Introduction

Objectives

- To learn how to define and call Function procedures.
- To understand the concept of "Call by Value" and "Call by Reference."
- To learn when to use "Call by Reference" when defining Sub procedures.

Class-Level Variables

In the previous unit, you were introduced to the syntax and role of subprograms in Visual Basic. Besides the built-in function procedures, it is possible for the programmer to create "user-defined" functions. In Visual Basic, these are called "Function procedures."

Of course, you may ask, "Why can’t I just define my subprogram as a Sub procedure?" Technically, it is possible to do so. However, to appreciate the usefulness of Function procedures better as well as to understand how to address similar problems using Sub procedures, consider the following problem:

Write a subprogram that calculates the average of three real numbers.

A proposed solution based on what has been described in the previous unit is the following Sub procedure:

```vbnet
Public Class Form1
    Dim avg As Double

    Sub Average_of_3(x As Double, y As Double, z As Double)
        avg = (x + y + z) / 3
    End Sub
```
This method obtains three values from three TextBoxes that you can assume are components of a suitable windows form. The intent is to display the result in a fourth TextBox, TextBox4.

While this approach does calculate the average of three numbers, the problem is that the result, `avg`, must be accessible to both subprograms. In order to make this possible, `avg` must be declared in a `Dim` statement that is outside the body of either Sub procedure. This is referred to as making it a "class-level" variable. However, this is not usually a good practice because it makes the variable vulnerable to unintentional change.

A more satisfactory way to communicate the value that is computed in a subprogram to a calling program is by associating the value with the name of the subprogram. This is, in effect, the purpose of a function.

**Function Procedures**

In the previous example the Sub procedure, `Average_of_3`, computes the average of three values and returns the result using a class-level variable, `avg`. If a different type of subprogram is used, however, the need for the class-level variable can be eliminated. Specifically, `Average_of_3` can be defined as a "Function" procedure:

```vbnet
Function Average_of_3( x As Double, y As Double, _
    z As Double) As Double
    Return (x + y + z) / 3.
End Function
```
The main difference between this definition of Sub **Average_of_3** and the Sub procedure defined previously is that, rather than assign the value of \( \frac{x + y + z}{3} \) to a class variable with an assignment statement, the value of the expression is "returned" to the calling program with a **Return** statement. The value of the expression is "assigned" to the name of the subprogram, in this case **Average_of_3**.

The way in which a Function procedure is called is different from the way a Sub Procedure is called. It is expressed the same as if it were a built-in function that was called—in an expression where the function to be called is named and a set of arguments to be used by the function is provided.

In the previous example, the calling subprogram, Button1_Click, can be revised as follows:

```vbnet
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs)
    Dim a As Double, b As Double, c As Double
    a = TextBox1.Text
    b = TextBox2.Text
    c = TextBox3.Text
    TextBox4.Text = Average_of_3(a, b, c)
End Sub
```

If you examine the syntax of the Function procedure definition, you will note the following features that distinguish it from the Sub procedure solution:

1. The declaration statement begins with the word "Function" rather than "Sub."
2. There is data type information following the list of parameters that were enclosed in parentheses. This additional information indicates what data type will be used to represent the value that will be returned by the subprogram.
3. There is a "return statement" whose syntax is: **Return** <expression>. In the example, the <expression> is given by: \( \frac{x + y + z}{3} \). The value of <expression> is the value that will be returned by the Function.
4. The Function procedure definition ends with an **End Function** statement rather than an **End Sub** statement.

