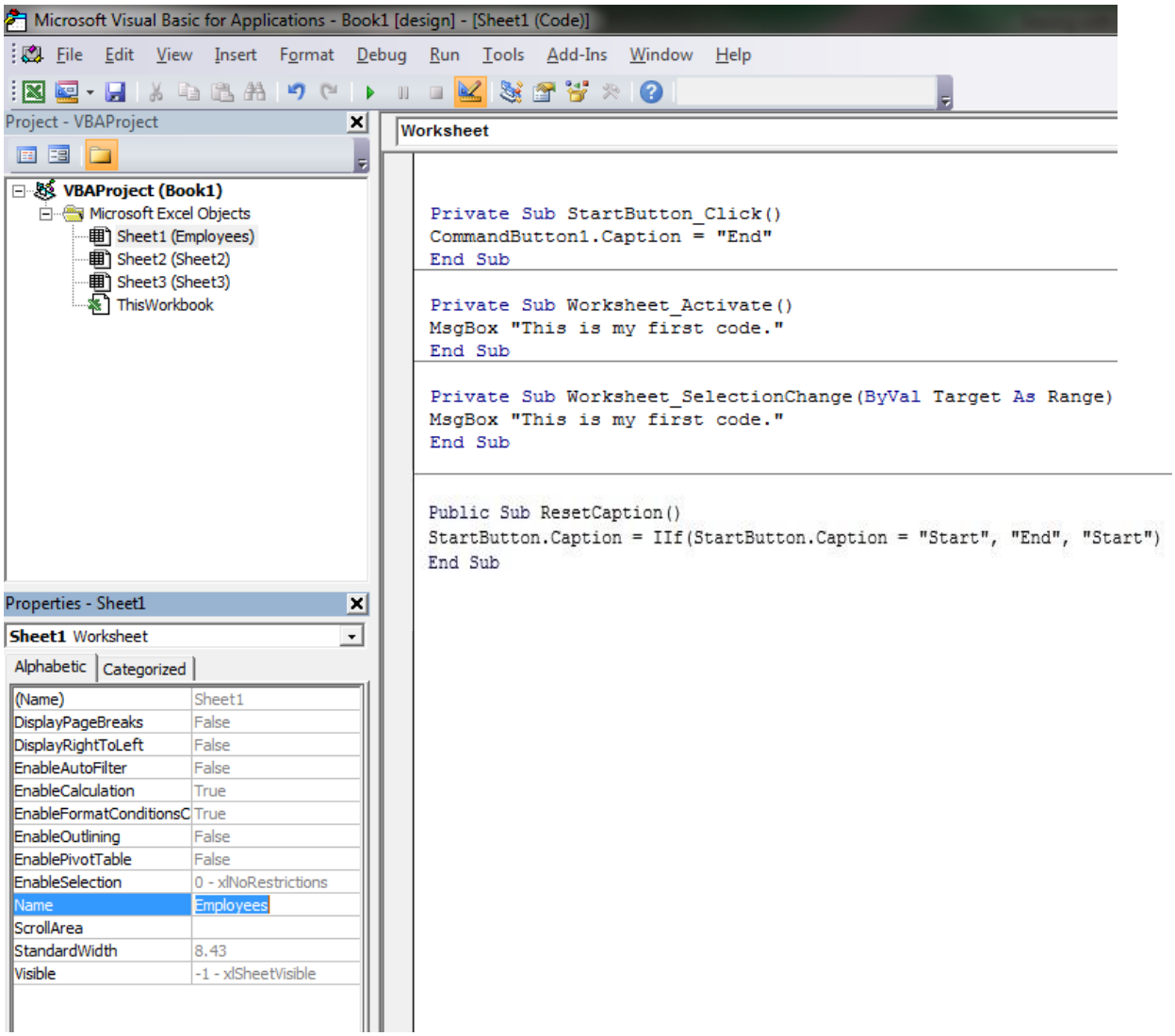
I changed the name but the code no longer works. What happened? Go back into the VB editor and look at the left dropdown box at the top of the code window. CommandButton1 is no longer there but

StartButton is. The code that you had written previously is for an object that no longer exists. If you click StartButton, a default blank event subroutine will open. You can copy your previous code into that subroutine. You do not need to remove the CommandButton1 subroutine as it will not be called upon, however, it would be sloppy programming to keep it. Just highlight the entire subroutine and press delete. Also, keep in mind that the code we wrote to change the button's caption from "Start" to "End" may still be running correctly, however, since the caption is already "End", we may not see any visible results occurring. Change the caption back to "Start" then rerun the code by clicking the button. Here is a little code to do this automatically. (see below in a routine arbitrarily called "ResetCaption"

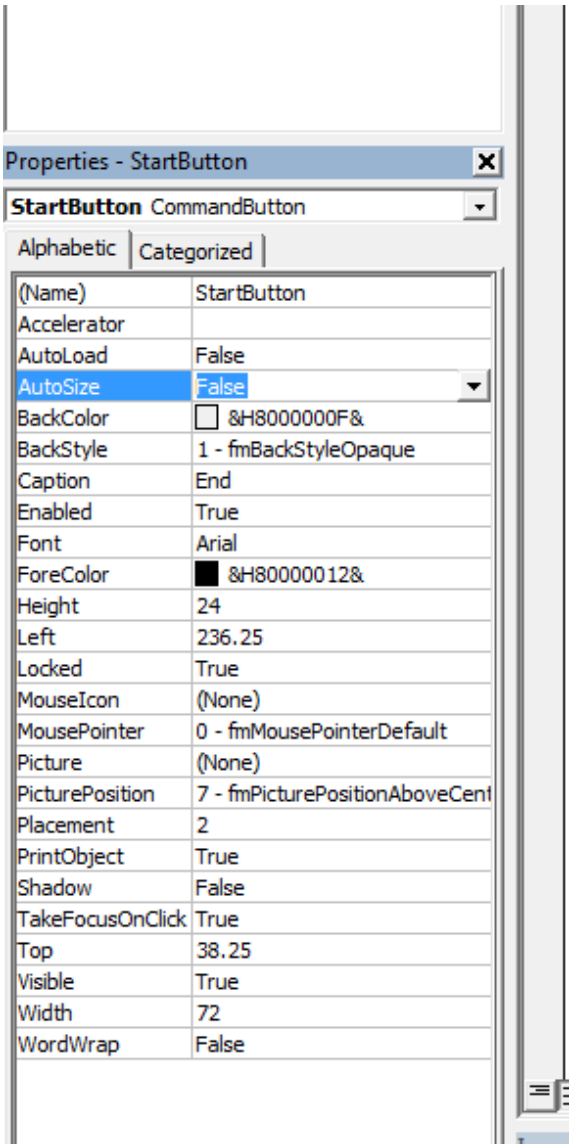
StartButton.Caption = IIf(StartButton.Caption="Start","End","Start")

The Properties Window:

We saw a properties window by right clicking a control on the spreadsheet while in Design Mode. But there is also a properties window within the VB editor. Go back to the VB editor and look below the Project Explorer Window. If the Properties window is not showing, then we must dock it. Click View> Properties window.



Try clicking the dropdown box in the properties window. Hey, look what's there….the StartButton. Click on it. Look familiar? It is the same Properties Window that you viewed for the button from within the spreadsheet but, you still have to be in design mode to view it or make changes.



Try changing the caption back to "Start" (without the quotes). See what happened on the spreadsheet. You can see by this point that having 2 monitors would be a huge advantage instead of switching back and forth between spreadsheet and VB editor views.

Dot notation

Recently in this tutorial you noticed a new format in the code CommandButton1.caption = "End". The dot (.) notation denotes that what follows is a property or method of the preceding object. In the above code, caption is a property of commandButton1. If you were to write a line of code on a sheet called Materials referencing a commandbutton called Search located on the sheet, you could just

reference it as Search.*property.* You do not need to specify the container which is the sheet or userform that is holding the collection of objects because it is understood. In the example above, since you are writing code for the Search button on the Materials sheet, Materials can be left out because it is understood. If, however, you were referencing the Search button from another Sheet or userform, you must include where to find the object. In this case the code would be:

Worksheets("Materials").Search.property.

