Using VBA in Microsoft Excel

Visual Basic for Applications (VBA) is available in Microsoft Excel, Word, PowerPoint, and Access. Using VBA, we can extend the power of Excel by writing custom programming code that make it easier to perform complex calculations.

VBA can also be used to write harmful macros in Office applications. Because of this once you add some VBA to your spreadsheet, when someone opens it Office will post a warning that the spreadsheet contains macros that could be harmful.

At the back of this handout is a section called Visual Basic for Applications that contains a review of Visual Basic syntax and special information pertaining to Excel.

Resources

This document and an accompanying Excel spreadsheet are available online at http://www.bae.umn.edu/teaching/courses/tutorials/

The University has licensed 225 computer books online at this web address
http://sciweb.lib.umn.edu/subject/safari.html

You may also be able to find tutorials and information on the internet by searching for "Visual Basic for Applications."

What's inside

Tips and Hints

Press Alt-F11 to get into the Visual Basic Editor

Put all your code into a module which is added by choosing Insert | Module. You may insert more than one module.

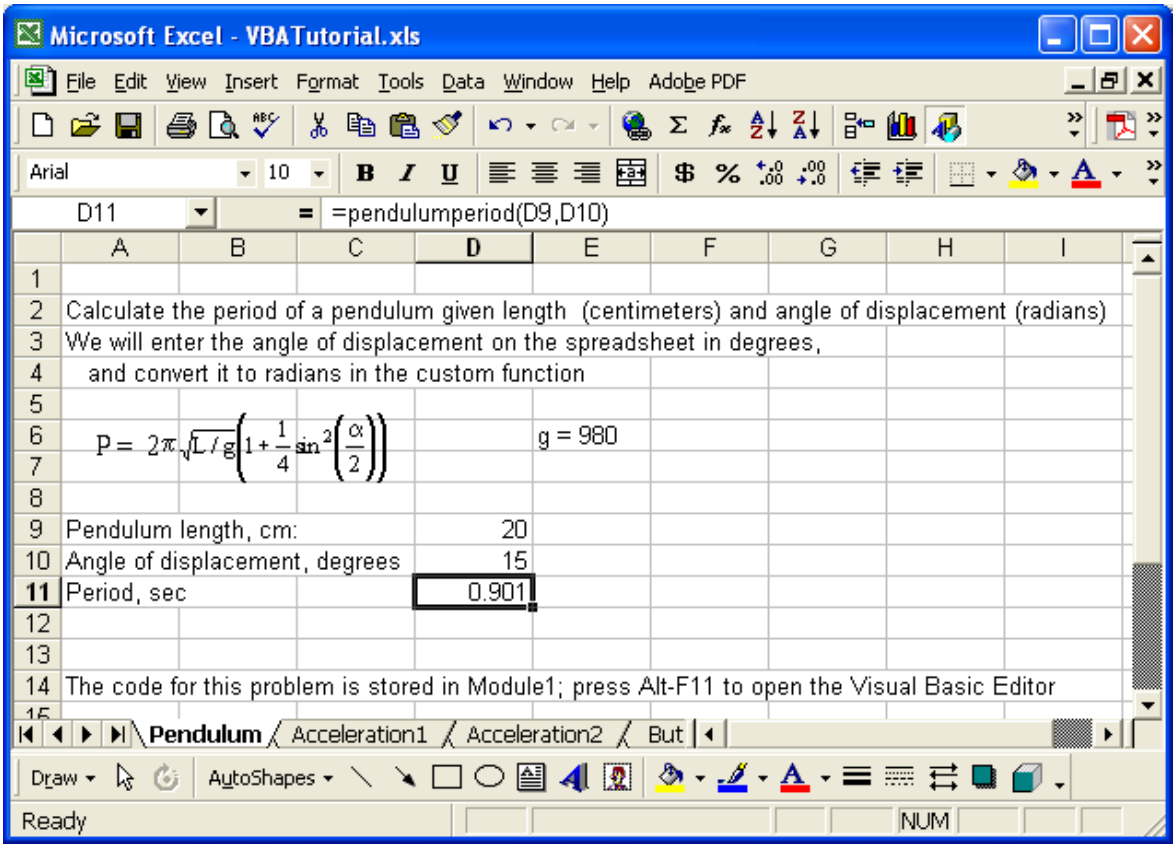
Press F9 to force your spreadsheet to recalculate.

If you get into an infinite loop press Ctrl-Break (the break key is to the far right of the function keys – it may also be labeled Pause).

Entering a Custom Function

Below is an Excel screenshot showing the equation for calculating the period of a pendulum given the pendulum length (centimeters) and angle of displacement (degrees). On the page below the illustration of the function is pasted in as a graphic—not entered into Excel.

This would be a complicated equation to enter in Excel. We can instead write a custom function, named pendulumperiod(length,displacement) in this example, to calculate this. Cell D11 contains a call to that function.



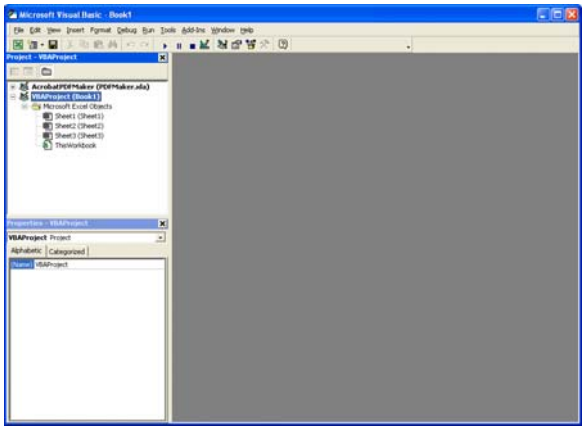
This example is a good illustration of when a custom function comes in handy, but it's a bit complicated for typing in during a class. So let's start with something simple.

Area of a Rectangle

To get started, enter the information below on the spreadsheet.

Length	Width	Calculated Area
10	10	
10	15	

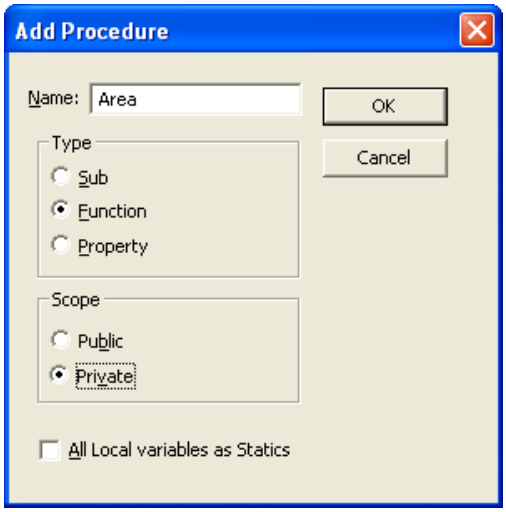
Now we need to write the function to calculate the period. In the **Tools** menu, choose **Macro…** and then **Visual Basic Editor** (note that pressing **Alt-F11** also will get you into the Visual Basic Editor).



This will bring you to a screen like this.
In the **Insert** menu, choose **Module**. This will add a new blank page called *Module1* to your screen, and Module1 should also appear on the left side of the screen listed under Modules.

If your module page doesn't start with the phrase *Option Explicit* you may wish to add it. This forces you to declare all variables (see *Variables* in the appendix to learn about declaring variables) and is a useful tool in preventing mistakes. You can make this the default behavior by opening **Options** in the **Tools** menu and checking "**Require Variable Declaration**."

In the **Insert** menu, choose **Procedure**. This will bring up a new dialog box. Enter a name for your function (`Area`), and select Type *Function* and Scope *Private*. Click OK to close the dialog box.



The lines below will be added to your module.

```
Private Function Area()

End Function
```

This is the skeleton of your Area function, where we will put programming statements that will calculate the area of a rectangle. You could also have typed this in directly if you wished; it is not necessary to use the dialog box once you have memorized the syntax. In fact, if you enter the first line (`Private Function…`) Visual Basic will automatically add the `End Function` line for you when you press Enter.

