

Write Your Own Sub Procedures and Functions in Visual Basic 6

Sometime after a beginner finishes with their first program, they either ask themselves or me about the need or importance of writing their own Procedures (in Visual Basic there are two types of Procedures---Subprocedures---I'll discuss the differences shortly.)

I should tell you right up front that if you *never* write a procedure or function of your own, the world won't end, and your programs will still run---and just as efficiently. For those of you who read my book, Learn to Program with Visual Basic, you may have noticed that it wasn't until the final chapter of the book that I took the code I had written in the previous 14, and placed some of it in my own Procedures. Placing the code in Procedures had no impact on how fast the program ran.

Then why bother?

Benefits of writing your own Procedures

The real benefits of taking code that you have written, and placing it in Procedures of your own is threefold: Modularity, Portability, and Readability.

Modularity

When you start to write your own Procedures, your code naturally becomes very modular. What does that mean?

In short, Modularity is a programming term that means that the code in a procedure (let's say an event procedure for the sake of discussion) is short and to the point. In Visual Basic, as you know, we write code and place it in event Procedures so that when an event is triggered (usually by the user doing something, such as clicking the mouse or entering something into a TextBox), that code is executed. If you look in an event procedure, and see several hundred lines of code, most likely that code is not modular. Modular code seldom (there are exceptions to any rule!) exceeds the length of your code window, and should perform just a single function.

This is a big joke to my students in my university classes, because whenever I hear or see someone proclaim that the code in a procedure should perform only one function, I dare that person to define a function for me. It sounds simple, but it really isn't.

For instance, if you are writing a program to process payroll, is a single function the calculation of the employee's Net Pay. No, that's too broad. As a programmer, you need to break that process into three 'modules'---the calculation of Gross pay, the calculation of payroll withholding taxes, and the calculation of net pay.

But even that's too broad. In the United States, the calculation of payroll taxes requires separate calculations of Social Security Taxes, Federal Taxes, State Taxes and local taxes. Theoretically then, each one of these calculations should appear in a separate module or procedure of their own---one for each one of these tax calculations.

Purists of modular programming would even argue that a payroll program should have a separate module or procedure written for each State's tax calculations. Since each State in the United States taxes payroll differently (some states don't tax at all, some tax a flat rate percentage, some have a graduated tax calculation), this makes a lot of sense. Trying to 'lump' all fifty states' tax calculations into a single module or procedure would make that procedure very long indeed.

Portability

Portability just means that the code in a procedure can be used, whole and intact, with no modifications, in another program. For instance, if you write some code that checks to see if a PC has a CD-ROM drive attached, if you place that code in a procedure called `IsThereACDROM`, you should be able to include that code, without modification, in any other program you write. That's portable code. Better yet, if you place that code, and other code like it, in a Visual Basic Standard Module, all you need to do is include the Standard Module in every Visual Basic program you write, and you access to every clever procedure you'll ever write. That's portable code, and it's a big advantage that comes when you take the clever code you write and place it in Procedures of your own.

Readability

The third benefit of writing your own Procedures is readability. I've already mentioned that a by product of writing your own Procedures is modularity---the length of your code becomes shorter---both in the event Procedures that call the Procedures that you write, and in the Procedures themselves. Procedures that are short in the number of lines of code written are just naturally easier to read---and code that is easier to read is code that is easier to modify---either by you, the original programmer, or by the person who may modify it in the future.

Types of Procedures

There are two types of Procedures in Visual Basic---SubProcedures and Functions. Both types of Procedures do **something**. The only difference between the two is that a Function **returns** a value to the Procedure that calls it, and a Subprocedure does not. Let me illustrate each one for you with a very simple code example---we don't need to get too fancy here.

Subprocedures with no arguments

Here's an example of some code that we can place in the click event procedure a command button which adds two numbers, and displays the result in a message box. Again, nothing fancy here, but it will illustrate my point perfectly.

```

Private Sub Command1_Click()

Dim intFirst As Integer
Dim intSecond As Integer
Dim intResult As Integer

intFirst = 3
intSecond = 19

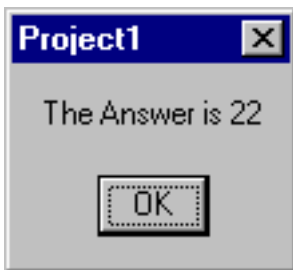
intResult = intFirst + intSecond

MsgBox "The Answer is " & intResult

End Sub

```

All I'm doing here is declaring three Integer type variables, assigning values to two of them, adding them and assigning the result to a third variable. At that point, I then display the value of their variable in a Message Box. If we run this program, and click on the command button, this is what we'll see...

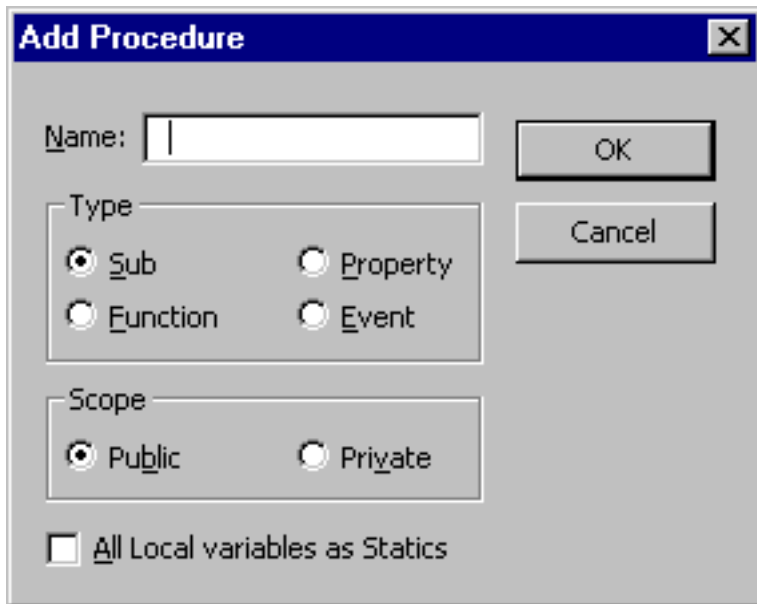


Now let's say we decide that this code is a perfect candidate to be placed in a procedure. How do we do that?

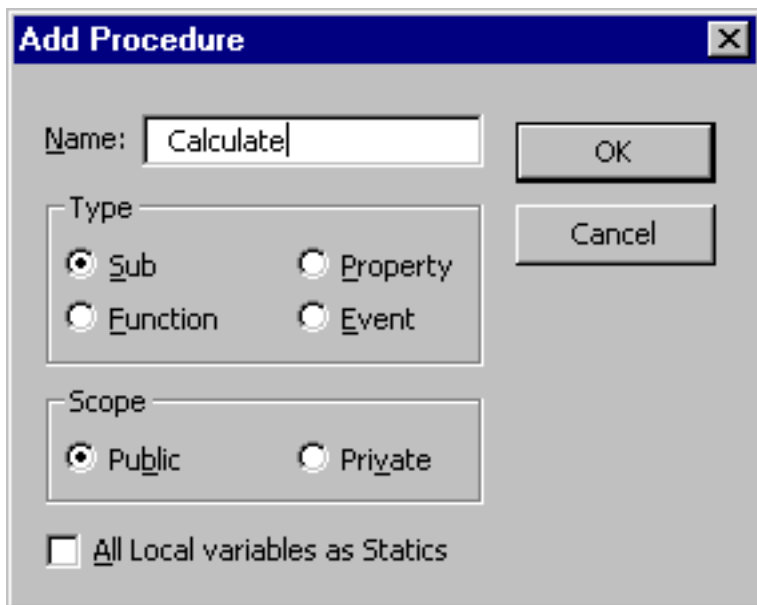
Let's work with a Subprocedure first. Remember, a Subprocedure is code that does something, without returning a value to the calling Procedure.

