CHAPTER 9 DEBUGGING & ERROR MESSAGES

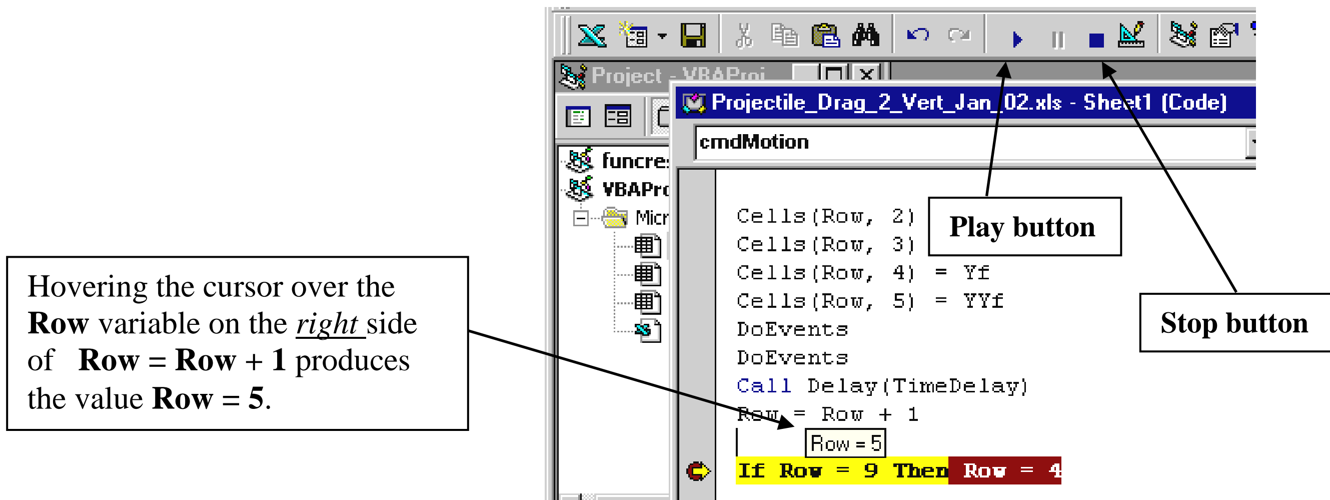
Tracking down the source of program bugs and error messages can be frustrating. Although a number of general approaches can be described, it is a difficult process to teach, and it remains a bit of an art that you'll develop only from experience. Two methods are considered below in a very cursory fashion; greater detail will be added in 2003.

9.1 Setting Breakpoints in the Code

Code execution can be interrupted, and variable values verified, by setting breakpoints at various locations. A **breakpoint** is created by clicking in the grey border on the left side of the code window. A large dot will appear and the entire code statement will be highlighted. When the program is now run execution will stop at the breakpoint. Hovering the cursor over any variable will produce a small box displaying the most recent value of the variable value. Clicking on the VCR-type **Play** button allows execution to continue. If the breakpoint is in a loop, the next iteration of the loop will occur and execution will again stop at the breakpoint. Multiple breakpoints can be set and the **Play** button used to shift to successive breakpoints. [To remove the breakpoint , click on the dot. If this doesn't work then the program is still in interrupt mode, press the VCR-type **Stop** button and then click on the dot.]

```
Cells(Row, 2) = t
Cells(Row, 3) = 0
Cells(Row, 4) = Yf
Cells(Row, 5) = YYf
DoEvents
DoEvents
Call Delay(TimeDelay)
Row = Row + 1

If Row = 9 Then Row = 4
```



Nonsensical values : If the cursor is hovered over the **Row** variable on the *left* side of **Row** = **Row** + **1** the value **Row** = **5** will also be displayed, which would seem to make no sense at all ! Similar values of variable **Row** would also be displayed in each of the **Cells** addresses a few lines above. The explanation for this bizarre algebra lies in the fact that **Row** is not really a variable in the usual sense, but represents a storage location that holds the current value of the variable. Obviously the storage location can only hold a single value which is the last value that the variable had at the moment that the breakpoint was encountered. In the code fragment displayed above{**Application # 10 b)** page 6-16} the value of **Row** was initially **4** in the **Cells** statements {such as **Cells(Row, 5)** = **YYf** }but this value was increased to **5** by the statement **Row** = **Row** + **1**. The breakpoint was then encountered and execution interrupted. At the time of the interruption the value stored in storage location **Row** was **5**; no matter where the cursor is hovered, the value of **Row** will be displayed as **5**.