Notice the dot that separates the components of the string. If a button was on a userform called Ingredients, you could reference it from another userform, worksheet, or module by:

Ingredients.Search.property

If you were writing the code from inside the Ingredients code window, you would not have to include the "Ingredients." prefix. Crudely, you could follow the rule:

Where.Who.What

Properties and Methods

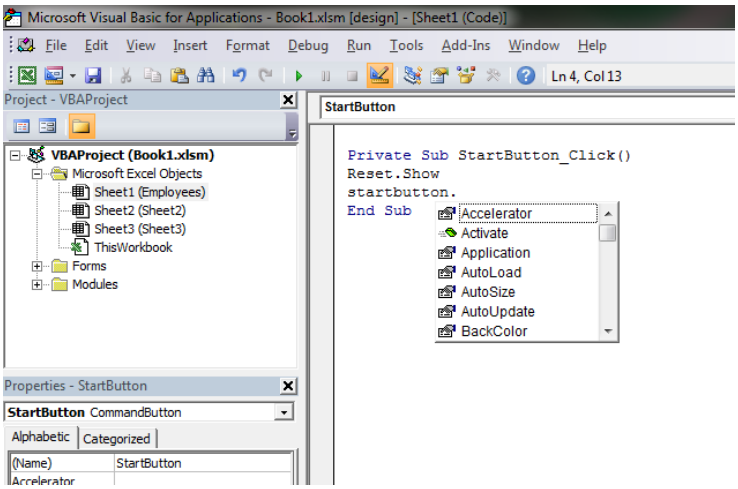
Up to this point we have been talking about properties of an object. Properties are adjectives that describe the object. You hair may be brown, you may be tall, or you may be handsome like me. These are all descriptions and controls have them as well. Hopefully, you have gotten a glimpse of them in the properties window. But controls can also have actions called methods. For example, an object can have the following methods: Activate, copy, paste, delete, select, send to back, update, etc. If you wanted to give a cell the focus, the select method could be applied.

Writing the code from within the Sheets code window:
Range("A1").select

Writing the code referencing the cells from a module, userform, or another worksheet's code window:
Worksheets("Materials").Range("A1").select

So how do you know what properties or methods are available for an object? Since Office 97, Microsoft has employed IntelliSense. When you type an object in the VB Editor followed by a dot (.), a dropdown box will give you a selection of available properties and methods. Select a property or method and it will automatically be placed after the dot. If you click within the wording of the property or method then press F1, a help screen will open giving you additional information on what it is, what it does, and how to use it. Instead, you may be provided with links documented elsewhere which will give the information you seek. This is a valuable learning tool.

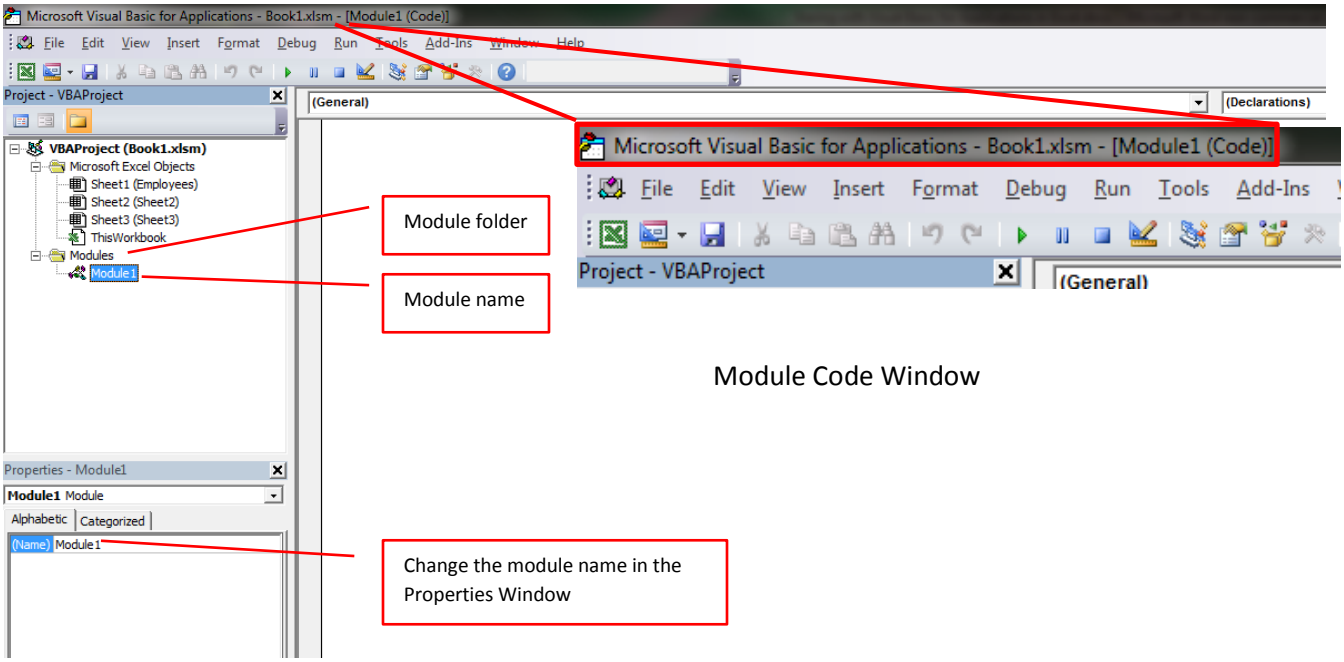
Note: This must be an existing object for IntelliSense to display the dropdown. There are objects referring to cells and/or worksheets that are built-in to VBA (Range object above), but they cannot reference cells on a non-existing sheet. Worksheets("Paycheck").Range("A1").Select will not have a dropdown because the sheet Paycheck does not exist. The same is for a control and a userform.



Modules

So, if the code window for a sheet should only contain code for the sheet and the controls on the spreadsheet, where does all the other coded subroutines go? "What other kind of code is there?", you ask. Well, let me give an example. Suppose you want to write code that sorts a column. You could place that code onto the code window for a sheet. But what if you wanted to be able to sort columns for all sheets? You would have to write the code on each sheet's code window. What if, in the future, you amend the code? You would have to amend every sheet. And if you had 10 sheets, it could get very daunting. So, the rule of thumb is, when you have a common code that needs to be accessed from anywhere in the workbook, place it in a module. OK…..what is a module? A module is a collection of subroutines or functions that are usually public, meaning they can be accessed from anywhere in the notebook. Where are the modules and how do I create one? Open the VB editor and click on the Insert Menu. You will see a choice between: Procedure, UserForm, Module, Class Module, or File. Click on Module. You can rename the module anything you want but a good practice is to create different modules that pertain to different aspects of your project. For example, you might create a module called Calculations which contains all subroutines or functions that perform calculations. You can create a second module that may perform cell manipulation such as sorting columns or rows, spell checking, truncating strings, etc. And maybe even a third that performs maintenance such as cell validation, save and close, or print procedures. When you first create a module, it will be given the default names Module1. Module2, etc., and placed within a Module folder. To change the name, click on the module. Below, in the Properties Window, you can change the name to anything without a space or special characters. An underscore, however, is OK to use. The name of the module is insignificant as the VB will find the routine as long as it is Public. However, having different modules for specific types of routines or functions makes it easier to import into a new project a group of procedures to do certain related things. For example, if I created a module called accounting which contained several accounting routines that I wrote, every time I write a new accounting spreadsheet, I would import the accounting module into it. The related routines are neatly bundled and easily accessible to any new project.

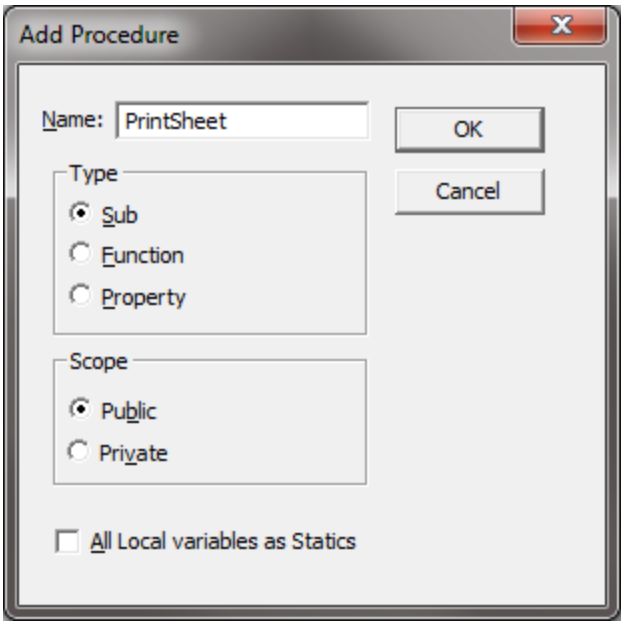
If you double click the module, a code window will open just like it did for the sheets. The module will display all the subroutines that live there. Right now, it will be blank but we will be adding some soon.