In Visual Basic, there are two ways to create your own Procedures.

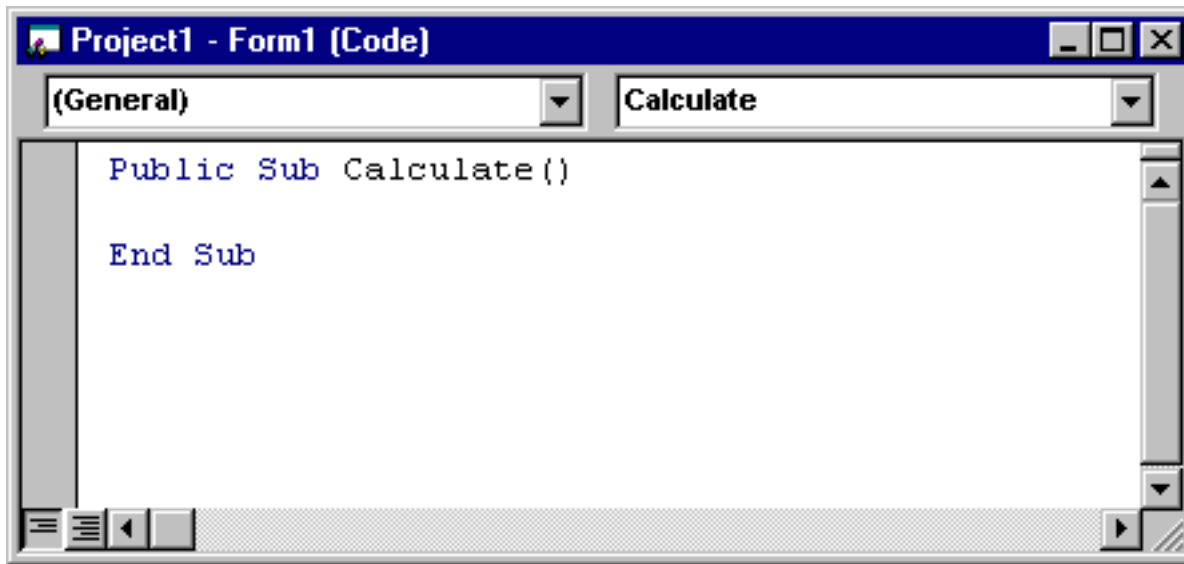
With your Code Window open, select Tools-Add Procedure from the Visual Basic Menu Bar, and the following dialog box will appear.



Let's call the Procedure Calculate (use Mixed case in naming your procedures), and leave the Type as 'Sub' for Subprocedure, and the Scope as Public.

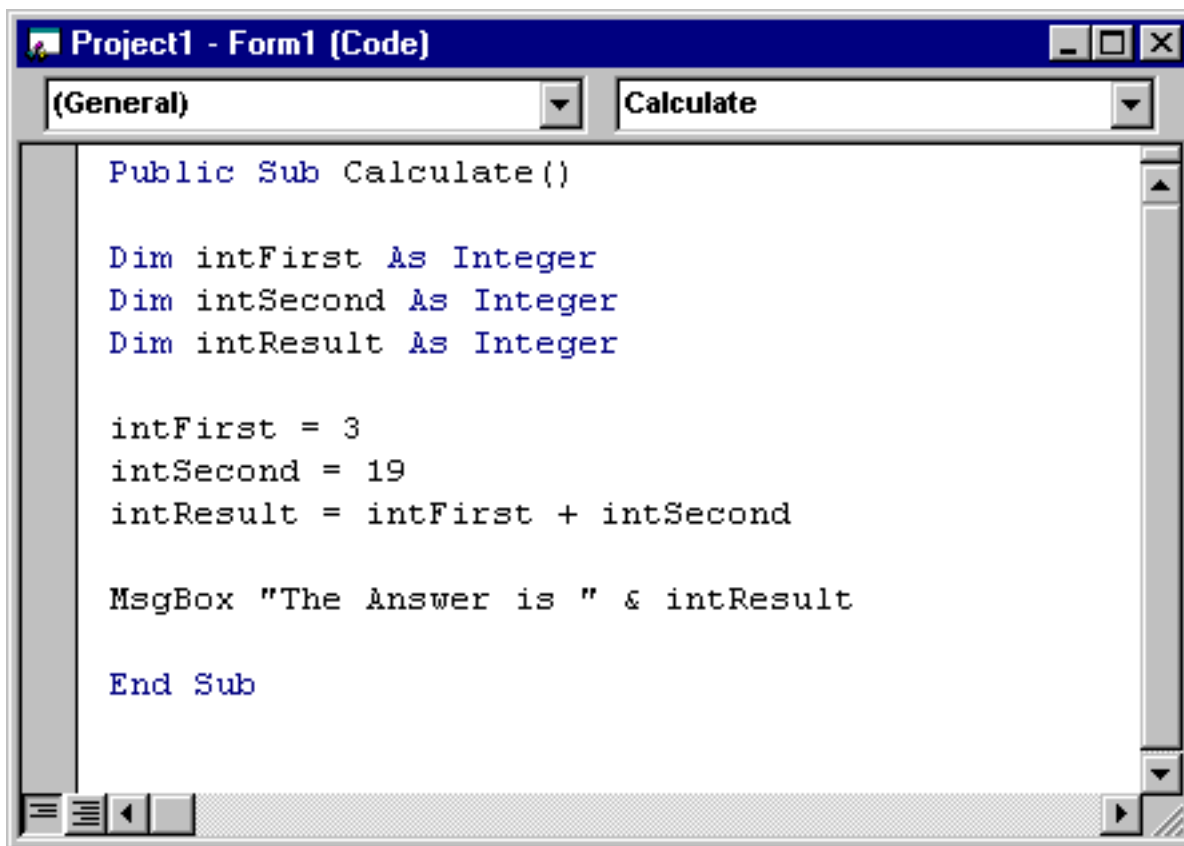


When you click on the OK button, Visual Basic will create a Subprocedure for you in the form called 'Calculate', and you'll then see it in the Code Window.



It's worth noting here that we could also create this procedure in a Standard Module, in which case it would be easily accessible and executable from every form in our application.

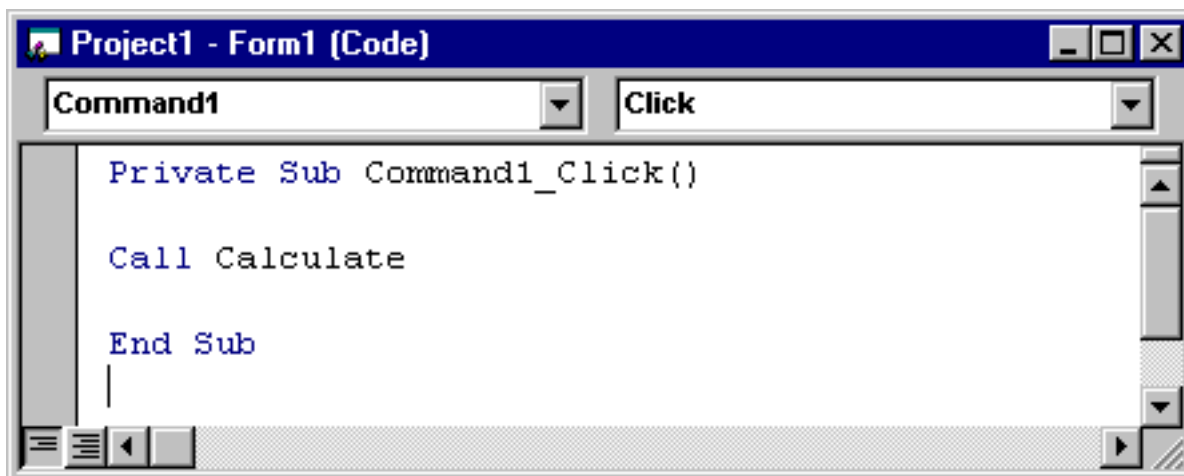
What we need to do at this point is take the code out of the Click Event Procedure of the command button, and place it in the 'Calculate' Subprocedure. We can do that by selecting the code in the click event procedure, and 'cutting and pasting' it into the Subprocedure. This is the way the Subprocedure should read after we have done that.