Unfortunately, we often make use of combinations of loops that may involve hundreds of thousands of iterations and while program bugs may be evident in the first few iterations there are occasions in which anomalous behaviour may only occur well into the loops[1]. Obviously it is not practical to scroll through the iterations to get to the problem region by repeatedly pressing the **Play** button. A sneaky way around this problem is to use an **If** statement that only allows to breakpoint to be encountered near the region of interest. For example, suppose strange behaviour started occurring in **Application # 10 b)** at about 1.41 seconds. The following **If** statement would be bypassed until 1.4 seconds at which point the dummy statement **t = t** would be executed. Setting a breakpoint on the dummy statement would mean that the breakpoint would only be encountered at times greater than or equal to 1.4 seconds.

```
If t >= 1.4 Then
t = t
Else
End If
```

```
Cells(Row, 2) = t
Cells(Row, 3) = 0
Cells(Row, 4) = Yf
Cells(Row, 5) = YYf

If t >= 1.4 Then
t = t
Else
End If
```

9.2 Using Debug.Print Statements to Write Out Variable Values

The **Debug.Print** method can be used to print the value of any variable or constant into the **Immediate Window** of the **VB Editor**. The variable name is typed after the command **Debug.Print** (leaving a space in between); anything entered within a set of quotes will appear as text during the print out. The following statements could be added to **Exercise # 1 (Fooling Around)** in Chapter 1.

```
Debug.Print z
Debug.Print strString1 & z
Debug.Print "The value of x = " & x & "   The value of y = " & y
Debug.Print "The value of z = " & z
```

The output corresponding to these print statements appears in the **Immediate window** as :

```
Immediate
-28.35
The value of z is -28.35
The value of x = 4.1   The value of y = -2.2
The value of z = -28.35
```

To open the **Immediate Window** go into the **VB Editor** : **View menu ⇒ Immediate Window**

---

[1] such program bugs are usually not really programming bugs but problems with the way the model or simulation has been constructed -- it may work well in most situations but fail in a few specific situations . In a few of the incremental iteration project applications such as the E Field plot or anything involving gravitational forces and fields, the presence of a singularity can lead to extremely large values of force which affect the kinematics equations via a correspondingly huge acceleration. Large accelerations, even acting over small micrsecond intervals, can cause the moving object to discontinuously jump to a strange location. Those of you working on the E Field project will encounter the situation where the field lines plot normally until a charge is approached at which moment the moving object (and the line) jump far from the original region of the problem.

Once again the large number of iterations that are involved in many of our simulations can be problematic owing to the volume of output produced. Strategies using **If** statements that limit access to the **Debug.Print** statements (as described in section 9.1 for limiting breakpoint encounters) can be useful.