The function needs some data to work with. Edit the first line so it looks like this.

```
Private Function Area(rlength As Double, rwidth As Double) As Double
```

We're using *arguments* named "`rlength`" and "`rwidth`" to hold the values of the length and width of the rectangle. In VBA all variables are passed by *value*, meaning that if you change the value of `length` inside your function, the change will not be returned to the Excel spreadsheet. (The other option would be "by reference;" variables passed by reference can be permanently changed inside of the procedure.)

A *function* returns only one value, and that value is returned through the name of the function. Therefore we also need to tell Visual Basic what type of value is going to be returned and we do that by adding `As Double` at the end of the statement.

Now we need to add a statement (or statements) to do the calculations. The complete function is shown below. It contains only two lines—a comment, which begins with a single quote and serves to provide documentation, and the statement that codes the equation for the area of a rectangle. On your screen, the comment statement is green. Comments can be placed anywhere, including on the same line as a calculation (at the end of the line only though). They aren't necessary, but are recommended.

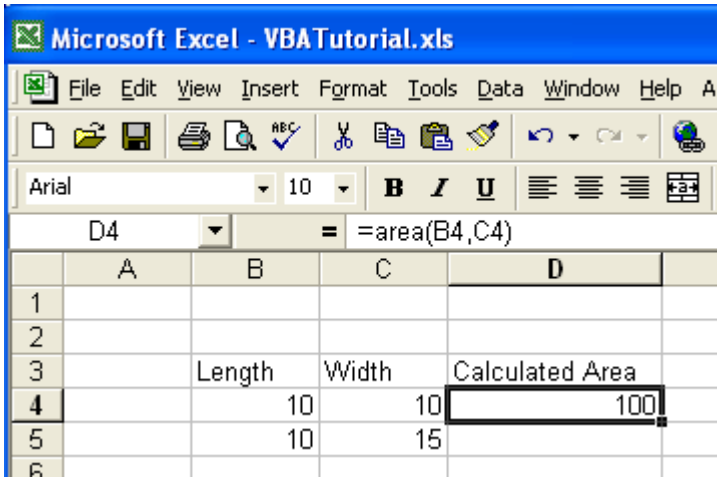
```
Private Function Area(rlength As Double, rwidth As Double) As Double
' Calculate the area of a rectangle
    Area = rlength * rwidth
End Function
```

You can check your function for syntax errors by compiling it: in the **Debug** menu, choose **Compile VBAProject**.

To use the function, return to the spreadsheet and enter it into the cell where you want the calculated value for the area to be stored. Use this syntax

```
=area(B4,C4)
```

where `Area` is the name you gave your function, and B4 and C4 are replaced on your spreadsheet by the cells where you stored the length and angle. The order in which you enter these is important—it must match the order that the variables are listed in your function. This is exactly the same way you use built-in functions such as `Sum` in Excel, except now we have created our own.



If all has gone well, the area will be calculated and placed in cell D4.

Press F9 to recalculate: If there are errors, you may need to review your function. You can press Alt-F11 to return to the screen for Module1. Now if you edit the function you will need to force Excel to update manually either by changing the value in one of the cells that is used in the function, or by pressing F9.

Private vs Public: Visual Basic *procedures* (functions and subs) can be either private or public. Private procedures are available only to other procedures in the same module sheet (i.e.

Module1). Public procedures are available to procedures in other modules that you may add to the spreadsheet.

Period of a Pendulum

Let's now return to the period of the pendulum example. Set up your spreadsheet like this.

Pendulum length, cm:	20
Angle of displacement, degrees	15
Period, sec	

Again, go to the Visual Basic module (press Alt-F11). You can insert a new module to hold this function, or just put it in the same module as the Area of a Rectangle example. In the **Insert** menu, choose **Module** and then create a **private function** named `PendulumPeriod`.

The complete function is shown below.

```
Private Function PendulumPeriod(plength As Double, pangle As Double) As Double

' calculate the period of a pendulum in seconds given the length in centimeters
' and the angle in degrees (angle is converted to radians in this function)

    Const g = 980   ' acceleration due to gravity cm/sec^2
    Dim pradians As Double

    pradians = pangle * 3.14159 / 180
    PendulumPeriod = 2 * 3.14159 * Sqr(plength / g) * (1 + 0.25 * (Sin(pradians / 2)) ^ 2)

End Function
```

Unlike the Area function, we are going to need some extra variables to store data for this one. The first two lines of code are *declarations of variables*, or *data storage locations in memory*, that we will need to use.

```
Const g = 980   ' acceleration due to gravity cm/sec^2
Dim pradians As Double
```

We entered the angle of displacement on the spreadsheet in degrees, but this function requires input in radians. Using a `Dim` statement, this function declares the temporary variable `pradians` which will hold the value of the angle of displacement in radians. The value of g is assigned to a constant (`Const`). A constant is a special data storage location which can't be modified by your program. A variable can be changed by your program. (But remember, changes to variables that are arguments, or parameters passed to the function through the first line, can't be reflected back on the spreadsheet.)

```
pradians = pangle * 3.14159 / 180
```

This statement converts the angle in degrees to radians so that it can be used in the equation.

```
PendulumPeriod = 2 * 3.14159 * Sqr(plength / g) * (1 + 0.25 * (Sin(pradians / 2)) ^ 2)
```

The last statement contains the equation for the period of the pendulum, and stores that result in the variable `PendulumPeriod`, which is also the name of the function. We don't have to declare `PendulumPeriod`; it is already implicitly declared by the name of the function. In fact, it would cause an error to add a `Dim` statement for this variable.

The value for π is expressed as 3.14159. We could have assigned that to a constant

```
Const pi = 3.14159
```

or declared a variable pi using the `Dim` statement and calculated it

```
Dim pi as double
Pi = 4*Arctan(1)
```

Then statements that refer to pi could then be written like this:

```
pradians = pangle * pi / 180
```

To use the function, return to the spreadsheet and enter it into the cell where you want the calculated value for the pendulum period to be stored. Use this syntax

```
=PendulumPeriod(D8,D9)
```

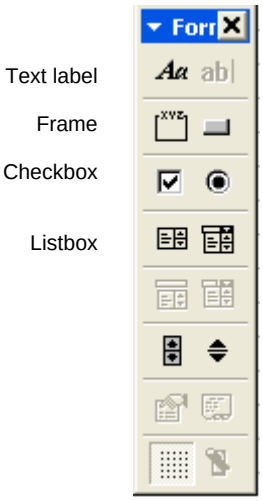
A Function Called by a Function

You may add a number of functions (or subs, another type of procedure) to the same module sheet. The functions do no need to be related in any way, but on the other hand, you may create functions that are not directly used by the spreadsheet, but instead used only by another function. Here is a simple example where we have created a function to calculate the value of π. The pi function has no parameters. In the `PendulumPeriod` function, 3.14159 is now replaced by "pi". Every time Visual Basic encounters pi it steps out to the pi function, calculates the value, and returns and plugs that value into place.

```
Private Function pi() As Double

    pi = 4 * Atn(1)

End Function

Private Function PendulumPeriod(plength As Double, pangle As Double) As Double

' calculate the period of a pendulum in seconds given the length in centimeters
' and the angle in degrees (angle is converted to radians in this function)

    Const g = 980   ' acceleration due to gravity cm/sec^2

    Dim pradians As Double

    pradians = pangle * pi / 180
    PendulumPeriod = 2 * pi * Sqr(plength / g) * (1 + 0.25 * (Sin(pradians / 2)) ^ 2)

End Function
```

Spreadsheet Objects: Menus

When creating a spreadsheet that you want to distribute to users, there are form objects called **controls** that help you to gather input in a user-friendly way.



In the **View** menu, choose **Toolbars…** then select the **Forms** toolbar. This displays a collection of different objects that you can place on spreadsheet. Some of the more common ones are labeled to the right.