The screenshot shows the Visual Basic IDE with the 'Project1 - Form1 (Code)' window open. The 'General' tab is selected, and the 'Calculate' dropdown menu is visible. The code in the editor is as follows:

```
Public Sub Calculate()  
  
    Dim intFirst As Integer  
    Dim intSecond As Integer  
    Dim intResult As Integer  
  
    intFirst = 3  
    intSecond = 19  
    intResult = intFirst + intSecond  
  
    MsgBox "The Answer is " & intResult  
  
End Sub
```

Now we need to add a single line of code to the click event procedure of the command button to call the Subprocedure. There are two ways to call the Subprocedure---with the Call Statement or without. I prefer to use the 'Call' statement because it alerts me to the fact that 'Calculate' is a procedure of my own.



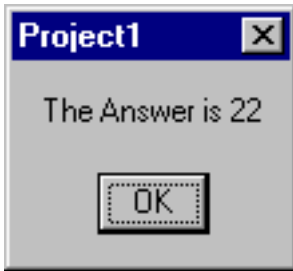
The screenshot shows the Visual Basic IDE with the 'Project1 - Form1 (Code)' window open. The 'Command1' dropdown menu is selected, and the 'Click' dropdown menu is visible. The code in the editor is as follows:

```
Private Sub Command1_Click()  
  
    Call Calculate  
  
End Sub
```

You can't see it in this screen shot, but when I entered the code in the clicked event procedure of the command button, I typed the Subprocedure name in lower case. Visual Basic should change the case to Mixed case (which is how I name all of my procedures). If Visual Basic leaves the name in all lower case, then you know that you have misreferenced the name.

If we then run the program, and click on the command button, the code in the click event

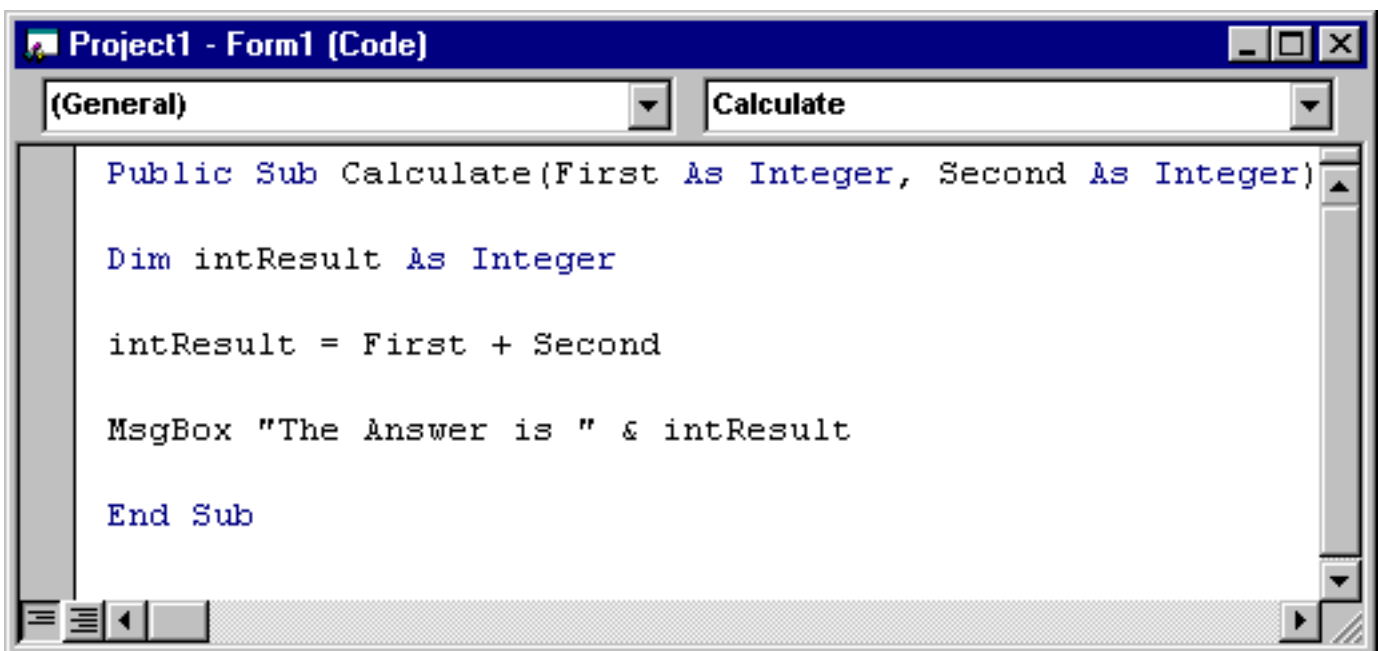
procedure finds the Subprocedure called 'Calculate' in the General Declarations Section of the form, runs it, and displays this message box.



Subprocedures with one or more arguments

You can argue that this Subprocedure, while it does the job, is pretty inflexible. The answer will always be the same--22.

Let's modify the Subprocedure to accept arguments for the two numbers to be added, which will make it much more flexible. To do that, we need to modify the Subprocedure to look like this..



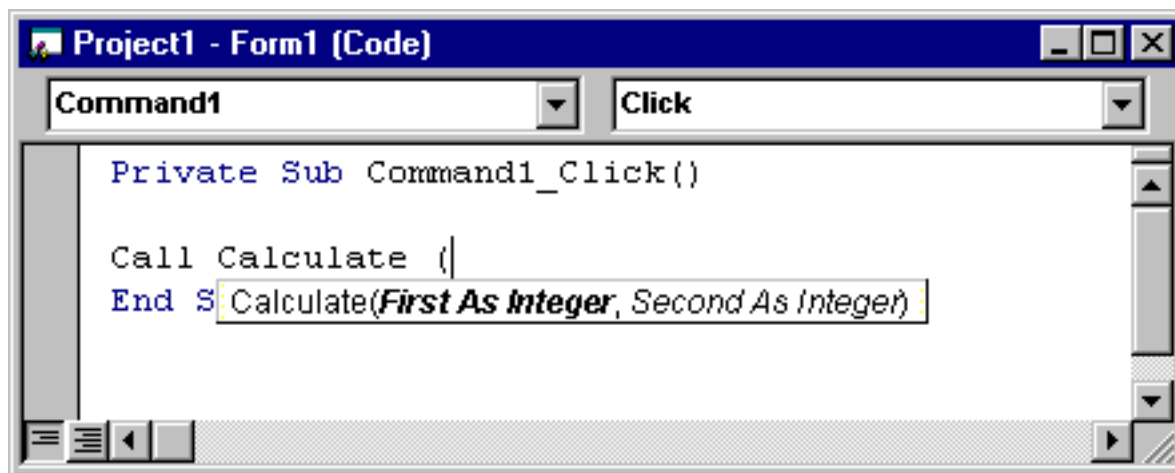
We've changed a number of things here.