9.3 Dye Tracing --Using Debug.Print Statements to Trace Program Execution

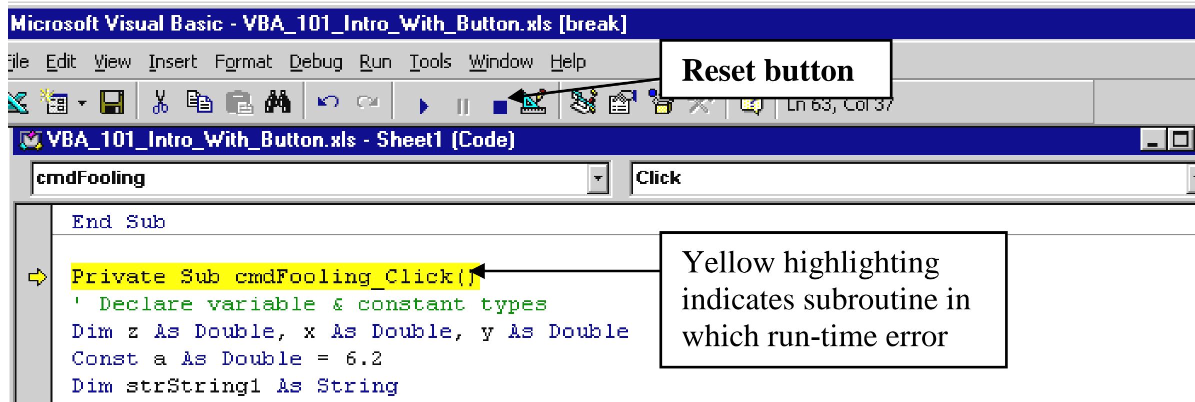
Occasionally your program will not seem to execute, yet no error messages are produced. Three common causes of this phenomenon are : i) the program is caught in an infinite loop (however, this would have been obvious due to the virtual screen freeze-up), ii) the program was not reset after an error message was produced, and iii) misspelled subroutine or event names that do not correspond to the control name [this usually occurs if you're typing in the subroutine shell instead of double clicking on the control to automatically create the proper shell and event]. When the control is activated, there is no code associated with it, so nothing happens. After having checked for these possible problems (as well as if you've included any write statements !) the next step is to determine exactly how far the program did execute. **Debug.Print** statements can be used to follow the course of program execution in a way that is entirely analogous to a physician injecting a dye into a patient's body prior to an X-ray in order to trace the flow. The procedure simply involves adding a number of **Debug.Print** statements that print out various text messages such as the following :

```
Debug.Print "cmdMotion sub --  point 1"
Debug.Print "cmdMotion sub --  point 2"
Debug.Print "cmdMotion sub --  point 3"
Debug.Print "cmdMotion sub --  point 4"
```

By placing one statement at the beginning of the subroutine, another at the end, and a few in-between, and then by moving the locations in later trials, it is possible to determine exactly where the program stopped executing.

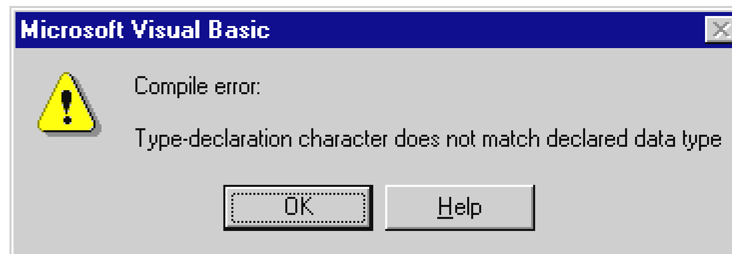
9.4 Common Error Messages

Error messages can occur either while code is being entered, in which case the code line changes to a red colour, or as the program is run. If the error occurs as the program is run, then clicking on the **OK** button of the error message box causes the first line of the subroutine in which the error occurred to be highlighted in yellow, and the actual problem error to be highlighted in blue. When a "run-time" error occurs the program must be reset before it can be run again by pressing the square **Reset** button (a VCR type Stop button which should be present as an icon on the toolbar, but which is also available in the **Run menu**).



Occasionally, if enough code is changed / added after an error, a message box appears informing that the program is being automatically reset.

Error # 1 :



The "**Help**" is generally of little value.

This message would have occurred in the Chapter # 1 exercise **Fooling_Around** if you neglected to include a space between the **x** and the **&**. The compiler adds a semicolon and tries to interpret the **x&;** as some data type.