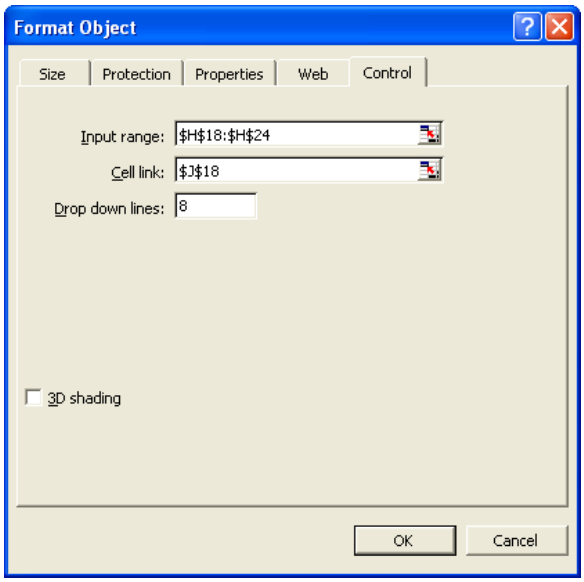
Drop-down menus or list boxes

Corn
Wheat
Barley
Oats
Sorghum
Soybeans

To create a drop-down menu, start by entering values for your drop-down menu or listbox on the spreadsheet. The example to the left shows a list of grains entered on a worksheet.

Next, select the drop-down menu or listbox from the forms menu and draw it on your spreadsheet. (A drop-down menu opens when the user clicks it. A listbox displays a list of items to select from; a scrollbar lets the user see more items if there are too many to fit in the box.)

Right-click on the new control and choose **Format Control**. A dialog box like this will appear.



The **input range** is the range of cells with the data that you want to show up in the menu, i.e. the cells which contain Corn, Wheat, etc. above.

The **Cell link** is the cell where you want to store the number of the user selection. Neither the input range nor the cell link need to be on the same worksheet as the actual menu control.

Drop-down lines lets you control how many menu items show when the user clicks the menu. If you have more items than is indicated, the menu will include a scroll bar. For example, if you have 10 items in your list, you may want to increase the number to 10 so the user doesn't have to scroll to see everything.

Once you've done this, try your menu. For the example here, if the user chooses Corn, the value 1 (i.e. the first selection in the list) will be placed in cell J18. If the user chooses Oats, the value 4 will be placed in cell J18. Your spreadsheet or VBA program can now use this information to take action based on the user's selection.

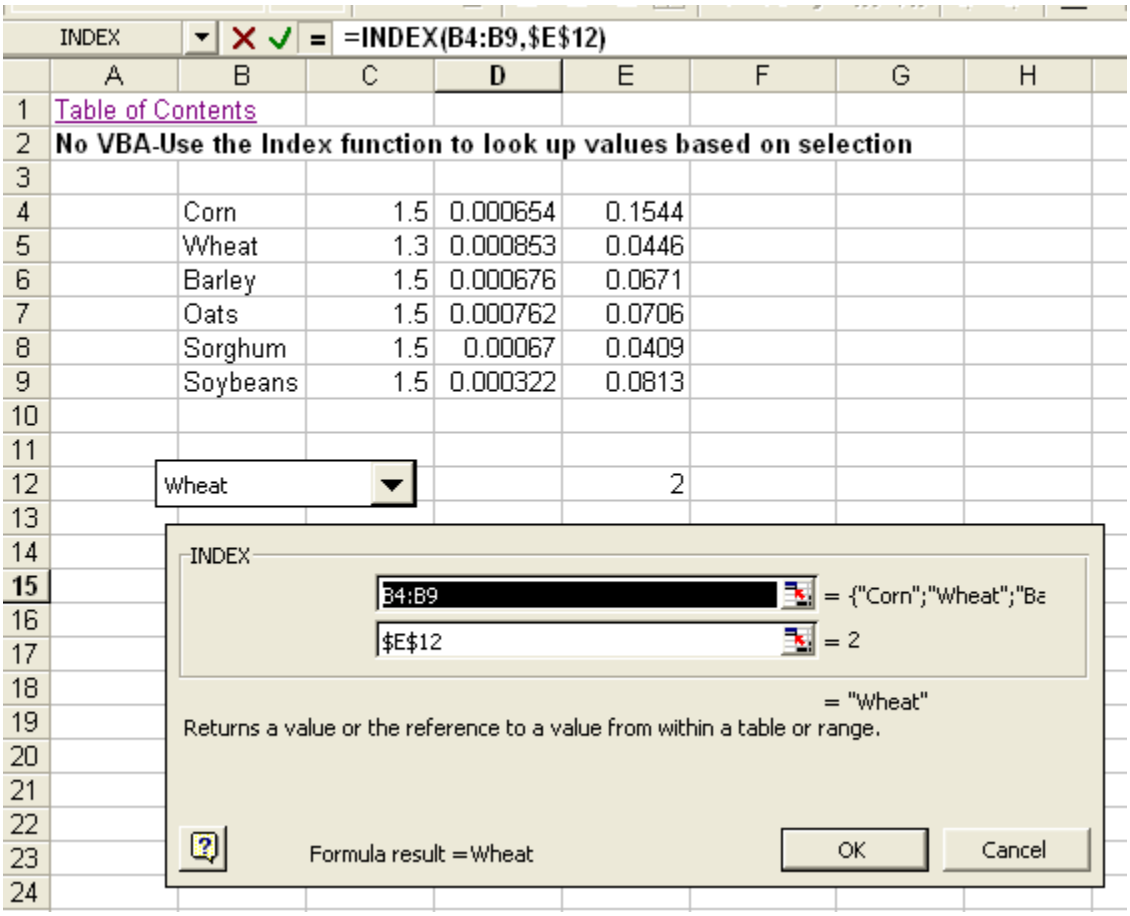
Example: Using a Menu Without VBA

It is not necessary to use Visual Basic code to take advantage of menus. You can use built-in lookup functions to look up information in a table on the spreadsheet based on the user's selection.

Add the following information to your spreadsheet.

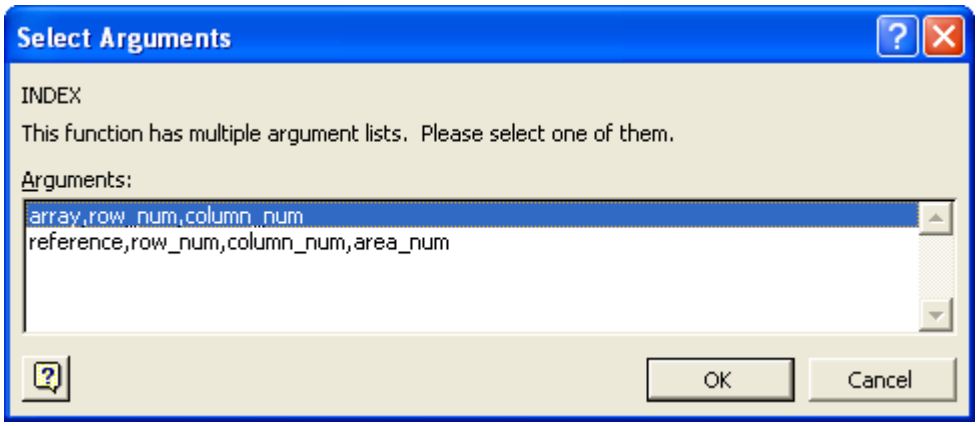
	Corn	1.5	0.000654	0.1544
	Wheat	1.3	0.000853	0.0446
	Barley	1.5	0.000676	0.0671
	Oats	1.5	0.000762	0.0706
	Sorghum	1.5	0.00067	0.0409
	Soybeans	1.5	0.000322	0.0813

The **Index** function uses the Link Cell to find the appropriate row in the selected column.



In the example above, we are using the Lookup function to find the name of the crop selected by the user and store that value in cell D15. Start by placing your cursor in the cell where you want

the crop name to go. In the **Insert** menu, choose **Function**. For this application, I used the first type of index lookup function (below).



Next, you need to indicate

Array—the table of data that we're interested in checking
Row_num—which row in the table we want – this is our Link Cell
Column Num—optional – you could choose from more than 1 column

Once you've set this up, making selections in the menu will cause the data stored in cell D15 to change. For the example shown, that data will be "Corn."