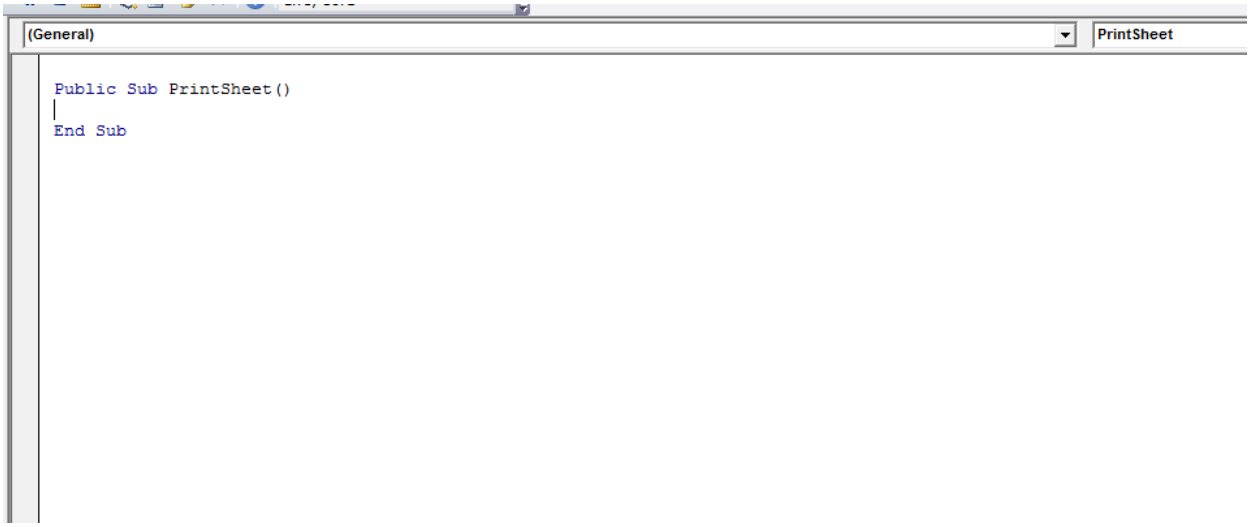
Module Code Window

In the left dropdown box you will see General only. On the right, there will be no selection available because the module contains no procedures at this time. Change the name of the module to Maintenance.

Let's create a subroutine that will printout a sheet when a button is clicked. Earlier, we created a commandbutton called StartButton. We will modify that code plus add additional code to a module to perform the process. Assuming you have already created a module called Module1, Let's change the name to Maintenance (described above). To create the subroutine, click back up on Insert Menu on the ribbon and select Procedure. The ellipses after the word procedure is standard Windows format that indicates that another dialogue box will open. The Add Procedure dialogue box opens and you must enter a name for the procedure. No spaces or special characters can be used except an underscore. It is good to give it a name that reflects its function. Let's call this procedure "PrintSheet" (without the quotes). Select either a sub (which is a subroutine) or a function. Do not bother with properties for now as this is a topic of another discussion. I will also explain the difference between a function and a subroutine later so for now, just select a sub. You can also choose the scope of the procedure as Private or Public. Private would indicate that the routine can only be accessible from within the module. Public, as mentioned before, can be accessed from anywhere in the workbook. Public is the Default Scope and is accessible from any open workbook! Public is only necessary if the Module is set to Private, via the Module Private directive at the top of the module, if you want to make a Sub or Function visible outside the module.



The Add Procedure dialogue box will close and the Module's code window will now have a blank routine with the name you gave it. Notice on the right drop down box there is an available routine called PrintSheet. The selections will increase the more routines added. We are now ready to add some code.



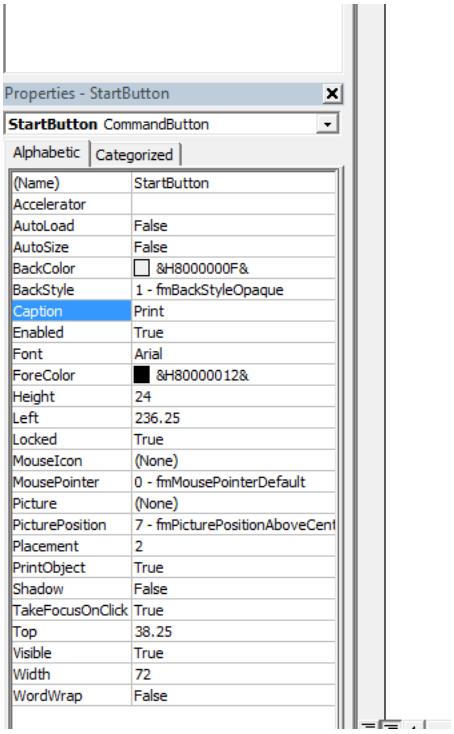
Copy and paste between the Subroutine's heading and End Sub statement the following line:

ActiveSheet.PrintOut

It should look like this:

So what does this routine do? When called, this routine will take whatever sheet is displayed (or active) then print it out. The line of code is a sentence in a specific format that the program understands (able to compile). The next question is, how do we make the code run? Easy, we attach it to the click event subroutine of the StartButton that we created earlier.

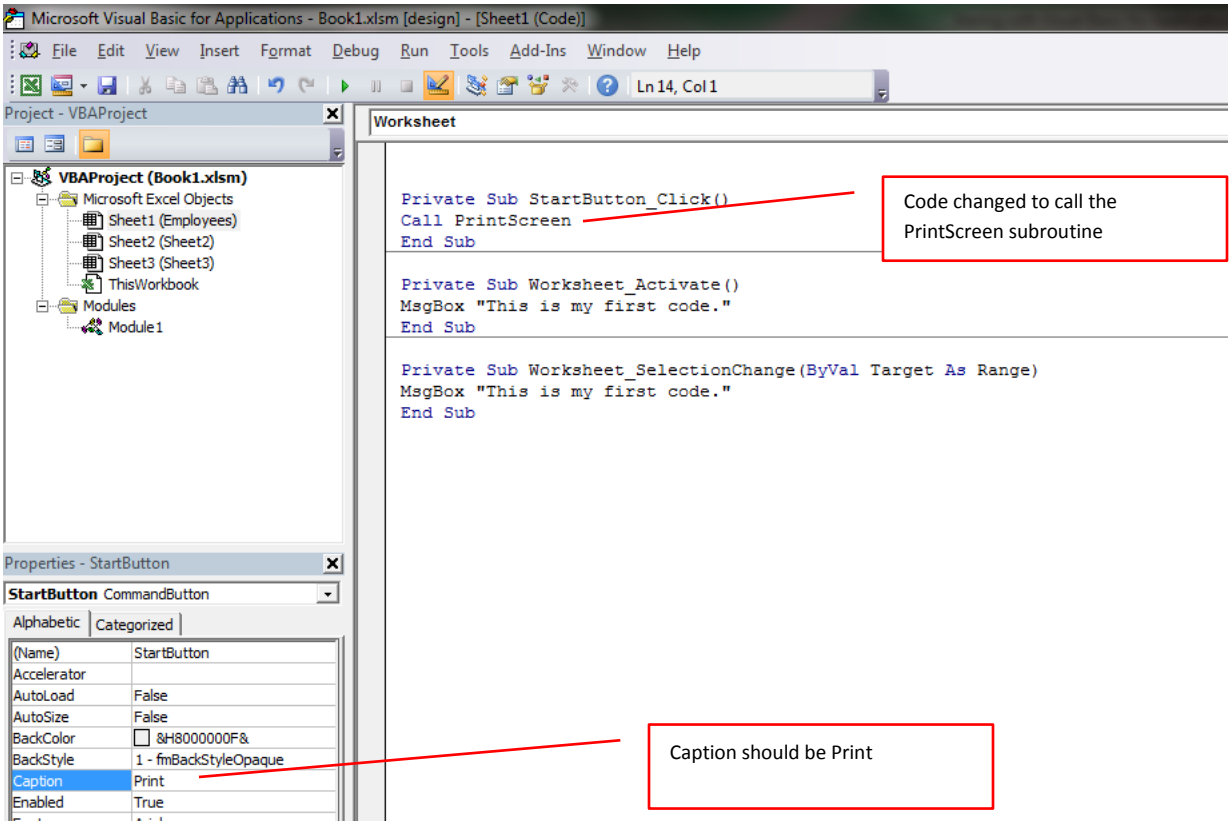
Open the code sheet for Sheet1(Employee) by double clicking it. You will see the code for the StartButton. First, let's change the Caption for the button to Print. We could do this through code as it runs but we want the user to see "Print" before clicking on the button so they know what to expect to happen. We will change the caption through the Properties Window. You should know how to do this by now but just in case, in the properties window below the Project Explorer Window, change the Caption. Remember, you must be in Design Mode to make changes or the StartButton will not be accessible.



Looking at the code currently in the click event for the StartButton is code to display a message box with a message. We want to overwrite that line with code that tells it to run the code we create in the subroutine called PrintSheet. Overwrite it with the following line:

Call PrintSheet

It should look like this:



Following through with the workflow of this code, the user wants to print the worksheet. He/she clicks on the button that says “Print”. The click event for the button is fired. The code in the click event says run the PrintSheet routine in a public module. The PrintSheet subroutine is found and its line of code is executed. The routine sends a print command to print the active worksheet which is the sheet that the user is viewing and the print button located. It is a fairly logical process when you think about it. So far, the codes that we have been writing are simple, one line codes. The procedures can become very complicated with hundreds of lines of code involving decision making based on criteria and with actions performed based on other criteria. As you learn more about VB programming, the above statement will become clear.