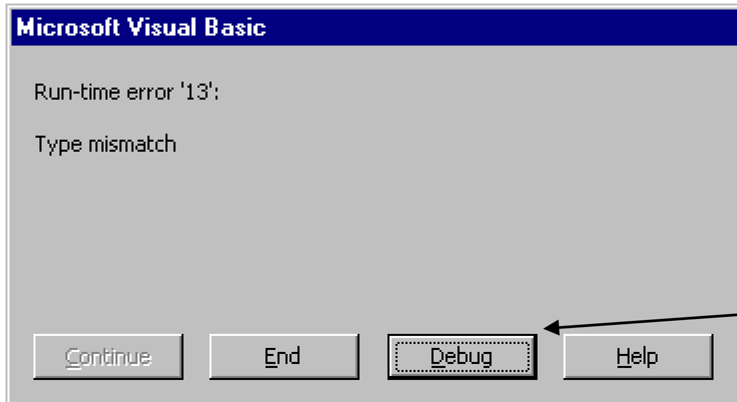
```
'  Calculate value of z
z = -x ^ 2 + a * y + 2.1

' Output statement
Debug.Print z
Debug.Print strString1 & z
Debug.Print "The value of x = " & x&; "   The value of y = " & y
```

Surprisingly, the compiler automatically inserts a space after an **&** (that is, you cannot leave it out, even if you try !).

Deciding whether a space is required when entering code will initially be confusing. Sometimes, such as with algebraic expressions, and in the presence of equal signs (=), the compiler automatically adjusts the spacing after the line has been entered; at other times you may encounter significant problems.

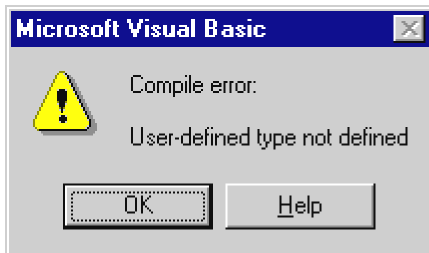
Error # 2 : If a variable is declared as a particular data type, and an attempt is made to assign a different data type value, a "**Type mismatch**" error will occur. In the Chapter # 1 exercise the variable **x** was declared as **Double** precision. Inadvertently entering the text **abcdef** into the cell from which the value of **x** is to be read generates an error.



	A	B	C	D
1		INPUTS		OUTPUTS
2	x =	abcdef		
3	y =	-2.2		
4				
5				
6	Run Fooling			
7				

Pressing **Debug** causes the offending statement to be highlighted in yellow.

Error # 3 : A spelling mistake in a data type declaration statement results in the compiler not recognizing the data type, and assuming that it is a custom type that the User should have defined.



VBA_101_Intro_With_Button.xls - Sheet1 (Code)

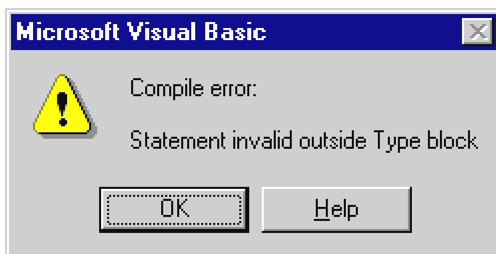
cmdFooling | Click

```
Private Sub cmdFooling_Click()
' Declare variable & constant types
Dim z As Double, x As Double, y As Double
Const a As Double = 6.2
Dim strString1 As Strig
```

Spelling mistakes.

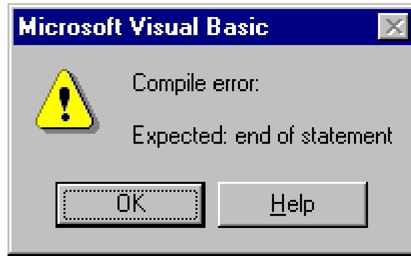
Dim z As Double, x As Dooouble, y As Double

Error # 4 : Omitting the **Dim** keyword results in the following message.



z As Double, x As Double, y As Double

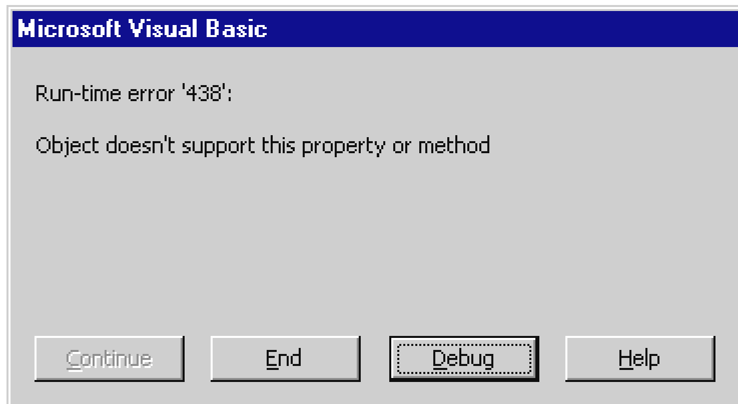
Error # 5 : Spelling **Constant** in full instead of using the abbreviation **Const** produces the error message :