Notice that the subprocedure header has changed---instead of an empty set of parentheses, we're now telling Visual Basic that when this Subprocedure is called, it will be 'passed' two Integer arguments. The names 'First' and 'Second' are totally arbitrary---we could the arguments anything we want. Notice also naming the arguments within the Subprocedure header servers as the declaration for the arguments---there's no need for a separate declaration of them in the body of the Subprocedure.

For that reason, the only 'Dim' statement remaining is for the declaration of the variable `intResult`. The values for `First` and `Second` will be passed when we call the Subprocedure. Let's call it now, asking it to calculate the addition of 5 and 33.

I've been asked many times "When do I need parentheses and when don't I?" when calling procedures.

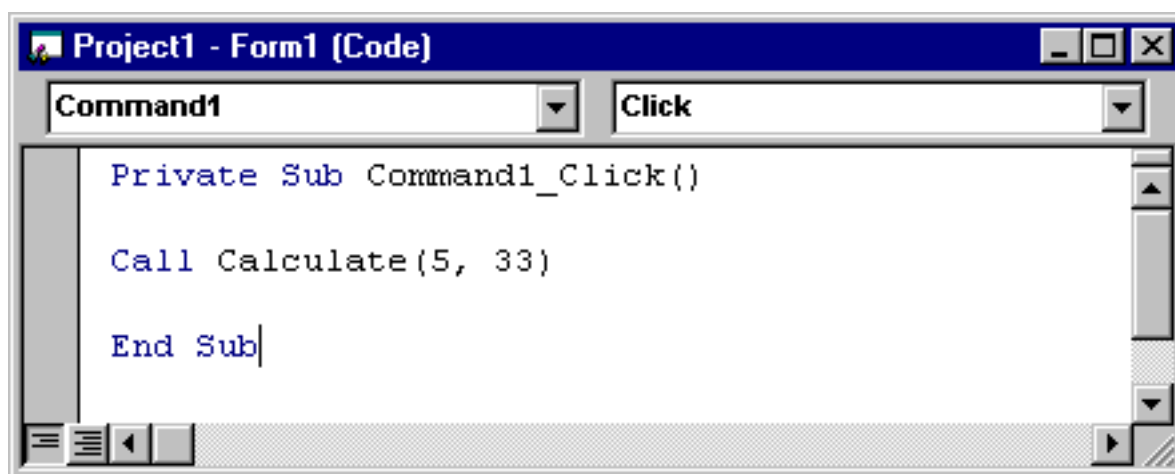
In short, if you use the Call Statement to call a procedure, and you are passing it an argument, you need to place the arguments in parentheses. Notice that as soon as I type the first parenthesis, Visual Basic gives me a 'hint' as to the number and type of the arguments required by the Subprocedure.



The screenshot shows the Visual Basic Code Editor window titled "Project1 - Form1 (Code)". The "Command1" dropdown menu is selected, and the "Click" event is chosen. The code in the editor is as follows:

```
Private Sub Command1_Click()  
  
Call Calculate (  
End S: Calculate(First As Integer, Second As Integer)
```

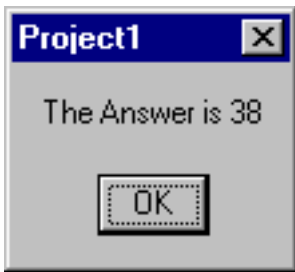
Pretty clever, that Visual Basic editor! How can we go wrong now, as we now know we need two arguments, both Integer data types. Here's the statement to ask the Calculate Subprocedure to tell us what 5 plus 33 is...



The screenshot shows the Visual Basic Code Editor window titled "Project1 - Form1 (Code)". The "Command1" dropdown menu is selected, and the "Click" event is chosen. The code in the editor is as follows:

```
Private Sub Command1_Click()  
  
Call Calculate(5, 33)  
  
End Sub
```

If we now run the program, Visual Basic will display this message box.

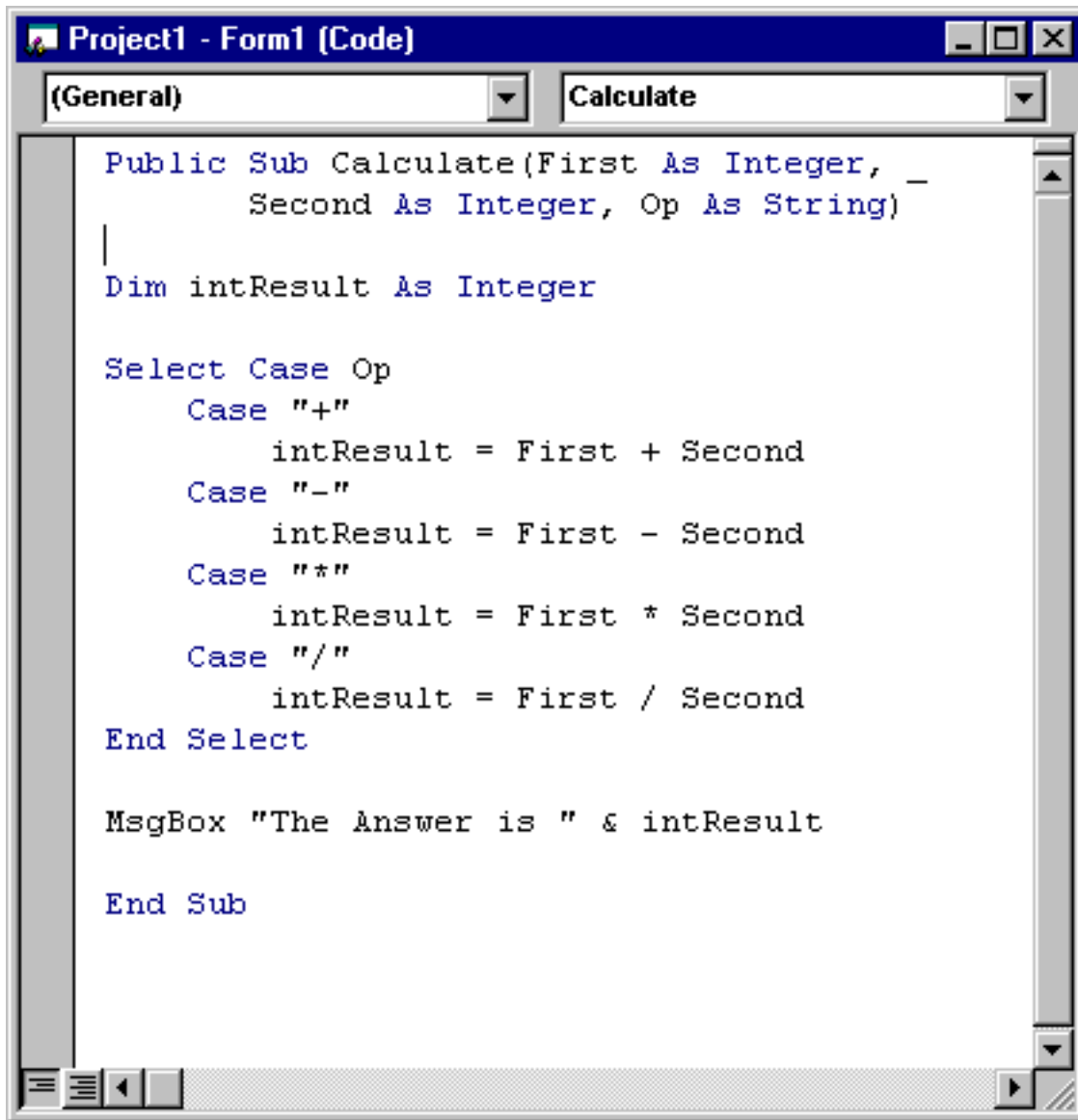


There, that's better. Now we've produced a very flexible Subprocedure.