In the same manner, you use lookups to gather data from columns C, D, and E.

This example is from data on resistance to airflow of grains. The accompanying spreadsheet contains a fan requirement table that was built by picking data from a table based on user selection, and then entering equations to compute Bushels, Total Airflow, Static Pressure, and Horsepower.

Example: Using a Menu with VBA

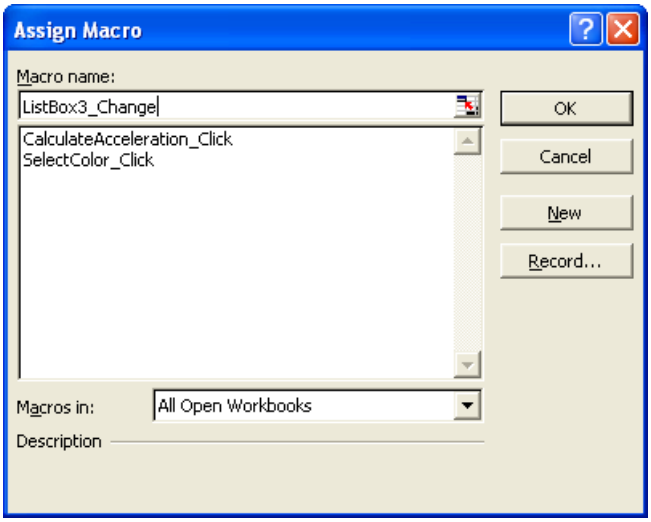
Another way to use the menu is to link it to a Visual Basic procedure that activates as soon as the menu is clicked.

Create a list of new inputs that looks like this:

Red
Green
Blue
Yellow
Transparent

Create a list box on your spreadsheet using these inputs. The list box works in the same way as a drop down menu; it just remains open all the time. If the list box is smaller than the number of items you have, it will include a scrollbar.

Right-click on the listbox and choose **Assign Macro…** This will open a dialog box that lets you assign a Visual Basic Sub (this is a different procedure from a function) to the listbox. It will give you a suggested name for the macro, in this example it is ListBox3_Change. You can use that name or type in a different one, however you should retain the "_Change" part of the name as it provides useful documentation about what kind of action triggers this Sub.



Click **New** when you have finished entering your Macro name.

You will be taken to the Visual Basic editor and a new module sheet will automatically be entered for you. It will contain lines like this:

```
Sub ListBox3_Change()

End Sub
```

A Sub is another example of a procedure. Like a function, it can have arguments in the parentheses and contain statements of Visual Basic code inside. Unlike a function, you can't call it from the spreadsheet. In regular Visual Basic programming you would use the Sub this way:

```
Call ListBox3_Change(parameter1,parameter2…)
```

Parameters are passed to the Sub inside the parentheses and results are returned to the program by changing the parameters.

On the spreadsheet, this is a specialized case of a Sub that is called when something—an "event"—happens to an object on the spreadsheet. Common "events" include Change (when the menu changes), or Click (when a button or radio button is clicked).

So, when we get into Visual Basic, we know that the list box has changed. We're here because we want to do something based on that.

The purpose of this particular list box is to color an area on the spreadsheet. Below is sample Visual Basic code. Note: we have no power over the names of the objects on the spreadsheet. Excel names them for you in order as they are created. Your listbox may have a different name. In the full version of Visual Basic, you have the power to name your objects (such as "ColorListBox") and to get information from them (such as what item is currently selected).

```
Sub ListBox3_Change()

    Dim theRange As Range
    Dim c As Range
    Dim menuitem As Integer
    Dim selectedcolor As Integer

    Set theRange = Range("D10:E13")

    menuitem = Range("D3")

    If menuitem = 1 Then selectedcolor = 3
    If menuitem = 2 Then selectedcolor = 4
    If menuitem = 3 Then selectedcolor = 5
    If menuitem = 4 Then selectedcolor = 6
    If menuitem = 5 Then selectedcolor = 0

For Each c In theRange
        c.Interior.ColorIndex = selectedcolor
    Next c

    Range("D4") = "My menu selection = " & menuitem
End Sub
```

This sub uses a spreadsheet variable type called "range" to exchange data with the spreadsheet.

In the first two lines, we declare two range-type variables: theRange, and c.

Then we declare two integer variables: menuitem, and selectedcolor.

The next statement sets theRange to be equal to a group of cells on the spreadsheet.

The next statement sets menuitem to be equal to whatever is stored in cell D3 (our link cell).

In the next series of statements, we use If-Then statements to pick a color based on the menu selection.

Then we use a special case of a loop to set every cell in that group of cells selected before to the color that the user picked.

Finally, to illustrate writing data on the spreadsheet, we fill cell D4 with information that changes as the menu changes.

Another way to implement this is using the Case structure; this is illustrated on the accompanying spreadsheet.

Interacting With Data on the Spreadsheet

In addition to passing variables as parameters in a called function, it is also possible for your VBA functions and subs to pick data off of a spreadsheet or write it back to the spreadsheet.

Reference to a cell

The code examples below refer to data in cells on a spreadsheet.

```
Range("F8").Value ' (refers to data stored in cell F8)
```

or

```
Cells(1,2).Value ' (refers to cell in row 1, column 2, or cell B2)
```

For example

```
' Get the value of rh from cell F8 on the spreadsheet
Dim rh as double
Rh = Range("F8").Value
```

This line obtains whatever is stored in cell F8 and assigns it to our variable `rh`. Now we can work with it in the program.

You can also take data from the program and put it back on the spreadsheet by assigning in the opposite direction.

```
Range("F8").Value = Rh
```

This statement puts whatever value is stored in the variable `rh` on the spreadsheet, in cell F8.

In Excel, the Range object represents any single cell or adjacent or non-adjacent block of cells. If you select a range of cells, then hold down the Ctrl key and select another range of cells, and then set a variable to this selection, the Range object would contain a combination of cells from both blocks.

The examples shown above are the shorthand way of referring to cells on a worksheet. They take data from the *active* worksheet (the one you're looking at). If you have more than one worksheet in your spreadsheet, you may want to take data from other worksheets that are not active, and that may be hidden from the user (for example, if you stored parameters in worksheets that you kept hidden so that users couldn't change the parameters). In that case, you will need to specify on which worksheet the data is stored. The statement below gets data from the worksheet named Acceleration2.

```
mg = Worksheets("Acceleration2").Cells(6, 2).Value
```

This statement gets the value of Cell (6,2) and assigns it to the variable named `mg`. You could also write this like this:

```
mg = Worksheets("Acceleration2").Range("B6").Value
```

Spreadsheet Objects: Buttons and Macros

Excel macros are created by recording keystrokes and saving them as a unit. We can then assign a macro to a button so that it can be executed quickly and easily. Macros are actually VBA code, so they're useful to know about because you can look at the code and learn about manipulating cells and spreadsheets. Essentially, a macro writes VBA code for you.

Example Macro: Converting Fahrenheit to Celsius

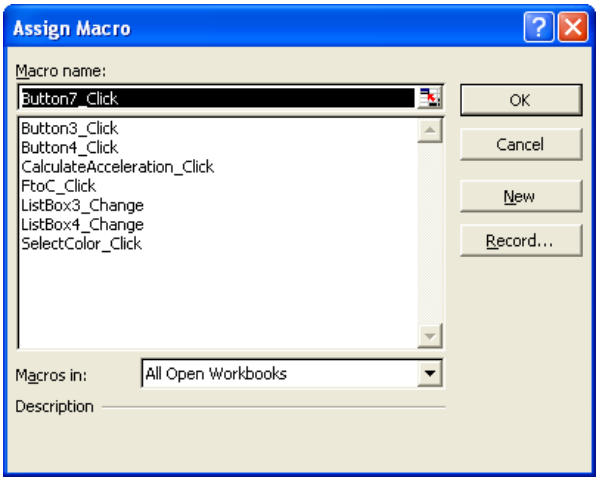
Start by putting this information into a spreadsheet. We're going to set up a macro to convert from degrees F to degrees C. The equation is

	F	C
	212	
	100	
	32	
	16	

```
C = (F-32)*5/9
```

Now draw a button object from your forms toolbar on your spreadsheet.