Subroutines versus Functions

Up to now, we have looked at samples of subroutines. A subroutine is a set of instructions to do something when we want it done. Previously, we wanted a message box to display or a sheet to print. A function goes one step further and gives us information **back** based on the information we give it. In Excel, you are probably familiar with the Square Root function. In a cell you might enter the formula:

=SQRT(16). The cell will display "4". You gave or passed the function a parameter to work on, in this case 16, the function returned the value of 4. This is an example of an Excel function but in VB but you get to write your own. A function does not necessarily have to deal with numbers, it can deal with text. Any routine that you want a returned value is a function. Let's look at some different examples:

1. Number: Like the square root function, if you provide the length, width, and height, in inches, you are retuned the cubic feet of concrete you will need to pour a slab. The function might look like: x = Concrete(100, 20, 4). 46.3 is returned and is given that value to x.
2. Text: You provide a string of text and you are returned a capitalized string of text that has the first initial of each word. This function may be written something like:
   x = anagram("Central Intelligence Agency") "CIA" is returned and given that value to x. The quotes indicate that it is a string of text and not numbers to be calculated.
3. System Info: Sometimes you do not need to provide a parameter. Let's say we write a function to return the name of the operating system we are using:
   x = OS(). Here we did not provide a parameter but the lines of code that we might have written returned "Windows" and set it to equal x.

The names of the above functions are fictitious. Call them what you like as long as their names do not conflict with built-in functions. Have you noticed that the value returned must be assigned to a variable if you want to run them from code? Simply put, a variable is the name of an area in memory that holds a value. In the above example, x was assigned the string "CIA". If you used the variable x in lines of codes it would be the same as using "CIA". Example:

Print x will produce the same results as Print "CIA".

Functions that we write can be accessed directly from the spreadsheet as well. Select a cell then in the formula bar, type an equal sign followed by the function. You will see a dropdown box with the function listed. Complete the formula as you would any other built-in Excel function. Not to confuse things but subroutine can also have parameters passed to them, however, they do not return a value. Say you wanted to save the file you are working on. You had written a subroutine to accomplish this task. But what do you call the file? You may place a call to the subroutine with a line of code that looks like this:

Call SaveFile ("School Project 4")

The string "School Project 4" is passed to the subroutine that we wrote and the file is saved with the name we passed. Since the subroutine does not return a value, the line of code is not set equal to a variable (like x= in the above examples)

Let's look at some of the above function scenarios and write actual functions:

We are going to create a function to calculate the cubic feet of concrete needed when provided with length, width, and height in inches. First, we should create a module that deals specifically with calculations. Click Insert> Module. Change the name of the Module to Calculations using the Properties Window. Double click the Calculations module to open the code window for it. Next we need to create

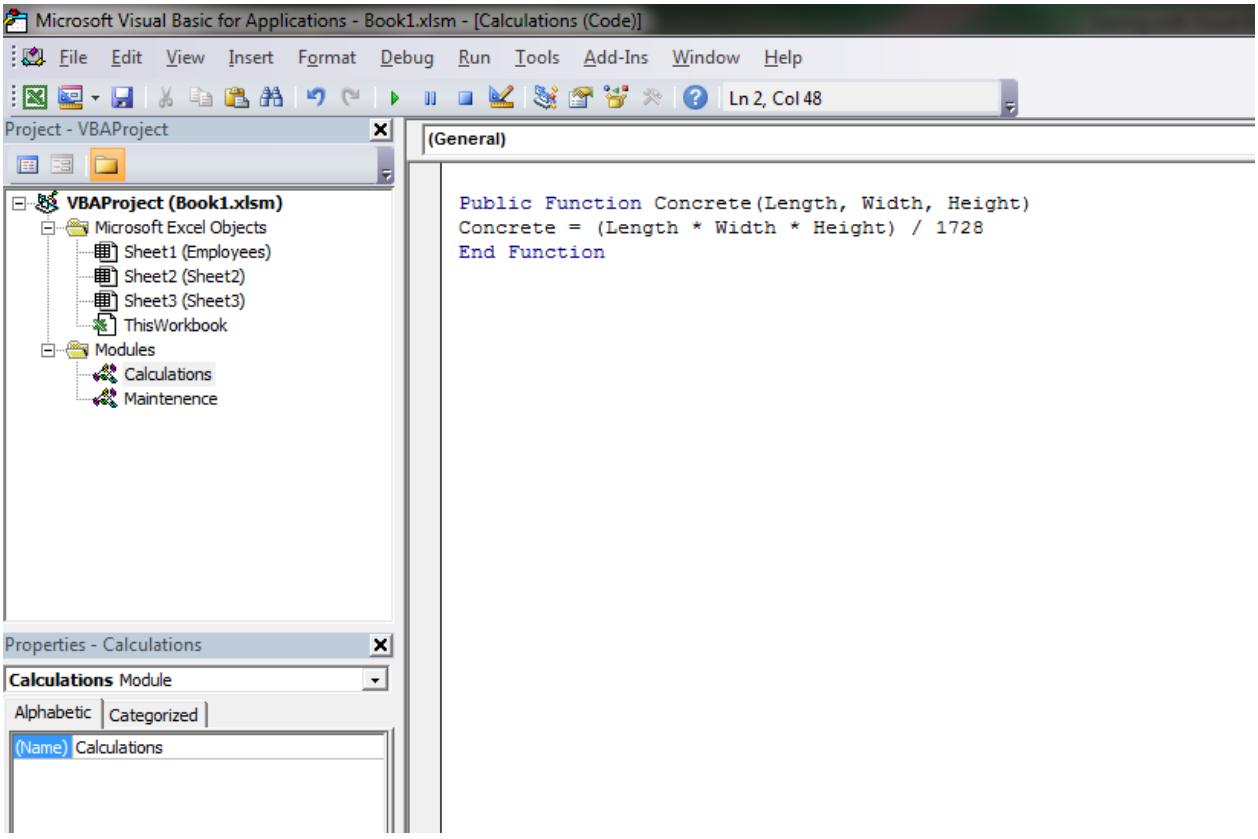
the function. Click Insert> Procedure…> enter the name Concrete> select Function> select Public> Click OK. The code window now shows a blank function with the name that we gave it.

Notice the words Function and End Function? When we created subroutines, their heading was Sub and End Sub.

Enter the following line of code into the function:
Concrete = (Length * Width * Height) / 1728

This is the formula for calculating the cubic feet of concrete. The length, width, and height are variables passed to the function from the call procedure to complete the calculation. All three are needed for the calculation to work. If height is missing or text is sent instead of a number, the routine will fail, an error message produced, and the code will stop. So how does the function get what it needs? We must provide what parameters the function expects. This is done by changing the header to read:
Public Function Concrete(Length, Width, Height)