Subprocedures with one or more arguments

Suppose we want to take this a step further, and allow the user of our procedure to **optionally** specify the mathematical operation they wish to perform---that is, addition, subtraction, multiplication or division. Furthermore, if they don't supply an operator, let's presume they want to perform addition.

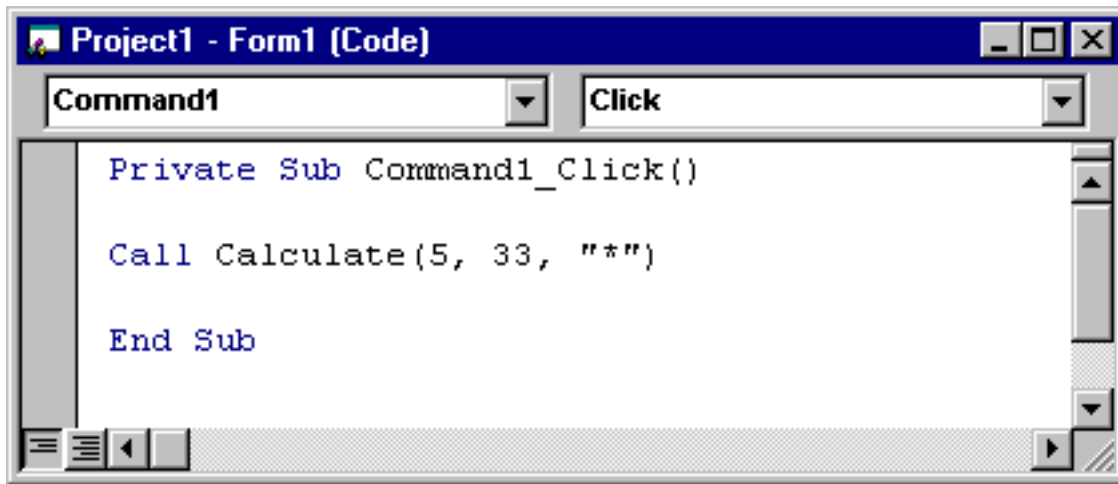
Before I show you how to declare an optional argument in a Subprocedure, let's first modify the Subprocedure by creating a mandatory third argument called 'Op', and adding a Select...Case structure, like this.

A screenshot of the Visual Basic IDE showing the code for a subroutine named Calculate. The window title is "Project1 - Form1 (Code)". The "General" tab is selected, and the "Calculate" control is chosen. The code is as follows:

```
Public Sub Calculate(First As Integer, _  
    Second As Integer, Op As String)  
|  
    Dim intResult As Integer  
  
    Select Case Op  
        Case "+"  
            intResult = First + Second  
        Case "-"  
            intResult = First - Second  
        Case "*"  
            intResult = First * Second  
        Case "/"  
            intResult = First / Second  
    End Select  
  
    MsgBox "The Answer is " & intResult  
  
End Sub
```

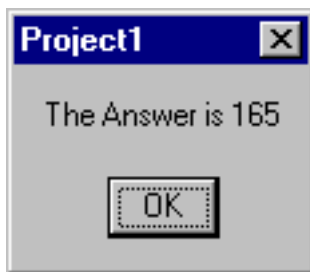
Notice that I have broken the Subprocedure header into two lines using the Visual Basic line continuation character--the hyphen (_). Remember, you can use the line continuation character anywhere within Visual Basic **except** in the middle of a quoted string.

Here's the code to call the Calculate Subprocedure which will display the result of 5 multiplied by 33.



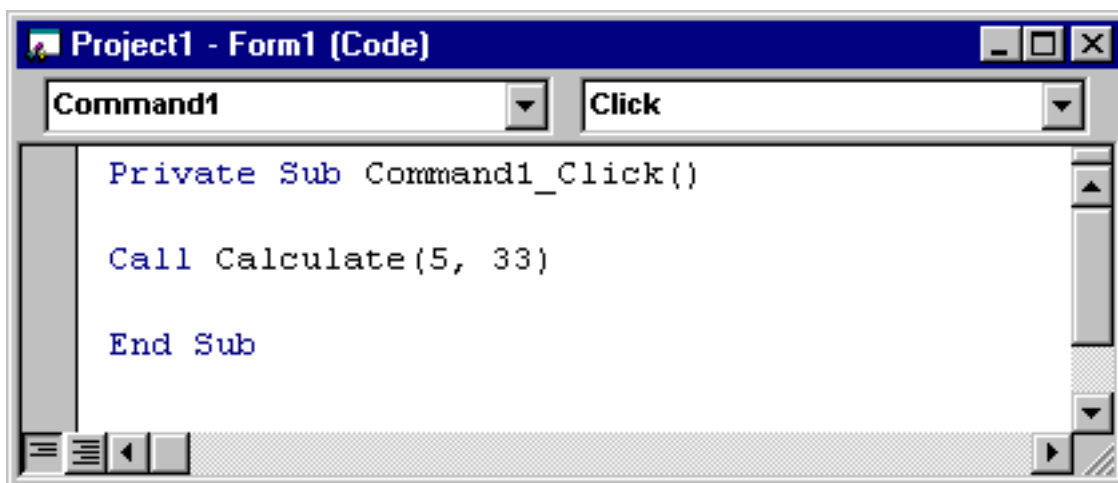
Notice that because I declared the third argument 'Op' as a string argument, the Call statement needs to pass the third argument as a string (that's why the asterisk is in quotation marks.)

If we now run this program, you'll see the following message box.



By the way, what happens if you don't pass the Subprocedure the proper number of arguments?

Let me remove the third argument from the Call Statement ...



.. run the program, and click on the command button. You receive a nasty Visual Basic

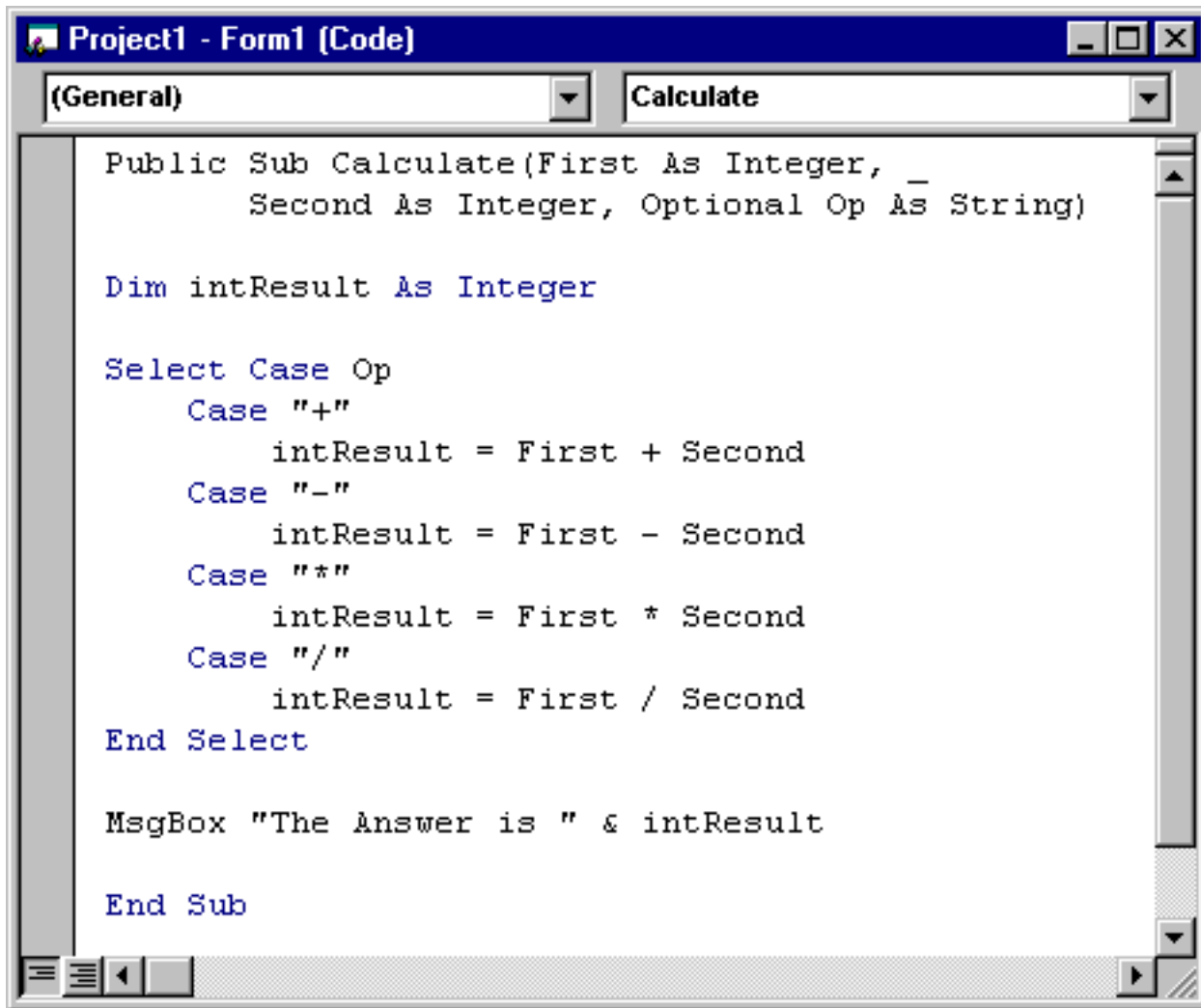
message like this...