You will immediately be given the opportunity to assign a macro to the button. The suggested macro name takes the name of the button and adds _Click at the end. Click **Record…** to start recording your macro.



You will be presented with another dialog box that again allows you to rename the macro, assign a keystroke to it, and add some informational text, if desired. You may change the name here if you want, or go on.

Once you've closed the second dialog box, you're ready to start recording. A small box like this

appears on the screen. Note: if you don't see it, in the **View** menu choose **Toolbars** and then turn on the "**Stop Recording**" toolbar. ***All keystrokes from this point on will be captured***. When you're ready to stop capturing keystrokes, click the stop box (on the left). Most likely you will want your macro to use **relative referencing** on the spreadsheet, so ***make sure that the right button is depressed***.

Keep in mind that you don't want any superfluous keystrokes now. Click in the cell where you want the temperature C to be calculated. Enter the equation, clicking the cell where the appropriate F is stored in place of typing F in the equation above. When you are finished, press Enter and then immediately click the stop button to stop recording.

Now put your cursor in the next cell where you want C to be calculated and click the button. The converted value should appear. If it's not right, you might have entered your equation wrong, or not used relative referencing by depressing the reference button.

You can rename your button to something more appropriate by right-clicking on it and choosing Edit Text.

To see what code was generated by the macro, press Alt-F11 to go into the Visual Basic Editor.

```
Sub Button7_Click()
'
' Button7_Click Macro
' Macro recorded 3/13/2004 by Deb Hansen
'

'
    ActiveCell.Select
    ActiveCell.FormulaR1C1 = "=(RC[-1]-32)*5/9"
    ActiveCell.Offset(1, 0).Range("A1").Select
End Sub
```

The first statement after the comments simply selects the active cell (when you clicked the cell in the C column).

The second statement contains on the left side “FormulaR1C1.” This is a “property” of the cell. The cell has a number of properties, including its color, border selection, font, etc. This property is meant to hold a formula in “R1C1” notation, or row and column notation. Look at the formula on the right side of the equal sign. Some of it is recognizable as part of the formula to convert F to C. In place of the actual cell that holds the temperature in F, we have “RC[-1]”. This could also be written as R[0]C[-1] and means, move 0 rows, and -1 column (i.e. backwards) relative to the current cell. That’s the location of the cell where F was stored relative to C.

The last statement was inserted when you pressed the enter key to move to the next cell down.

Macros are a good way to get started with automating spreadsheets. They give you limited additional flexibility and power. To get the most out of spreadsheet programming, you will need to progress to writing VBA code

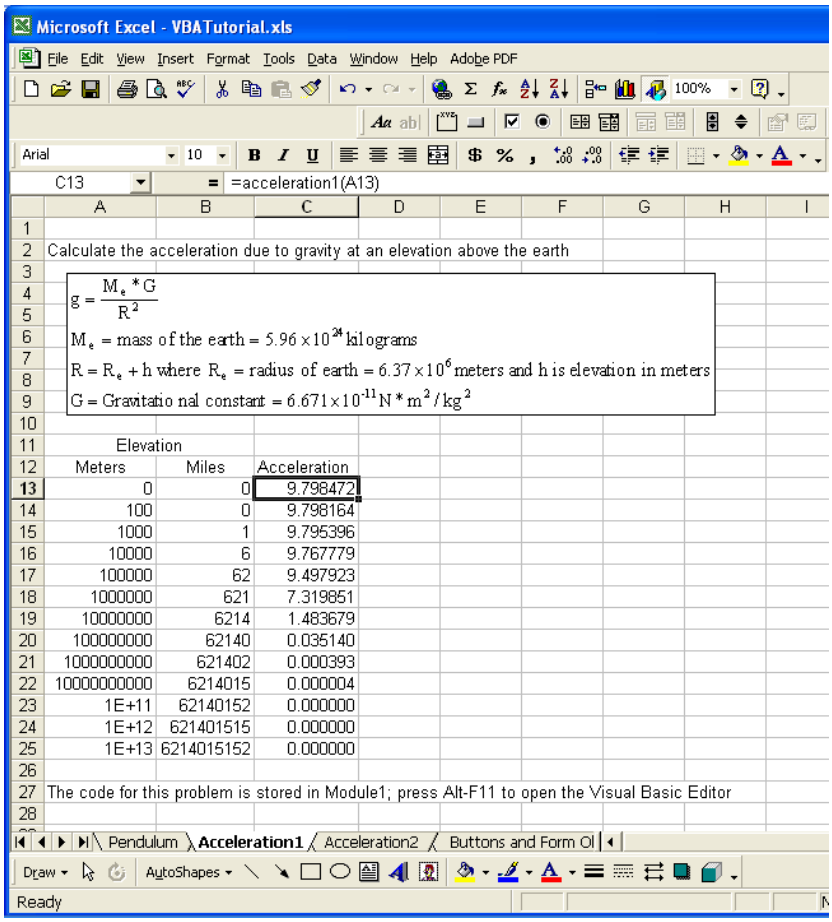
Example: Acceleration Due to Gravity

The spreadsheet to the right shows a series of calculations to calculate the acceleration due to gravity at an elevation above the earth in meters. The elevation in meters is also converted to miles for convenience.

The VBA code for this spreadsheet is shown below. The values in column C, starting with cell C13, are calculated by calling the function Acceleration1 with the elevation in meters as a parameter.

The values in column B, starting with cell B13, are also calculated by calling a function ConvertToMiles. This function is declared as a Public function because we are going to use it again in another example.

The code for this example is stored in the same module as the code for the pendulum example.

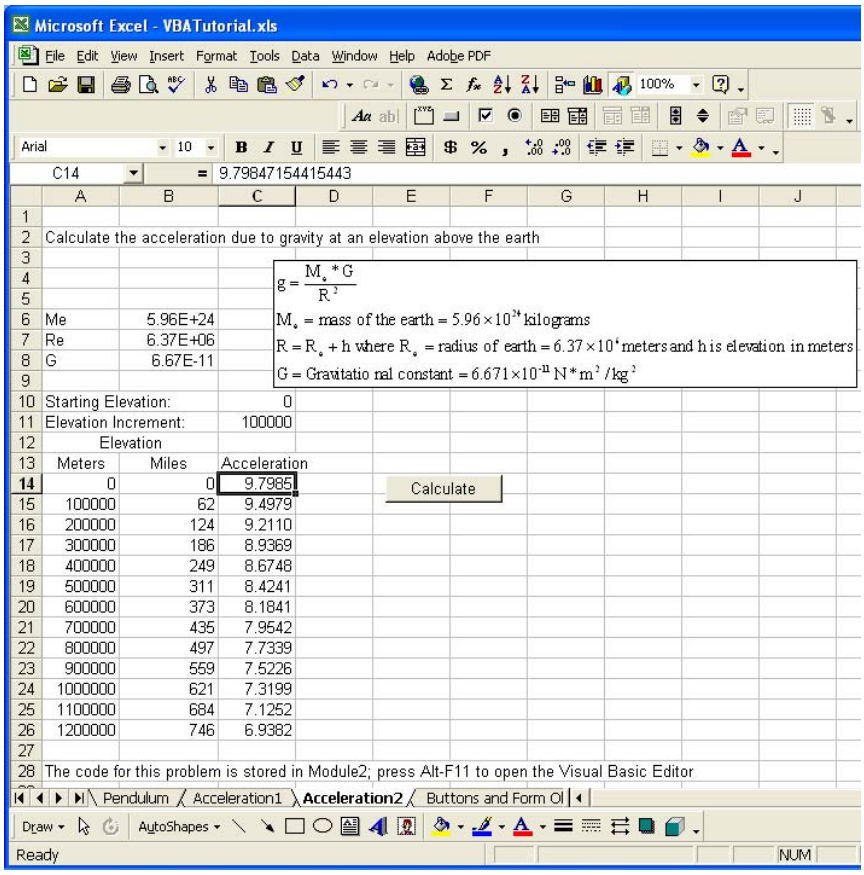



```
Public Function ConvertToMiles(h As Double) As Double

' multiply by 3.281 to convert to feet
' divide by 5280 to convert to miles

    ConvertToMiles = h * 3.281 / 5280

End Function


Private Function Acceleration1(h As Double) As Double

Dim mg As Double, re As Double, g As Double

       mg = 5.96 * 10 ^ 24
       re = 6.37 * 10 ^ 6
       g = 6.671 * 10 ^ -11

       Acceleration1 = (mg * g) / ((re + h) ^ 2)

End Function
```