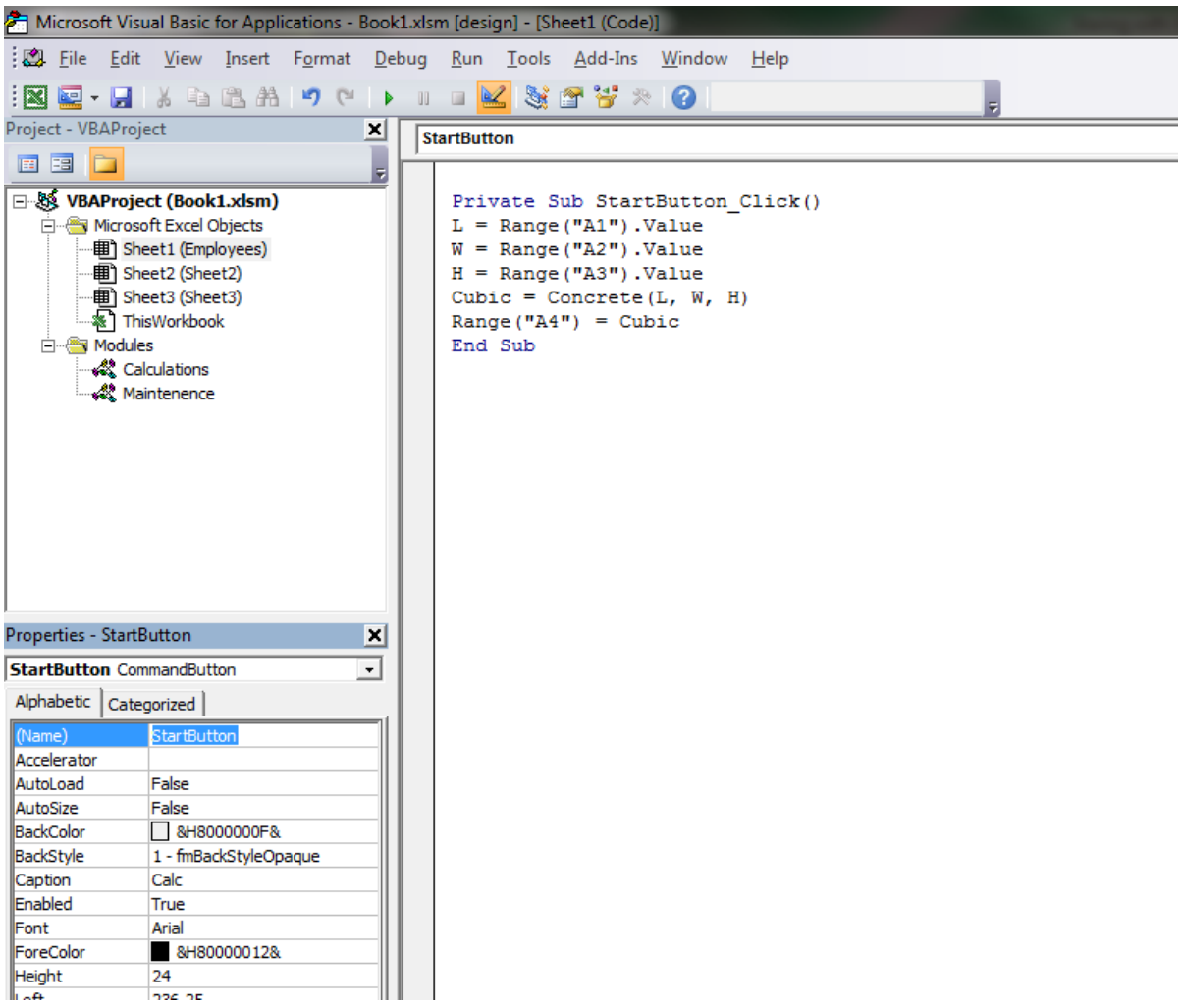
The function when called now expects to see three variables passed to it in the order they are received. The first will be the length; the second is width, and lastly, height. As the numbers are received, they are assigned to the variables (in order) within parenthesis then used in the code. The result of the formula is then set to a variable which is <u>always</u> the same name as the function, in this example, Concrete. Concrete is returned to the calling routine and set to a receiving variable (x = in the above examples). This is very important to remember. Many programmers make this mistake and forget to do this. The receiving variable does not have to be x. It can be what you choose but it cannot be a reserved variable used by the system. The function should look like this:



Now let's take a look at the calling procedure (The procedure that has the line of code that calls as well as passes the parameters if necessary to the function). Earlier in Sheet1, called Employees, we created a commandbutton and called it StartButton. Its caption has changed as we progressed but you should now know how to change the caption to "Calc" (without the quotes). Hint: Use the Properties window and make sure you are in design mode. Open the Sheet1 (Employees) code window to view StartButton's Click event routine. Remove the code between the headers and copy paste the following lines of code:

```
L = Range("A1").Value
W = Range("A2").Value
H = Range("A3").Value
Cubic = Concrete(L, W, H)
Range("A4") = Cubic
```

While you are at it, highlight the other two subroutines and delete them otherwise you will get message boxes every time you change cells. The calling procedure should look like this:



Return to the Excel spreadsheet and deselect design mode. In cell A1 enter 100. In cell A2, enter 200. In cell A3, enter 4. Now click the ”Calc” button. If everything has been entered correctly, you should see the result of 46.2963 in Cell A4 (formatted as general). If formatted as number with no decimal places, then you would see 46

Let’s take a look at the workflow of these events. We have a button on Employees sheet that when clicked, fires its click event and executes its code line by line. In the first line of code it is saying that whatever is in cell A1, assign it to the variable L, and so forth for lines 2 and 3. Line 4 makes the call to Concrete function and passes the values of L, W, and H which is actually sending the values in the order of 100, 200, and 4. Now to the function. The function receives the values 100, 200 and 4 and assigns them to the variables Length, Width, and Height respectively. Using the formula

Concrete = (Length * Width * Height) / 1728

Concrete is calculated with the numbers that the variables represent. Concrete is sent back to the calling routine and execution of the code continues. Concrete assigns 46.2963 to the variable Cubic. And lastly, the value for cell A4 is set to Cubic which is 46.2963

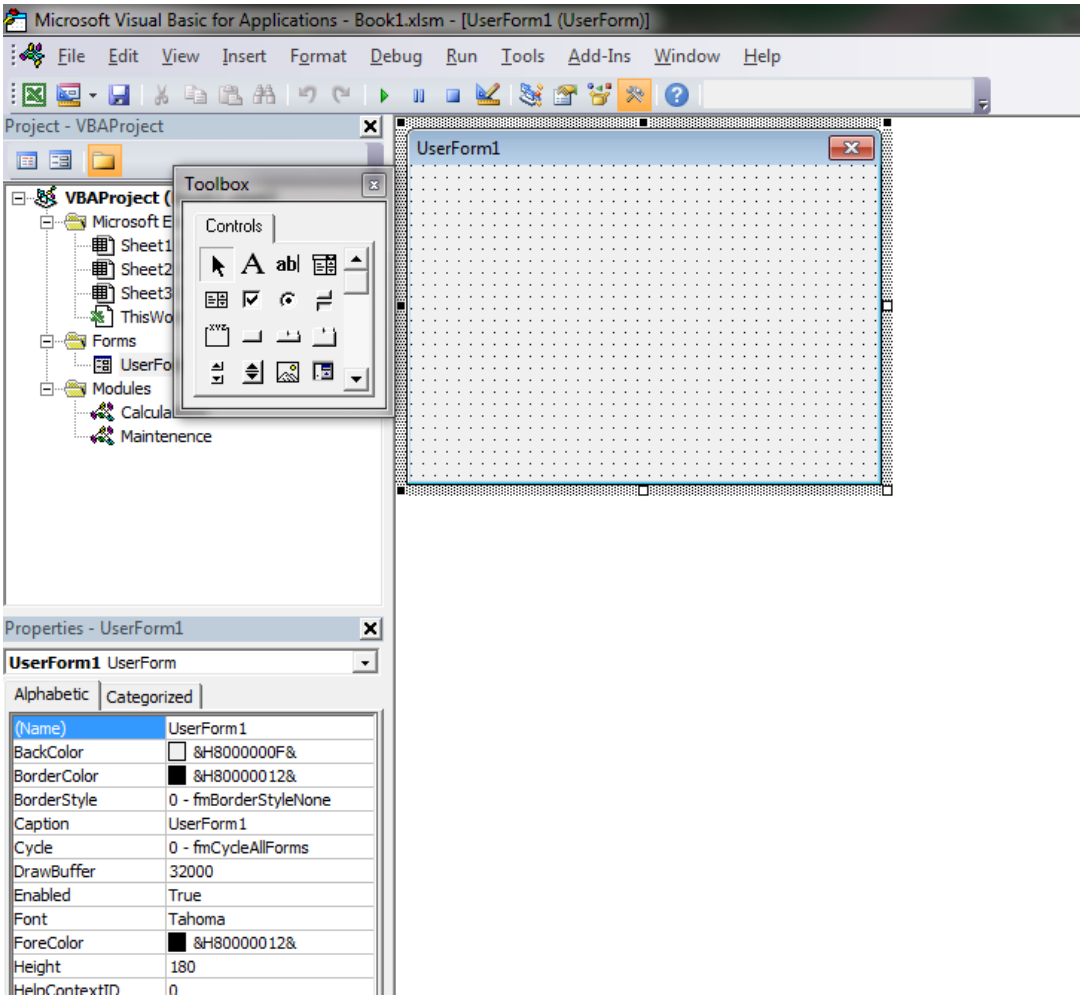
Yep, I know you are asking, “Why bother with VB code? I could have entered the formula or used the Concrete function right on the spreadsheet.” Yes that is correct but what if after the calculation, you wanted an automatic printout and a switch to sheet2 and then, and only then, are those numbers transferred to the new sheet and sorted. You can’t do that with Excel alone. If you Import a calculations module with this function, you will no longer have to place formulas in cells.

Since Concrete is a User Defined Function (UDF), it can also be called for the Excel interface like any other built in Excel function. In a cell type the following: =Concrete(100, 200, 4) then <enter>. You will see the result calculated immediately. The point is that UDFs, once created, can be accessed from code or from the Excel user interface (cells).