Note how the Function procedure solution has eliminated the need for a shared variable in order to supply the result. We can now state another programming convention:

When writing a subprogram that returns a single value, always use a Function procedure.
Call By Value

In the examples of subprograms provided so far, you may have noticed in the parameter lists of the built-in methods for the control objects that the keyword `ByVal` appears at the beginning of the declaration of each formal parameter. If you forget to provide it when you type the parameter list of your own Function (or Sub) definition, the VS IDE inserts it for you. As one example, when entered into the VS IDE, the declaration statement for the Function `Average_of_3` that was given earlier, will actually be expressed as:

```vbnet
Function Average_of_3(ByVal x As Double, _
                      ByVal y As Double, _
                      ByVal z As Double) As Double
```

The question is, "What is the significance of this keyword?" Its purpose is related to how formal parameters share argument values between subprograms.

When a value is passed to a subprogram, it is most often the case that the original value outside the subprogram should be preserved and not subject to change. Consider what would happen if the compiler did not provide this protection. The following program illustrates the problem:

```vbnet
Public Class Form1
    Dim pi As Double

    Sub Routine1( )
        pi = 3.14159
        Call Routine2(pi)
    End Sub

    Sub Routine2(n As Double)
        n = 0.
    End Sub
End Class
```

This program appears to change the value of pi to the number 0! This is not an acceptable situation. Therefore, to avoid such a possibility a particular strategy for passing values to a subprogram is used called "Call by Value." This technique makes a copy of the value of the argument and passes the copy to the subprogram. This means
that the subprogram cannot, either deliberately or accidentally, change the original value of the argument provided in the calling program.

However, there are times when the programmer wants or needs to be able to change the value of a formal parameter in a subprogram and to have that change occur in the calling program as well. For example, yet another way to consider writing the `Average_of_3` subprogram is as follows:

```vbnet
Sub Average_of_3(x As Double, y As Double, z As Double, _
    avg As Double)
    avg = (x + y + z) / 3.
End Sub
```

In other words, we can introduce a fourth formal parameter, in this case `avg`, whose purpose is to return the value calculated by the subprogram. The calling program might then be written as:

```vbnet
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs)
    Dim a As Double, b As Double, c As Double, avg As Double
    a = TextBox1.Text
    b = TextBox2.Text
    c = TextBox3.Text
    Call Average_of_3(a, b, c, avg)
    TextBox4.Text = avg
End Sub
```

This program will compile successfully and will execute as well without "complaint." However, when you examine the output in `TextBox4`, you should observe that the average value displayed is always "0", no matter what values you supply in the `TextBoxes` that define `a`, `b`, and `c`. The reason is that the VS IDE inserted the keyword "ByVal" for each formal parameter and thus is employing the "Call by value" strategy. So, for all formal parameters, including `avg`, the subprogram is unable to change the value of the variable `avg` in the calling program, `Button1_Click`.

The variable `avg` occurs in two places: the Sub procedure `Average_of_3` and the Sub procedure `Button1_Click`. You might think therefore that `avg` is the same variable in both places. However, that is not so because the variable `avg` is declared as a formal parameter in `Average_of_3` and independently in `Button1_Click` in a `Dim` statement. Therefore, in neither case is `avg` defined as a class-level variable and this means that
each version of \texttt{avg} is only accessible within the subprogram in which it is defined. The names may be the same, but they are different variables.

**Call By Reference**

So far, the only way to return a value to a calling program successfully is by defining the subprogram as a Function procedure and supplying the value to be returned in a Return statement. This is generally the preferred way, particularly when a single value is to be returned.

There is, however, a way to implement \texttt{Average_of_3} as a Sub procedure provided the programmer overrides the "Call by Value" requirement when the subprogram is written. To do so, you must indicate that a formal parameter will not use the "Call by Value" method for passing values by providing another option, called "Call by Reference."

With this option, the calling program does not pass a value to a subprogram parameter but instead passes the memory address where the value is stored. In Unit 4 when you were first introduced to variables, it was explained that a variable is simply a name for a memory location where the value will be stored. So rather than pass a copy of the value stored at a memory location, the actual memory location, called an "address," is passed to the subprogram. The "value" assigned to the corresponding formal parameter is this address.