Constant a As Double = 6.2

Should be **Const a As Double = 6.2**

Error # 6 : The following message can be produced by a number of different errors.




```
Worksheets("Sheet1").Range("D2").Value
Worksheets("Sheet1").Range("D2").Value = z
```

The first line, without the = **z** , is simply not doing anything and can't stand alone.

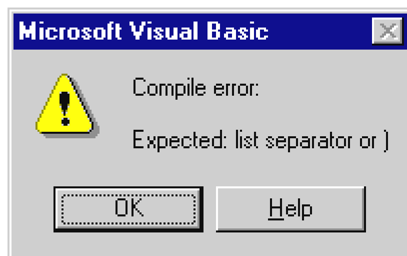
x = Worksheets("Sheet1").Cells("B2").Value

Using **Cells** requires a matrix type address such as **Cells(2, 2)**

x = Worksheets("Sheet1").Range("B2").Val

Misspelling : should be **Value** .

Error # 7 : Omitting the quotation mark confuses the compiler.

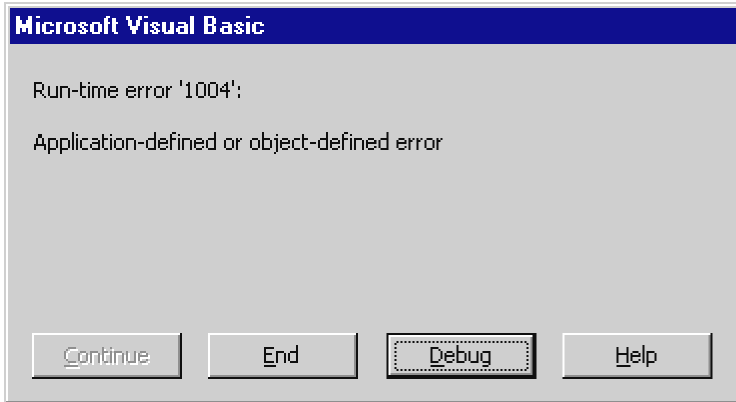


x = Worksheets("Sheet1").Range("B2).Value

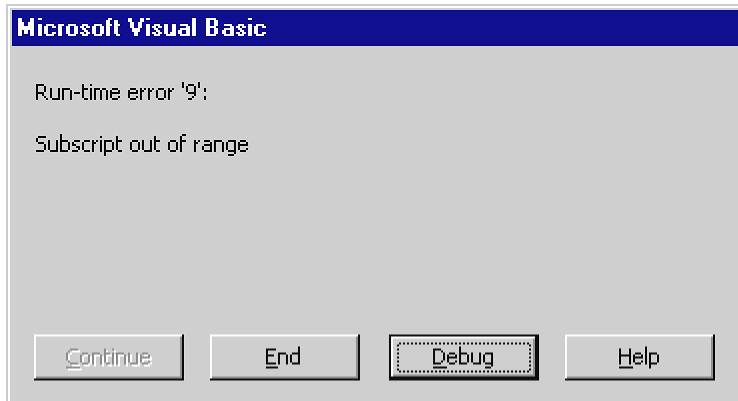
Should be **("B2")**

Error # 8 : An incorrect cell address causes **Run-time error '1004' :**

x = Worksheets("Sheet1").Range("B").Value

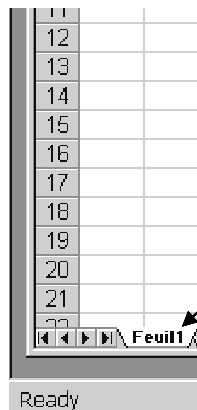


Error # 9 : Subscript out of range means that the worksheet being referred to is not recognized : In the first example below the particular sheet number was omitted; in the second example the use of a French version of Excel requires that **Sheet1** be replaced by **Feuil1** (however, it's much easier to right click on the **Feuil1 tab** at the bottom of the worksheet and **Rename** it as **Sheet1**).