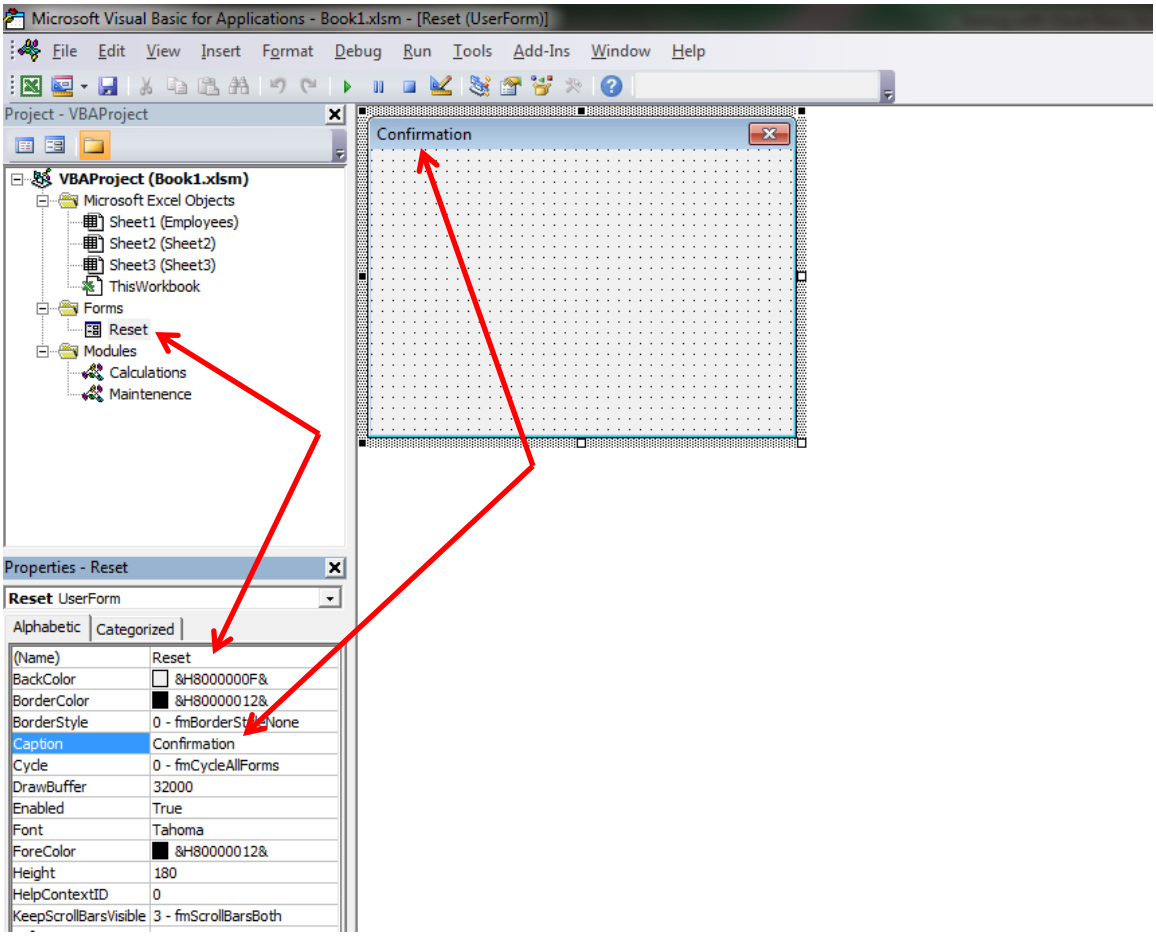
User Forms

User forms are popup boxes that are extremely useful for a number of purposes. As you create these forms, you can place controls on them just like you did with sheets. And just like sheets, as well as modules, they have their own code sheet. Their code sheet contains all the code for the form and all of its elements. So, why would I want to create a form? Example: Employees sheet asks the employee several questions. At the bottom of the list of questions you place a reset button to clear all the entries made. If the user accidently clicked this button, they would lose all of the data and would have to start over. Most programmers create a popup form to confirm the action. If the user has made a mistake, they can easily cancel the action else, go ahead and clear their entries. Let’s create a form

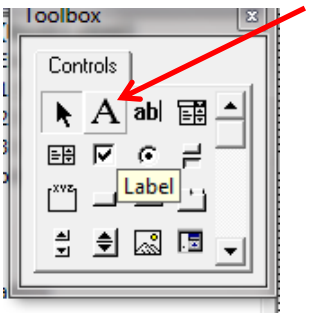
In the VB Editor click Insert> Userform. You will see the following screen appear:



Notice a few things. A folder called Forms appears in the Project Explorer window with a Userform1. The Userform1 opened with a blank form. The Properties shows all the properties of the Userform. In addition, and next to the Userform is the Controls tool box from which you can select controls to place on the form. First we will change the name of the form to a name that is applicable to what its use is. In the Property window, click on name and call it Reset. In the Project Explorer, the name will automatically change to Reset. We also want to change the form's caption to Confirmation. You should be familiar with how to do this by now.



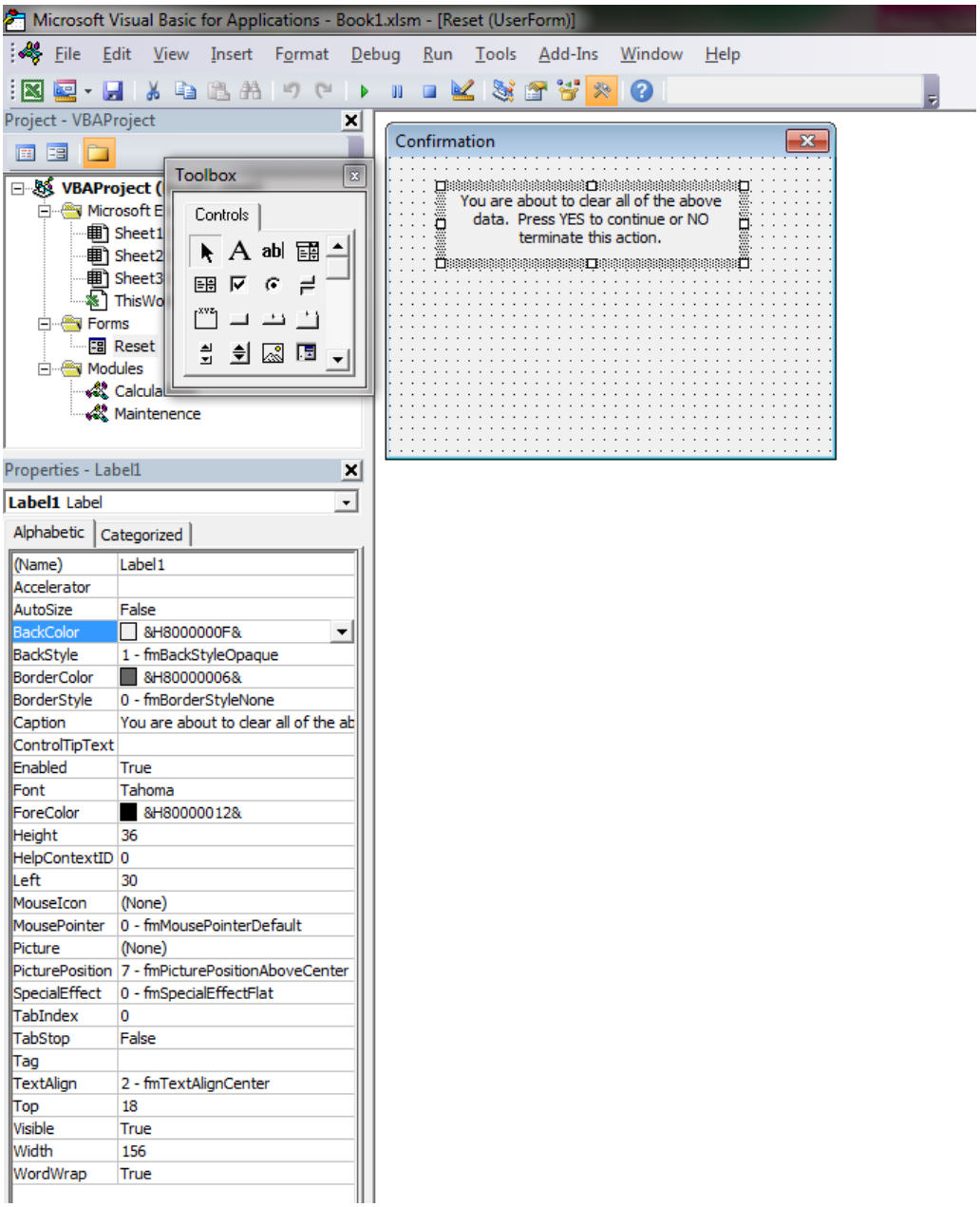
Take a look at some of the other properties. If you need additional information about the property click F1 for help. You will be given a brief explanation of the property and how the property can be changed with code if applicable. Try dragging the right lower corner of the screen to resize the form. Notice how the Height and Width values changed. If you click on the form, the control toolbox will reappear. We are going to create a confirmation message, a YES and a NO button. On the control tool box select Label and drag a label on the Reset userform.