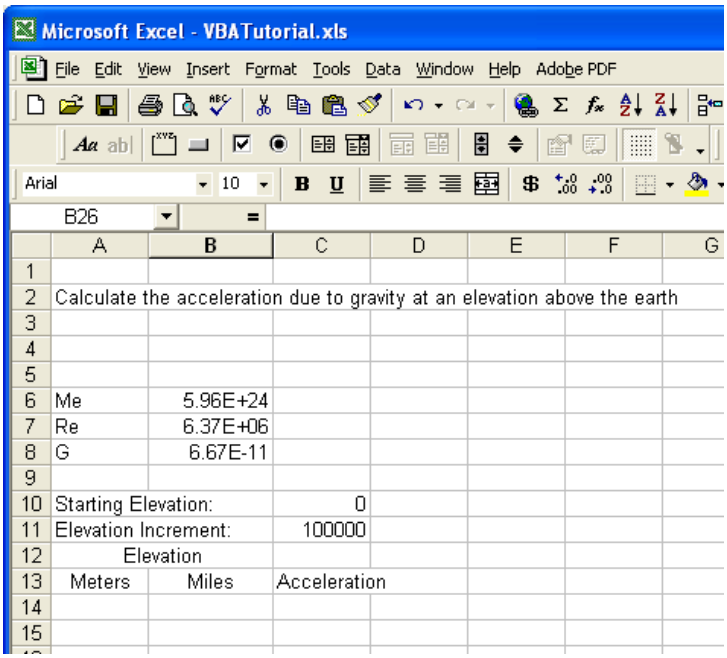
Function `Acceleration1` is a straightforward example similar to the pendulum example. Now we are going to rewrite it to let the VBA code get the values for the parameters from the spreadsheet. The VBA code will also calculate and fill out the entire table.

The figure to the left shows another implementation of the acceleration problem. This time the parameters Me, Re, and G have been stored in cells on the spreadsheet. Note that the data stored in cell C14, at the cursor, is simply a number, not a call to a function.

This spreadsheet also contains a button labeled Calculate. We are going to use this button to update the contents of the spreadsheet.



Start by setting up a spreadsheet with the information shown to the right. Name this worksheet “Acceleration2” as shown on the full example above (right-click on the worksheet tab at the bottom and choose rename). The actual name of the worksheet will be used in our program. It’s easier if you don’t include spaces in the worksheet name.

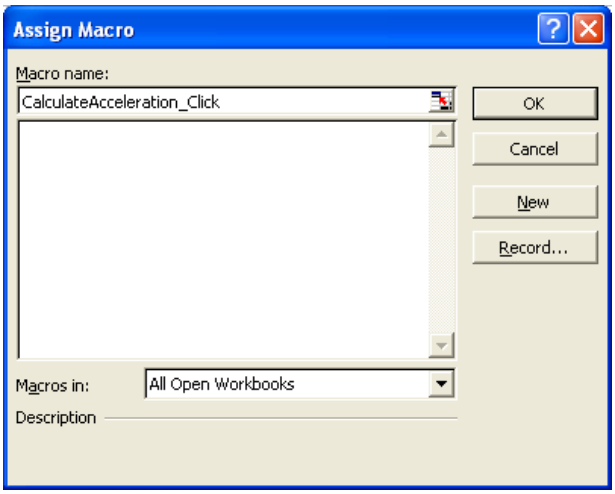


Adding a Button to a Spreadsheet

Open the Forms toolbar if you don't already have it (in the View menu, choose Toolbars… then select the Forms toolbar). To add the Calculate button to the spreadsheet, first click the button icon on the forms toolbar, and then draw the button on your spreadsheet.

Drawing the button will bring up the box below, which lets you assign a macro (a series of tasks) to the button. The default macro name is "Button1.Click". Change the macro name to something that better describes the button, in this case, `CalculateAcceleration_Click`. Then click the **New** button. This will take you into a new programming module in your spreadsheet. Your module will contain these lines:

```
Sub CalculateAcceleration_Click()

End Sub
```



Writing a Sub

Inside the CalculateAcceleration sub, we will put the code that we want to execute when the button is clicked on the spreadsheet. For now, return to the spreadsheet.

If you right-click on the button, you will bring up a menu with some options, including **Edit Text**. Choose this to change the text on your button to `Calculate`.

Go back to the VBA code for the spreadsheet (press Alt-F11). Modify the CalculateAcceleration_Click sub so it looks like this:

```
Sub CalculateAcceleration_Click()

    Call Acceleration2

End Sub
```

Go back to the spreadsheet and click the Calculate button. You will receive an error message "Sub or Function not defined." VBA can't find the sub Acceleration2 that will do the work when the button is clicked, because we haven't written it yet.

Now let's enter that sub.

```
Private Sub Acceleration2()

Dim mg As Double, re As Double, g As Double, h As Double
Dim StartElevation As Double, IncElevation As Double
Dim numrows As Integer, i As Integer

Dim GTable As Range, c As Range

' Define a range of cells on the spreadsheet to be used to hold calculations
' This is similar to a table

Set GTable = Worksheets("Acceleration2").Range("A14", "C26")

' Get the values of the parameters from the spreadsheet
' Include specifics about which worksheet the values are located on

mg = Worksheets("Acceleration2").Cells(6, 2).Value
' Another way to specify the cell on the worksheet
re = Worksheets("Acceleration2").Range("B7").Value
g = Worksheets("Acceleration2").Cells(8, 2).Value

StartElevation = Worksheets("Acceleration2").Range("C10").Value
IncElevation = Worksheets("Acceleration2").Cells(11, 3)

' Clear table by setting the values in the range that defines the table to blank

For Each c In GTable
    GTable.Value = ""
Next c

h = StartElevation

numrows = GTable.Rows.Count

' Use a loop to calculate the values in each row in the table

For i = 1 To numrows
GTable(i, 1) = h
GTable(i, 2) = ConvertToMiles(h)
GTable(i, 3) = (mg * g) / ((re + h) ^ 2)
h = h + IncElevation
Next i

End Sub
```

The first three lines in the sub declare some variables that we are going to need, including variables for the parameters (mg, re, h, startElevation, incElevation) and some variables used by the sub for the loop that fills the table.

The fourth dim statement declares some special variables as ranges (of cells). Gtable will be used to reference the cells that form the table of values on the spreadsheet, and c will be used when we clear the table prior to refilling it.

```
Set GTable = Worksheets("Acceleration2").Range("A14", "C26")
```

This statement assigns to GTable the range of cells that we want to use in our table. The set command allows us to declare the object Gtable and use it to refer to another object abstractly. Gtable now refers to cell A14 through cell C26. This covers an area of three columns and

thirteen rows. Once we have defined Gtable, we can use it as we use a two-dimensional subscripted array in programming. In this case, Gtable(2,3) refers to the value in row 2 (of the object, not of the spreadsheet) and column 3 (on our spreadsheet, column 3 holds the calculated acceleration). If we want to move the table to a different area of the worksheet, all we have to do is change the values on the right side of the assignment statement above.

Next we assign values to our parameters by getting them directly from the spreadsheet. In the code listed above, I've written these in two different ways to demonstrate the syntax. In order to ensure that we are getting the values from the appropriate worksheet, I've included that in the statement.

```
mg = Worksheets("Acceleration2").Cells(6, 2).Value
```

This could also be written as

```
mg = Worksheets("Acceleration2").Range("B6").Value
```

The assignment statement for the variable `re` uses this syntax.