which is Visual Basic's way of saying, "The Subprocedure needs three arguments---you passed me two."

Now back to this Optional business.

To create an Optional argument, all you need to do is precede the argument name in the Procedure heading with the word 'Optional'. Optional arguments must appear last in the Procedure header---in other words, we can't declare an Optional argument, and then follow it up with a mandatory one. You can have as many optional arguments as you need, but they must be declared after all of the mandatory arguments have been declared. Here's the modified Subprocedure header to tell Visual Basic that 'Op' is an optional argument.



```

Public Sub Calculate(First As Integer, _
    Second As Integer, Optional Op As String)

    Dim intResult As Integer

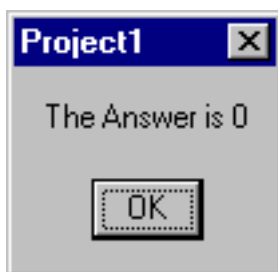
    Select Case Op
        Case "+"
            intResult = First + Second
        Case "-"
            intResult = First - Second
        Case "*"
            intResult = First * Second
        Case "/"
            intResult = First / Second
    End Select

    MsgBox "The Answer is " & intResult

End Sub

```

Notice the use of the word 'Optional' before the third argument 'Op'. If we now run the program, and call this Subprocedure with just the two required arguments, here's what we'll see...



The program doesn't bomb--but neither does it do what we intended.

The answer shouldn't be 0! Remember, we had intended that if the third argument wasn't supplied, that we would presume the user wanted to perform addition. The answer should be 38.

We need a way to determine if the user supplied us with the Optional argument, and any time that you declare a procedure with Optional arguments, you need to take into

account what your code should do if the optional arguments are not supplied.

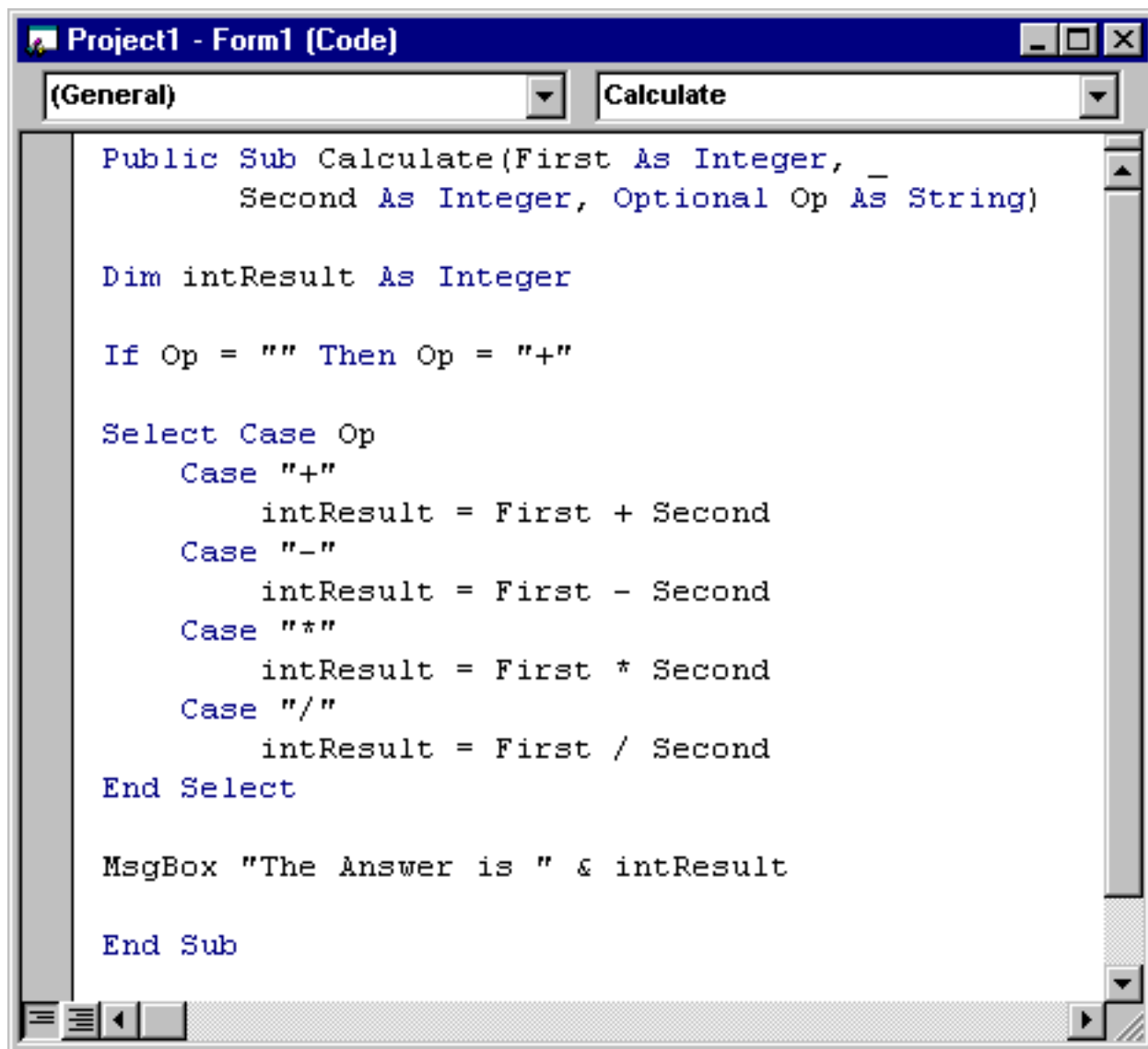
How can we check for a non-supplied Optional argument?

To do that, we'll need to make a slight change in our code, and there are two ways of doing that.

First, we can check to see if the Op argument is a empty string. If it is, we can then just assign the plus sign to it like so...

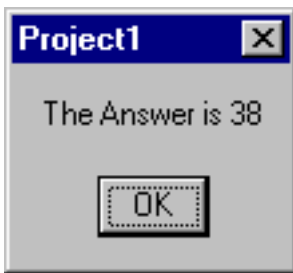
```
If Op = "" then Op = "+"
```

Here's the modified Subprocedure.