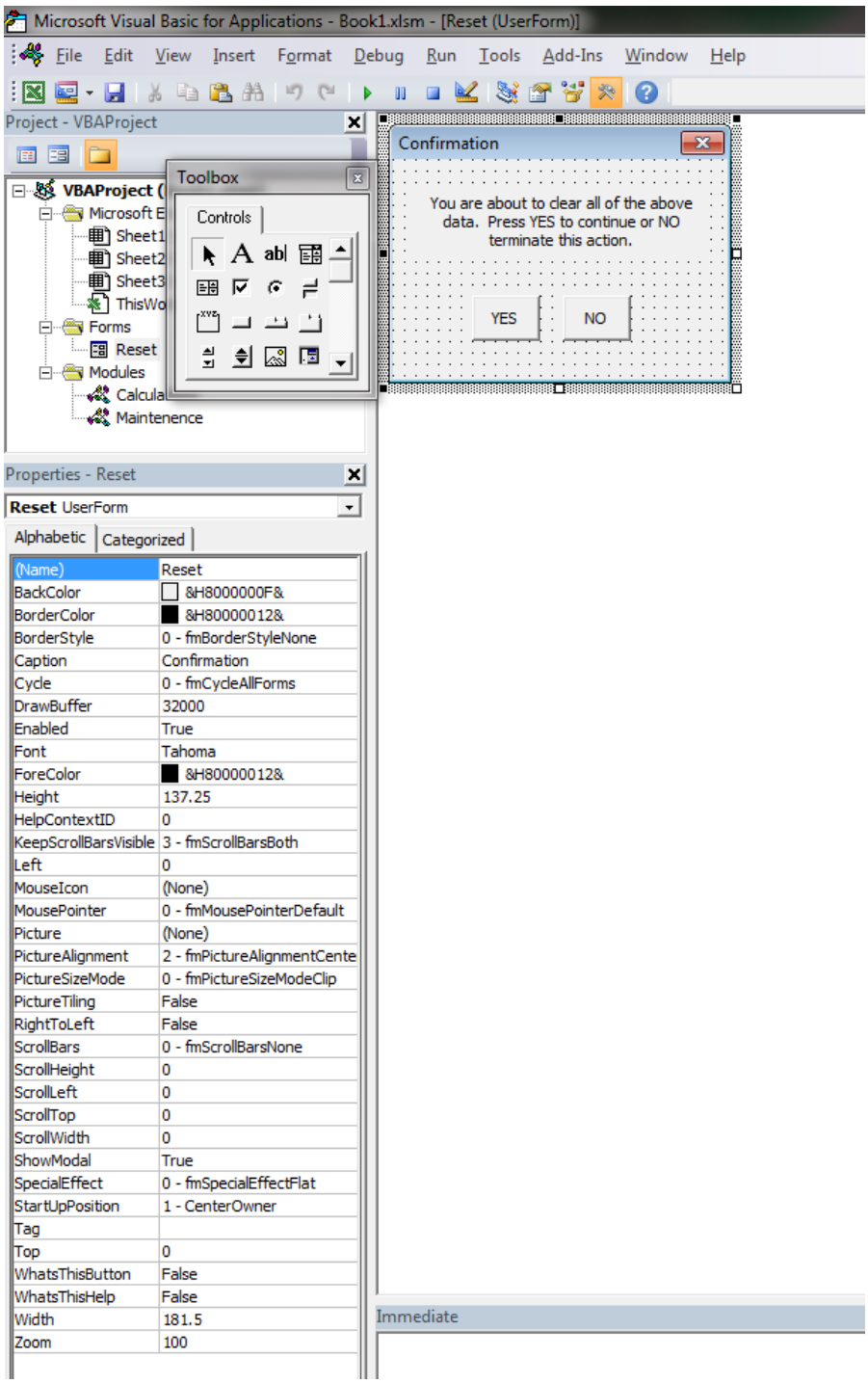
In the Properties window, you will see all the properties for the label. In the Caption property, enter the following string without the quotes:

"You are about to clear all of the above data. Press YES to continue or NO terminate this action."

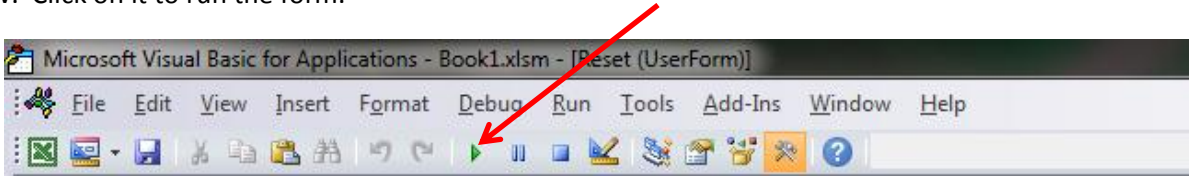
You may have to adjust the size of the control to accommodate the entire text by widening and/or increasing the vertical height. In the Text Align property, change it to center. You can make adjustments to the font's color, type, and size if you like. It should look something similar to this:



We need to add 2 buttons to the form so that the user can make a choice of what to do. If the Control tool Box is not visible, click on the form. Select the Button control and draw a button on the form under the label. In the Caption for the button, change it to YES. Also, change its name to Yes. Create a second button for No and change its name and caption. Adjust the size and positions on the buttons. When all is in place, you can adjust the final size of the form. You do not want a lot of blank space on the form so keep it as small as possible. You form should look similar to mine:



If we were to run the form, we would not see any actions performed because all event subroutines for the buttons, label, and userform are blank. Let's try it any way. On the toolbar, you will see the run arrow. Click on it to run the form.



The form will appear as it runs and waits for a user action. If you click the Yes button, nothing happens because the click event subroutine for Yes button contains no code. The same for clicking the No button, userform, or label. Click on the "X" in the top right corner to close the form and end the run of the Reset userform.

We are now going to write some code for the Yes/No buttons. Let's start with the No button. If the user does not want to continue what should the code do? Simply just close the form and let the user continue to work on the spreadsheet. Open the code window for the form by either double clicking the No button or right clicking the No button and choosing View Code. The code window for the form will open with a blank event click subroutine for the No button. We stated before that we wanted the code to just close the form. Add this line of code to the routine:

```
Reset.Hide
```

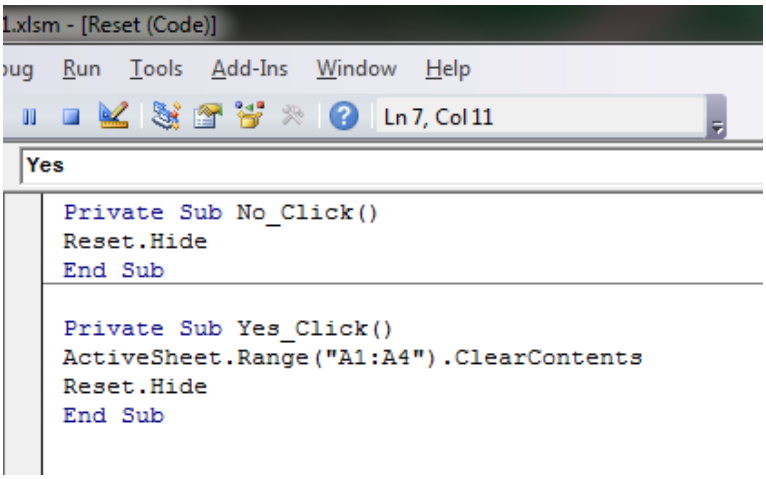
This line of code will close the form. Next, let's work on the Yes button. While you are in the userform's code window, in the dropdown box at the top, select Yes. A blank event click subroutine for the Yes button will appear. What do we want this code to do? Well, the user has indicated that they want to clear the form. In our current Employee worksheet we have Cells A1 through A4 populated with concrete data. To clear the 4 cells, the simplest code to do so would be the following:

```
ActiveSheet.Range("A1:A4").ClearContents
```

Now that we are finished, we need to close the form and let the user continue to work. Enter the code to close the form:

```
Reset.Hide
```

Your code window for the Reset userform should look like this:

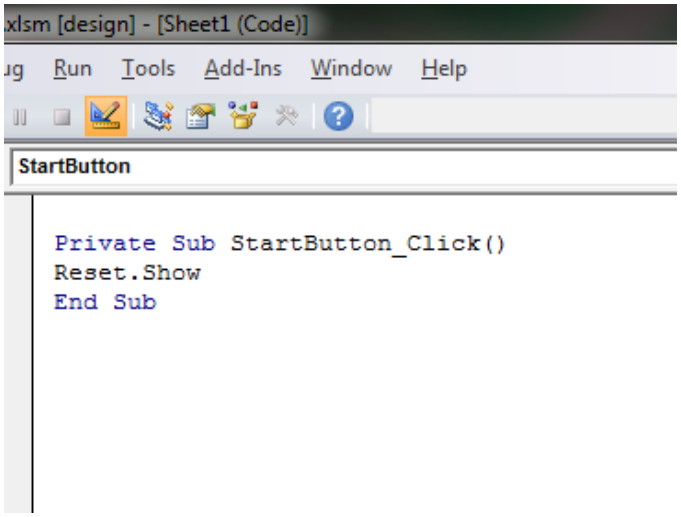


What does this code do when the Yes button is clicked? The first line states, on the active sheet, meaning the sheet that is actively being viewed, with the cells that range from A1 to A4, clear their contents. The second line closes the form. Well what opened the form in the first place? The question

is, what initiates this whole series of events to happen? It is when the user clicks the reset button and we need to write code for that event. We want the Reset userform to appear to ask for confirmation when the user clicks the reset button. Double click sheet1 (Employees) in the Project window. Make sure in the Window header that you are in the Sheet1 Code window. In the dropdown box at the top, select StartButton and you will be directed to the click event subroutine. Since it is the only code in the Sheet's code window it is easy to find. Change the 5 lines of code to the following line:

Reset.show

It should look like this:

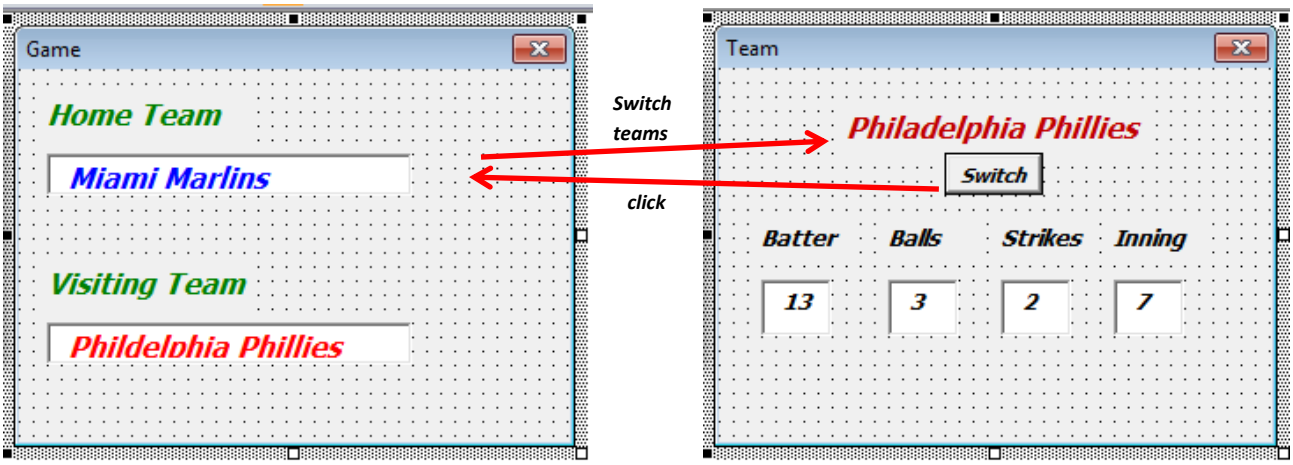


The button has the word Calc displayed on it. We need to change the buttons caption to read Reset. Make sure you are in design mode then click on the drop box in the Properties window and change the caption. We are now ready to run the code. Go back to the worksheet and deselect Design Mode. Click the Reset button. When the Reset form appears, click NO. The form should just close. Repeat the process but now Click YES. The contents in cells A1 through A4 should clear.

Let's take a look at the workflow of these codes. The user has entered data on the worksheet in specified cells. To clear the data, the user clicks the reset button. The click event subroutine for the button fires and executes its code to display a confirmation userform. A label informs the user that they are about to clear the data. Two choices are presented to the user via buttons. Clicking NO fires the click event subroutine for the No button. The code does nothing more than to close the Reset form which aborts any further action. If the user clicks YES, the Yes button click event subroutine is fired which states to clear the contents of Ranges A1 through A4 on the Active sheet which happens to be Employees and close the Reset userform.

New Concept: It is always wise to indicate where the object, whether it be a button, dropdown box, a cell, etc., is located.

If you designed two forms and your code from one form referenced an object (ex: textbox) located on a different form, you must place the reference in you statement so the code knows where to find it. Example: Userform TEAM has a button (Switch) whose action is to assign the value of a textbox located on Userform GAME to a label on TEAM. However in the code, you did not specify that the textbox is located on GAME. The code executes when the Switch button is pressed. But because there is no reference to Game userform, the code assumes that the control, HomeTeamTextbox, is on the same form that the code is running from. If there happens to be a textbox on TEAM with that exact name it will use that caption, but that was not our intent. If not present, an error message will appear because the object cannot be found.



Wrong: Code fails because it cannot find HomeTeamTextbox on the Team userform

```
Private Sub Switch_Click()  'Click event for the Switch Button
    TeamLabel.Value = HomeTeamTextbox.Value  'Assign TeamLabel the value of HomeTeamTextbox
End Sub
```

Correct: Code indicates where to find HomeTeamTextbox

```
Private Sub Switch_Click()  'Click event for the Switch Button
    TeamLabel.Value = Game.HomeTeamTextbox.Value  'Assign TeamLabel the value of
    HomeTeamTextbox located on userform Game
End Sub
```

New Concept: An apostrophe followed by text is a comment which is not executable code. A comment's fore color is green

Recording Macros

Macros, routine, and procedures are words that mean the same thing: A collection of computer commands systematically assembled to achieve a purpose when executed. However, actions can be recorded in the Excel UI and the code is automatically generated for each step. Recording a macro

initiates when you click the Record Macros icon on the developer tap in 2010. In 2003, click on Tools> Macro> Record New Macro. A dialogue box with a default format name of Macro# is suggested but you can change this to anything you like. Click OK. Complete a task, for example, right click a cell and copy it. Right click a different cell and click paste. Now click Record Stop. All the mouse actions were being recorded and are placed in a new or existing module. Macros can only be recorded in the UI. Not in the VB Editor. View the code to learn the code associated with an action. Click F1 on the code components to learn more about them. Try experimenting with the code to see what other properties do. This is the best way to learn VB code

Important Facts to Remember

1. Each worksheet, module, and userform will have its own code window. The workbook itself has a code sheet too.
2. Controls are added to a worksheet in the Excel user interface (UI) whereas, with userforms, controls are added to them in the VB Editor
3. Add a control by selecting the control in the Control Tool box the dragging a rectangle with the crosshair cursor then release.
4. Controls have event subroutines for different actions placed on them. Each event subroutine can have its own code to perform different or similar actions.
5. Make sure you are writing code in the correct Code window by verifying in the window's title bar.
6. Never use spaces or special characters in the name of any object (sheets, userforms, controls, etc.) and give it a useful name
7. Design Mode must be selected to modify a control on a worksheet whether from the Excel UI or the VB Editor.
8. Selecting a property in the Properties Window and pressing F1 will give additional information on the property
9. Most properties changed at design time in the Properties Window can be changed at runtime with VB code.
10. Using IntelliSense with the dot (.) notation will give you clues as to what properties and/or methods are available. Click on the typed wording of the property then press F1 for a detailed description.
11. Dual Monitor setup is very handy to simultaneously display the Excel UI and VB Editor.
12. Place common code that will be accessed from multiple places in a module and the subroutine's header is preceded with the word Public if the **module** was declared Private when it was created.
13. Create a module for each type of action and export them for use in future related projects.
14. Insert message box lines of code to temporarily halt code execution and display the progress of or to verify a variable's value (Msgbox *variablename)*
15. Functions are often passed a parameter and always return a value. Subroutines occasionally passed a parameter but never return a value

16. The function must be set to a variable in the format: variable = Function (parameter) to be run from code. Answer = SQRT(9). It can also be used by entering the formula directly in the cell.
17. The value returned from a function is a variable with the same name as the function.
18. Clicking on a userform will make the Control Tool Box appear.
19. Double clicking on a userform or a control OR right clicking and selecting view code will open the code window for the userform.
20. Get in the habit if including the name of the sheet and/or Userform followed by a dot (.) when referencing controls.
21. Use comments frequently to help explain your code.
22. Experiment recording macros and viewing the code it produces.

Conclusion

Volumes of books have been written with in-depth details on the multitude of features that VBA has to offer. We have only scratched the surface here but the intent is to provide you with the basics so that you can understand the flow of code. More importantly, if you are presented with a sample of code to achieve a certain task, you will have the skills to be able to build the components, add the controls, and insert code in the proper places. To teach how to code is a book by itself. An excellent free place to start would be http://www.garybeene.com/vb/tutor.htm . But in the Important Facts to Remember section, I gave some additional tips to expand your knowledge of the code. Hopefully, you realize that you can take writing a spreadsheet to the next level. And do not think this is limited to spreadsheets. Much of the material we covered here pertains to other Office products such as Access, PowerPoint, Outlook, and Word. Consider expanding further by writing standalone applications with Visual Basic.Net.