x = Worksheets("Sheet").Range("B2")

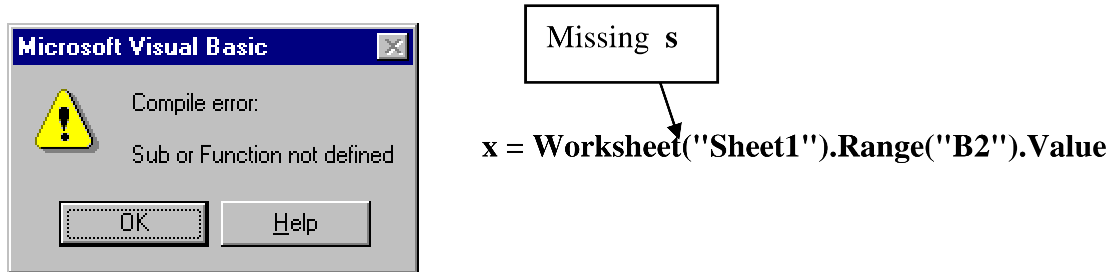
Should be **"Sheet1"**



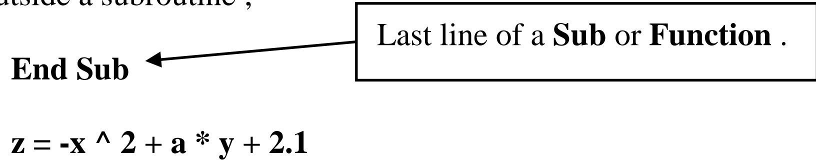
Rename as **Sheet1** or change reference in argument to **"Feuil1"**

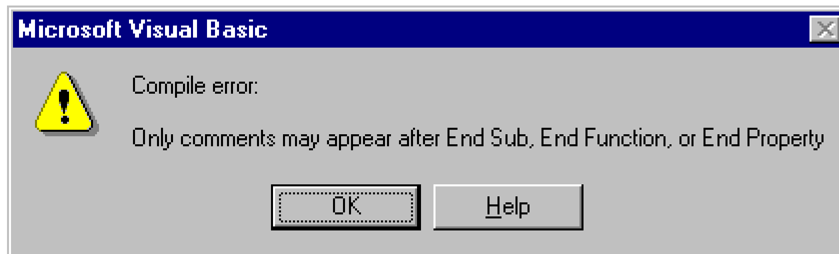
x = Worksheets("Sheet1").Range("B2").Value

Error # 10 : Leaving the "**s**" off Worksheet**s** causes the compiler not to recognize the object and assume that it's some function or subroutine that hasn't been defined.