Next, we use a special type of loop to go through each item in the "collection" defined by Gtable and set the values to blank This clears that area on the spreadsheet. A collection is a group of objects of the same type. In this case, the collection is a group of spreadsheet cells.

```
For Each c In GTable
       GTable.Value = ""
Next c
```

We can take advantage of the fact that Gtable is a collection and access each value without having to know how many there actually are, using this special case of the for loop. Visual Basic knows how many objects are in the collection.

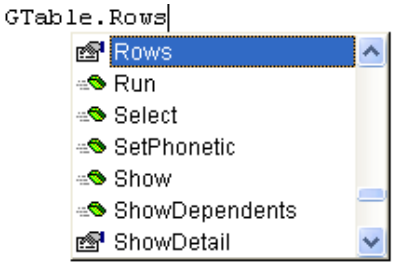
Finally, we get to the meat of the program—the place where we actually calculate and fill the table.

```
h = StartElevation                                                    '1

numrows = GTable.Rows.Count                                           '2

' Use a loop to calculate the values in each row in the table

For i = 1 To numrows                                                  '3
       GTable(i, 1) = h                                               '4
       GTable(i, 2) = ConvertToMiles(h)                               '5
       GTable(i, 3) = (mg * g) / ((re + h) ^ 2)                       '6
       h = h + IncElevation                                           '7
Next i                                                                '8
```

First we assign the starting elevation value to the variable h. (line 1)

Then we determine how many rows are in our table by using the built-in property `Gtable.Rows.Count` to access the row count property for this collection (line 2). A similar

method, `Gtable.Columns.Count`, that lets you determine how many columns are in your table. As you type, Visual Basic will pop up a menu that contains a list of properties and methods available to this control. `Rows` is a property (see the icon of the hand holding the property sheet). The other icons shown in the figure to the right denote "methods," or things you can *do* with this object. Using the method "show," for example, you might be able to "show" or "hide" this object. This doesn't make sense for this particular table, but the method is there.



In lines 3 through 8, we have the loop that creates the table. The loop iterates `numrows` times. Each time through the loop, we assign the current value of h to `Gtable(i,1)` which is the first column of data, the value of h converted to miles to `Gtable(i,2)`, which is the second column of data, and the acceleration due to gravity to `Gtable(i,3)`, which is the third column of data. By storing the calculated values in `Gtable`, the data is automatically transferred back to the spreadsheet, because we mapped `Gtable` to a chunk of cells on the spreadsheet.

When the loop finishes, we exit this sub and return to the `CalculateAcceleration_Click` sub. Once we're back in this sub, we could add more statements to do something else, but since we haven't that sub is also finished and when we exit it, we are returned back to normal operation on the spreadsheet.

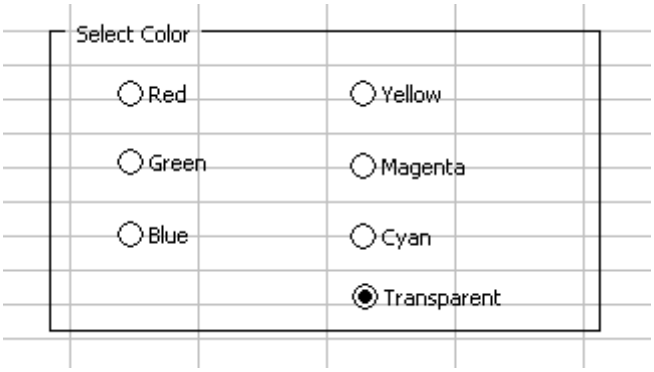
Spreadsheet Objects: Option (Radio) Buttons

Option, or radio, buttons are generally used when only one choice is available to a user. In order to make option buttons work interactively (where the previous selection turns off if the user makes a new selection) they have to be in a frame together.

First draw the frame object on the spreadsheet. Then draw radio buttons inside of it. The radio buttons have a cell link, just like the menu list. The first radio button you add to the frame will generate a value of 1. The second will generate a value of 2, and so on.
You can now write a VBA procedure that uses the value stored in the link cell. You could activate this procedure with another control, such as a regular button, or you could assign that procedure directly to the radio buttons.

To assign the procedure directly to the radio buttons, right click each one. They will present a default procedure based on the name assigned to the radio button by Excel (OptionButton1_Click, OptionButton2_Click, etc.). If you want the radio buttons to work interactively, you need to assign the same procedure to all of them. You may want to enter a new name that is more descriptive of what you want to do (for example, SelectColor). Then click New to add this sub skeleton to a Visual Basic module. Now assign the same sub to the rest of the option buttons.

There is no way to rename controls on a Microsoft Excel spreadsheet. (In a real Visual Basic program, you are able to name your controls in a way that makes sense.) Therefore it is very difficult to check the value of a VBA control from the program, because it is hard to know what the name of the control is. The best way to get the value of a control is to store it in a cell using the cell link when you are formatting the control, and have your program get that value using the methods described earlier.



The figure shows a frame entitled Select Color that contains seven radio buttons. When the user clicks one of the radio buttons, he is selecting a color.

When putting radio buttons on a spreadsheet, you have some tools for making them look nice. In the Draw menu (usuallyfound at the lower left corner of the spreadsheet), there are some selections such as align or distribute, nudge, etc., to help you arrange the buttons.

Press Ctrl and Shift together, then select radio buttons you want to line up. In the Align or Distribute menu, choose Align Left to make their left sides align together. Choose Distribute Vertically to arrange them so they have an equal amount of space between them.

Below is some program code that sets the color of some cells on the spreadsheet when radio buttons are clicked. It uses the Select-Case statement to choose the color for the cells. See the next page for a reference that shows how colors are assigned to numeric values.

```
Sub SelectColor_Click()

    Dim theRange As Range
    Dim c As Range
    Dim SelectedColor As Integer

    Set theRange = Range("G13:H16")

    Select Case Range("G11")
        Case 1
           SelectedColor = 3
        Case 2
            SelectedColor = 4
        Case 3
            SelectedColor = 5
        Case 4
            SelectedColor = 6
        Case 5
            SelectedColor = 7
        Case 6
            SelectedColor = 8
        Case Else
            SelectedColor = 0
    End Select
    For Each c In theRange
        c.Interior.ColorIndex = SelectedColor
    Next c

End Sub
```

Here is another way to use color to enhance your spreadsheet. Let's say that you had created a table of values and you wanted to color cells that were "out of range" based on some definition. Assign your table to a range object, and then iterate through it, assigning colors to the cells based on whether the value of the cell was in range or out of range.

```
Sub EvaluateRange()

Dim theRange As Range
Dim c As Range

Set theRange = Range("B13:K22")
For Each c In theRange
    If c.Value > Range("K6").Value Then
        c.Interior.ColorIndex = 6
    Else
        c.Interior.ColorIndex = 0
    End If
Next c
```

This code assigns the cells from B13 through K22 to the object "theRange." Then it iterates through theRange (For Each c in the Range) and compares the value stored in the cell to some value on the spreadsheet. If the value in the cell is larger than that in cell K6, the colorindex value of the cell is set to 6 (which is yellow), otherwise it is set to 0 (which is transparent).

The MsgBox Method

You can use a MsgBox to get interactive input back to you while your VBA code is running. A line like this

```
MsgBox ("Selected Color = " & Range("G11"))
```

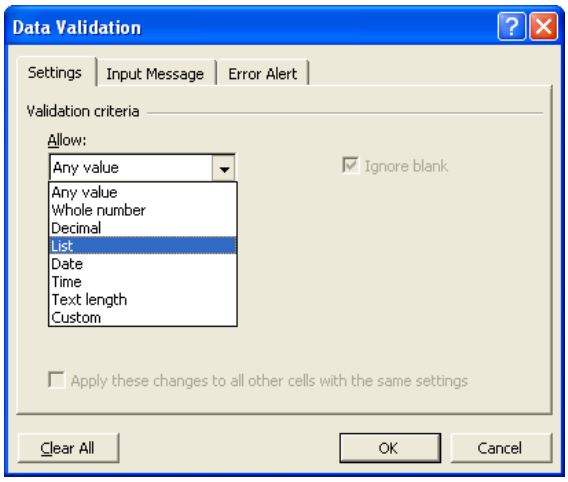
will pop a message box up on the screen. You can give yourself intermediate data if you're having problems debugging your program, or you can use it to give information to the user. The MsgBox can have a lot of parameters. In the line above, we are just concatenating text and a numeric value to put text in the box (the & sign concatenates the things on either side of it; Visual Basic automatically converts the number stored in Range("G11") to text.