The image shows a screenshot of the Visual Basic Code Editor window titled "Project1 - Form1 (Code)". The window has a menu bar with "General" selected and a dropdown menu showing "Calculate". The code in the editor is as follows:

```
Public Sub Calculate(First As Integer, _  
    Second As Integer, Optional Op As String)  
  
    Dim intResult As Integer  
  
    If Op = "" Then Op = "+"  
  
    Select Case Op  
        Case "+"  
            intResult = First + Second  
        Case "-"  
            intResult = First - Second  
        Case "*"  
            intResult = First * Second  
        Case "/"  
            intResult = First / Second  
    End Select  
  
    MsgBox "The Answer is " & intResult  
  
End Sub
```

Now if we run the program, and pass the Subprocedure just the two required arguments, the Calculate Subprocedure is smart enough to presume addition, as we get this result.



That's better. I mentioned that there are two ways to check to see if the optional argument has been supplied.

The second way is to use the Visual Basic function `IsMissing`, which was designed to do exactly what we did ourselves a minute ago in code. One *quirk* of the `IsMissing` function, however, is that it only works properly if the Optional Argument is declared as a Variant. That means we need to change the Optional variable declaration of 'Op' to a Variant, in addition to using the `IsMissing` function in our code.

```

Public Sub Calculate(First As Integer, _
    Second As Integer, Optional Op As Variant)

    Dim intResult As Integer

    If IsMissing(Op) Then Op = "+"

    Select Case Op
        Case "+"
            intResult = First + Second
        Case "-"
            intResult = First - Second
        Case "*"
            intResult = First * Second
        Case "/"
            intResult = First / Second
    End Select

    MsgBox "The Answer is " & intResult

End Sub

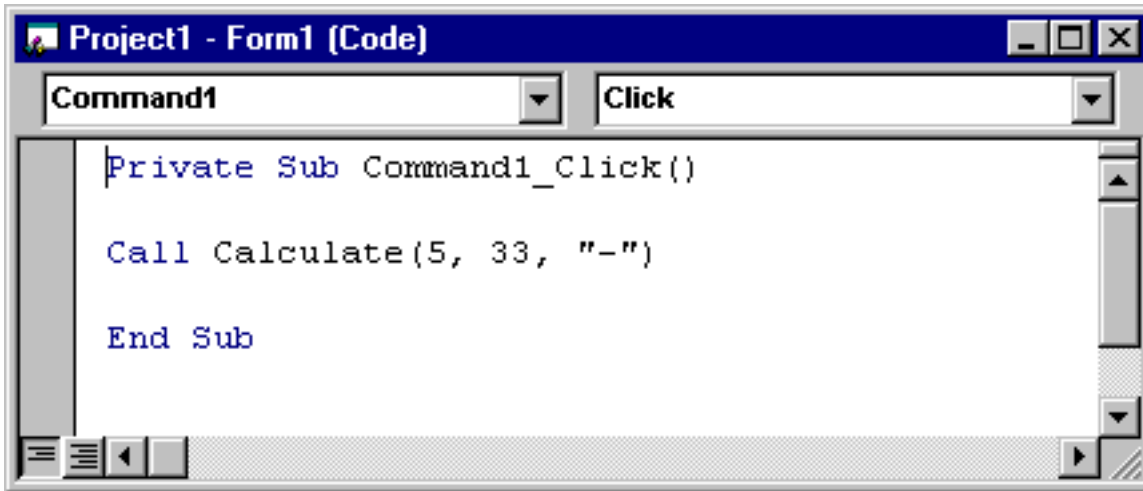
```

Notice the differences here---we've changed the declaration of the Optional argument

'Op' to a Variant---and now we're using the IsMissing function to determine if the Optional third argument has been passed to the procedure.

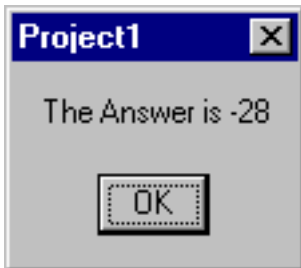
If we run this program now, the result is correct---a message box with the value 38 in it.

By the way, don't forget to ensure that the program behaves properly with all three arguments passed. Let's change the Call statement to calculate 5 minus 33 like this...



```
Project1 - Form1 (Code)
Command1 Click
Private Sub Command1_Click()
    Call Calculate(5, 33, "-")
End Sub
```

The answer should be -28...



Functions

Having spent a good deal of time discussing Subprocedures, you may think that creating your own functions is going to be a great deal more difficult.

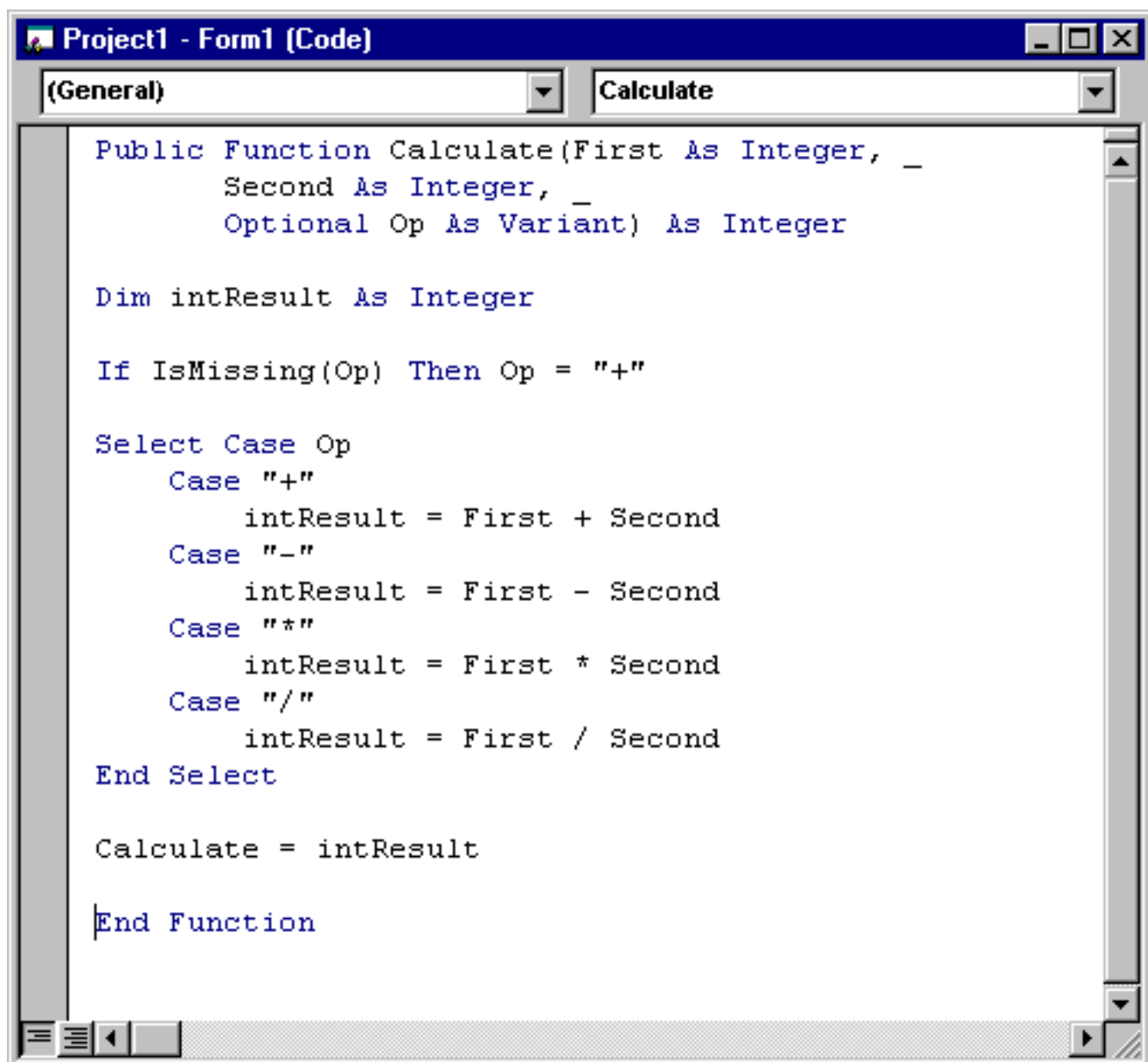
Not at all!