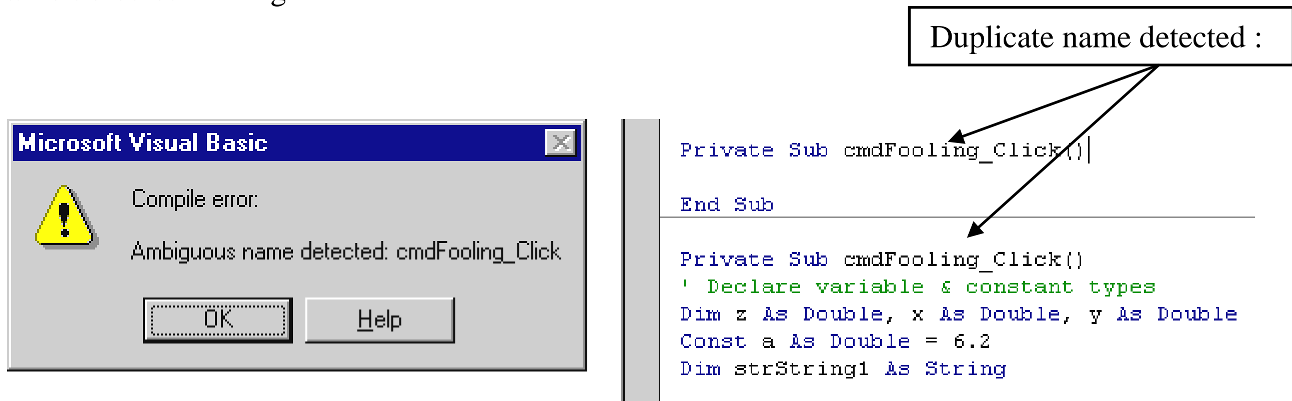


Error # 11 : Except for the **Option Explicit** and public data type declarations at the top of the code window, all code statements must be contained inside subroutine or function shells. The calculation statement shown below was located outside a subroutine ;





Error # 12 : You might inadvertently produce a second subroutine shell for some control. When the control is activated the compiler doesn't know which subroutine to execute and gives **an Ambiguous name detected** message .



Error # 13 : The "Nothing Happens When I Press the Button" Error

If nothing happens when you press a button (other than a small dotted box appearing in the button) The "fault" may be due to one of the following :

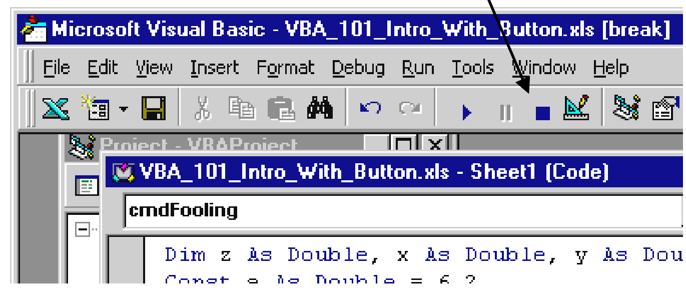
a) if you just corrected a bug after an error message was produced, you may have forgotten to reset the program by pressing the **Reset** button in the VB Editor ,

Press **Reset** after fixing a bug.

	A	B
1		INPUTS
2	x =	4.1
3	y =	-2.2

Run Fooling

Clear



b) the control's name does not match the subroutine name. Either check the control name by opening the **Properties** window for the control (enter **Design Mode** and click once on the control), or double click on the control (in Design mode) and check that you are brought into the proper subroutine (if a new subroutine shell is created then you must have misnamed the original shell name). It's always best to create subroutine shells by double clicking on the control; otherwise you may misspell it when typing.

Private Sub cmdFoooling_Click()

General Cautionary Notes :

Save frequently to avoid disappointment when a screen freeze up occurs.

Always keep a back-up diskette up to date.

Work on only one Excel workbook file at a time; if 2 or more files are open you may inadvertently end up working on (and corrupting) the code of the wrong file (which can easily happen if the code is similar). Always check that only one file is open; if you're copying and pasting code, close the source file before pasting. Note : when a copy of a workbook is created via **File menu ⇒ Save As** the original file is automatically closed and only the newly named file remains.

INDEX OF ERROR MESSAGES :

Compile Error : Ambiguous name detected	**Error # 12**
Compile Error : Expected : end of statement	**Error # 5**
Compile Error : Expected : list separator or)	**Error # 7**
Compile Error : Only comments may appear after End Sub, End Function…	**Error # 11**
Compile Error : Statement invalid outside Type block	**Error # 4**
Compile Error : Sub or Function not defined.	**Error # 10**
Compile Error : Type-declaration character does not match declared data type	**Error # 1**
Compile Error : User-defined type not defined	**Error # 3**
Run-time error '9' : Subscript out of range	**Error # 9**
Run-time error '13' : Type mismatch	**Error # 2**
Run-time error '438' : Object doesn't support this property or method	**Error # 6**
Run-time error '1004' : Application-defined or object-defined error	**Error # 8**
The "Nothing Happens When I Press the Button" Error	**Error # 13**