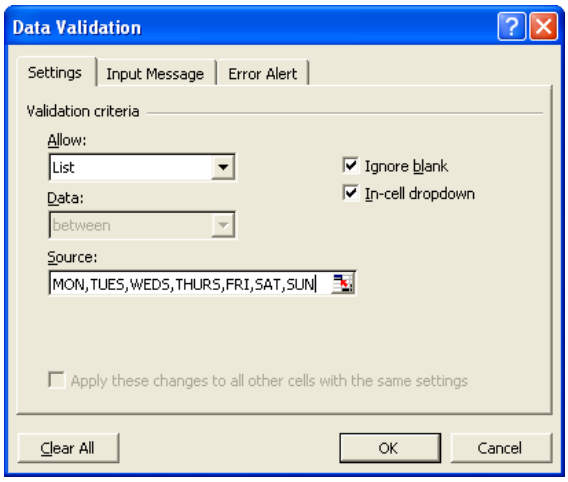
Adding a Drop-Down Menu to a Cell

An easy to restrict the contents of a cell is to add a simple drop down menu to it. Select the cell that you want to validate, then choose Validation in the Data menu. A box like the one to the left will pop up. Note that you have a number of selections. Choose List to make a drop down menu.

Next you will be asked to provide a data source. You can either type the data in directly, or link to a list on the spreadsheet.

The dialog box also gives you some options for messages to the user, etc.





Visual Basic for Applications

Visual Basic is a sequential programming language that is very similar to the BASIC that was the first programming language for many people. Below is a quick review of some Visual Basic syntax.

Variables
Variable names must begin with a letter, can't contain embedded special characters such as periods and can't be longer than 255 characters.

Variables are declared using the Dim statement. Each variable has to be declared separately, i.e.

```
Dim x as double, y as double
Dim StudentID(1 to 10) as long
```

Not

```
Dim x, y as double
```

This statement will declare x as type "variant," i.e, we aren't assigning a type to it. Only y is declared as a double in this statement. You can get away with not assigning types to variables, however it does make programs run more slowly and could cause errors because VBA has to guess what type the variable should be.

Datatypes	***Description***	***How to declare***
String	Characters of any kind; can be of almost any length	String
Integer	Integers in the range –32,768-32767	Integer
Long Integer	Integers in the range –2,147,483,648-2,147,483,648	Long
Single Precision	Decimal numbers, accurate to 7 places*	Single
Double Precision	Decimal numbers, accurate to 16 places*	Double
Boolean	Can be assigned only the values *True* or *False*	Boolean

Assignment statements
The equal sign denotes assignment

```
Dim x as double, y as double
Dim StudentID(1 to 10) as Long

Y = 10
x = y + 2
StudentID(1) = 12345678
```

Program Control

If-Then-Else

```
If condition Then
       Statements
End If
```

```
If condition Then
       Statements
Else
       Other Statements
End If
```

Select Case

```
Select Case selector
       Case valuelist1
              Action1
       Case valuelist2
              Action2
       :
       Case lastValue
              lastaction
End Select
```

```
Select Case selector
       Case valuelist1
              Action1
       Case valuelist2
              Action2
       :
       Case Else
              otheraction
End Select
```

Loops

```
For i = m to n step s
       Statements
Next i
```

```
(step is optional; default value is 1)
(step can be positive or negative)
```

Condition must be true to go through at least one iteration of the loop

```
Do While condition
       Statements
Loop
```

Condition isn't evaluated until after one iteration; you will always iterate at least once

```
Do
       Statements
Loop Until condition
```

Arithmetic Operators (in order of precedence)

Operator	***Definition***
^	Exponentiation
*	Multiplication
/	Division
\	Integer Division
Mod	Modulus
+	Addition
-	Subtraction

Logical Operators

Operator	***Definition***
Not	Flips its operand's logical value
And	Performs **And** operations (both operands must be true to give a true results)
Or	Performs **Or** operations (returns true if either of its operands is true)
Xor	Performs **exclusive or** operations (returns true if either but not both of its operands is true—unless Null is one of its operands)
Eqv	Compares two logical values

Comparison Operators

Operator	***True if***	***False if***
<	Expression1 < Expression2	Expression1 >= Expression2
<=	Expression1 <= Expression2	Expression1 > Expression2
>	Expression1 > Expression2	Expression1 <= Expression2
>=	Expression1 >= Expression2	Expression1 > Expression2
=	Expression1 = Expression2	Expression1 <> Expression2
<>	Expression1 <> Expression2	Expression1 = Expression2

Some Built-in Functions

Type	**Syntax**	**Returns**	**Input**
Absolute value	Abs(x)	Value of the same type that is passed to it	
Exponential (e^x)	Exp(x)	Double	Double
Natural logarithm (ln x)	Log(x)	Double	Double, > 0
Square root	Sqr(x)	Double	Double, >= 0

Note: to square a number, use this syntax: x^2.
In general to raise a number to the power or an exponent, the syntax is *number^exponent*

Cosine	Cos(x)	Double	Double expressed in radians
Sine	Sin(x)	Double	Double expressed in radians
Tangent	Tan(x)	Double	Double expressed in radians
Inverse tangent (arctan x)	Atn(x)	Double	Double
Integer portion of a number	Fix(x)	Integer	Double
	Int(x)	Integer	Double

The difference between **Int** and **Fix** is that if *number* is negative, **Int** returns the first negative integer less than or equal to *number*, whereas **Fix** returns the first negative integer greater than or equal to *number*. For example, **Int** converts -8.4 to -9, and **Fix** converts -8.4 to -8.

Random number	Rnd(x)	Single	Single (see documentation for rules)
Sign of a number	Sgn(x)	1 if positive 0 if zero -1 if negative	any valid numeric expression
Rounding of a number	Round(expression,dec) where dec is number of decimal places desired		
Numeric value of a string	Val(string)	Number of appropriate type	String

Color Reference

Color	ColorIndex	RGB Values
Black	1	0,0,0
White	2	255,255,255
Red	3	255,0,0
Green	4	0,255,0
Blue	5	0,0,255
Yellow	6	255,255,0
Magenta	7	255,0,255
Cyan	8	0,255,255

Some Common Cell Properties

You can only set these properties from Visual Basic code.

ActiveCell.Borders or Range("F8").Borders or Cells(1,2).Borders…	.Color or .ColorIndex	Color := RGB(red,green,blue) where red, green, and blue are values between 0 and 255 ColorIndex = val, where val is an integer between 0 and 55
	.LineStyle	XlContinuous, xlDash, xlDashDot, xlDashDotDot, xlDot, xlDouble, xlSlantDashDot, xlLineStyleNone
	.Weight	XlHairline, xlThin, xlMedium, xlThick
ActiveCell.Font	.Bold	True or False
	.Color or .ColorIndex	Same as above
	.Italic	True or False
	.Name	A font name in quotes, such as "Courier"
	.Size	Numeric values (8-12 are common)
	.Subscript	True or False
	.Superscript	True or False
ActiveCell.Interior	.Color or .ColorIndex	Same as above
	.Pattern	XlPatternAutomatic, xlPatternChecker, xlPatternCrissCross, xlPatternDown, xlPatternGray16,xlPatternGray25, xlPatternGray50, xlPatternGray75, xlPatternGray8, xlPaternGrid, xlPatternHorizontal, xlPatternLightDown, xlPatternLightHorizontal, xlPatternLightUp, xlPatternLightVertical, xlPatternNone, xlPatternSemiGray75, xlPatternSolid, xlPatternUp, xlPatternVertical
	.PatternColor or .PatternColorIndex	Same as above