As I mentioned earlier, the only difference between a Subprocedure and a function is that the fact that the Function returns a value to the calling procedure.

Let's take the Calculate Subprocedure we've already written, and take the responsibility of displaying the result away from it, and instead return the result to the event procedure that calls it. As soon as we talk about returning a result, we know that we're dealing with a function. There are a few administrative changes that we'll need to make, like changing

the word 'Sub' to Function in the Procedure Header and the Procedure trailer. And, because a function returns a value, we need to declare the type of the return value in the header. Finally, the way that a return value is passed from the function back to the calling procedure is by assigning the return value to the name of the function somewhere within the body of the function. Here's the code for the function.

I created this function by modifying the existing subprocedure 'Calculate'. If you want, you can create the function 'by scratch' by selecting Tools-Add Procedure from the Visual Basic menu bar, and specifying 'Function' as the procedure type.



The screenshot shows the Visual Basic IDE with the 'Project1 - Form1 (Code)' window open. The 'General' tab is selected, and the procedure 'Calculate' is chosen from the dropdown menu. The code in the editor is as follows:

```
Public Function Calculate(First As Integer, _  
    Second As Integer, _  
    Optional Op As Variant) As Integer  
  
    Dim intResult As Integer  
  
    If IsMissing(Op) Then Op = "+"  
  
    Select Case Op  
        Case "+"  
            intResult = First + Second  
        Case "-"  
            intResult = First - Second  
        Case "*"  
            intResult = First * Second  
        Case "/"  
            intResult = First / Second  
    End Select  
  
    Calculate = intResult  
  
End Function
```

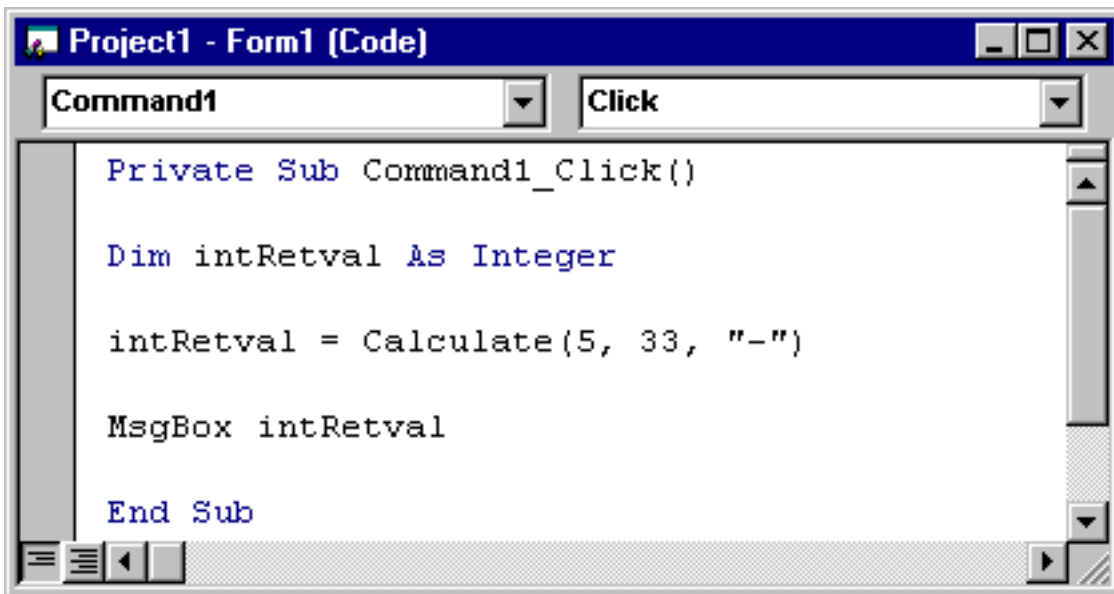
Take note of the following changes:

1. The procedure header and trailer now read **'Function'** instead of **Sub**.
2. **'As Integer'** appears at the end of the Function Header. That's the declaration for the return type.
3. This line of code

Calculate = intResult

 appears in the body of the Function. That's the assignment of the return value taking place.
3. The **Msgbox** statement has been removed from the code. We'll take care of this in the calling procedure.

Speaking of the calling procedure, let's change the code in the click event procedure to look like this:



```

Private Sub Command1_Click()

    Dim intRetval As Integer

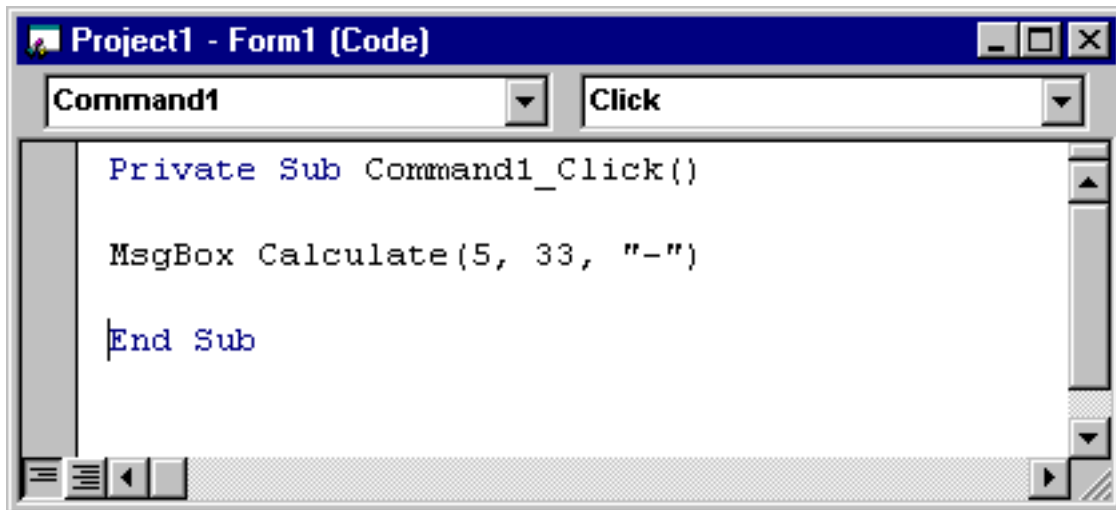
    intRetval = Calculate(5, 33, "-")

    MsgBox intRetval

End Sub
  
```

Since 'Calculate' is now a Function, we need to handle its return value. We can do that in two ways.

1. Assign the return value to a variable, as we're doing in the code above or
2. Use the return value in an expression, like this ...



Either way, if we now run this program, and click on the command button, this message box will be displayed...



That's perfect.

As I frequently say, there's more than one way to paint a picture. You've seen here that we were able to achieve the same functionality in our program using either a Subprocedure or a function.

Summary

Subprocedures and Functions can make your program modular, portable, and very readable---not to mention very powerful. After you are certain that your program is working properly, take a look at any code you have written which may be a candidate for inclusion in a function or a procedure of its own.