Knowing the memory location of a variable in the calling program allows the subprogram to change the value stored in that location. So, in the previous example, if you indicate that the formal parameter \texttt{avg} in the declaration of \texttt{Average_of_3} is to use "Call by Reference" rather than "Call by Value," the program will then work as expected without requiring any other changes to the program. You do so by using the keyword \texttt{ByRef} rather than \texttt{ByVal} when defining the formal parameters in the declaration statement for \texttt{Average_of_3}. That is, the new declaration statement is:

```
Sub Average_of_3(ByVal x As Double, _
                 ByVal y As Double, _
                 ByVal z As Double, _
                 ByRef avg As Double)
```

will ensure that the value of the variable \texttt{avg} in the calling program can be changed by the Sub procedure \texttt{Average_of_3} and that the new value will be accessible to the calling program, \texttt{Button1_Click}.

To be sure you understand the difference, you should enter the last solution given for the \texttt{Average_of_3} example without declaring that the formal parameter \texttt{avg} is Call by Reference and run the program with a simple input/output interface.
Then make only one change—add **ByRef** to the declaration of `avg`—then rerun the program and observe the result.

This example is provided primarily to illustrate that it is possible in Visual Basic to use a Sub procedure to return a value. However, as mentioned before, this is not considered good programming practice. The accepted convention is: When writing a subprogram that returns a single value, always use a Function procedure.

So when is it appropriate to use "Call by Reference" to return values? The answer is "When a subprogram needs to return more than one value!" In a later section of this Study Guide, you will be introduced to some "structured data types." These data types provide programmers with the ability to work with collections of values. For now, it is sufficient to recognize that such data types involve more than one value and therefore cannot be returned as the value of a Function procedure since only single values can be returned from this type of general procedure. However, there are situations where multiple values may need to be returned, even when structured data types are not being used.

Suppose for example that you wish to write a subprogram that obtains the x-co-ordinate and the y-co-ordinate of a point in the Cartesian plane supplied by the user through a windows interface. This subprogram is to be called by other subprograms whenever the co-ordinates of a point are required.

Since there are two values for the subprogram to return, you cannot use a Function procedure, so the subprogram will be implemented as a Sub procedure. Therefore, the values will have to be returned through the parameter list, and "Call by Reference" must be used for these parameters:

```vbnet
Sub Coords(ByRef x As Double, ByRef y As Double)
    x = TextBox1.Text
    y = TextBox2.Text
End Sub
```

**An Example Using Functions**

The following example calculates the distance between two points in the Cartesian plane and displays the result when the button is clicked. First, it must retrieve two points from the user, who must define the x-coordinate and the y-coordinate in `TextBox1` and `TextBox2` respectively for each point. The method then calculates the distance between the two points. A hierarchy chart for the program is as follows:
A proposed Form for entering the points and displaying the distance is:

Function procedures will be used to extract the x and y coordinates for each point as well as to compute the distance since all these activities return a single value. The source code is given on the next page:
Public Class Form1

' CLEAR Button initializes TextBox1 and TextBox and gives TextBox1 the focus
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    TextBox1.Text = "(0, 0)"
    TextBox2.Text = "(0, 0)"
End Sub

'CALC Button computes the distance from (x0, y0) to (x1, y1)
Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
    TextBox3.Text = Distance(TextBox1.Text, TextBox2.Text)
End Sub

'QUIT Button terminates execution
Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
    Me.Close()
End Sub

Function Distance(coord1 As String, coord2 As String) As Double
    Dim x0, y0, x1, y1 As Double
    x0 = X_Coord(coord1)
    y0 = Y_Coord(coord1)
    x1 = X_Coord(coord2)
    y1 = Y_Coord(coord2)
    Return ((x1 - x0) ^ 2 + (y1 - y0) ^ 2) ^ 0.5
End Function

Function X_Coord(coords As String) As Double
    Dim comma_posn As Integer, left_paren_posn As Integer
    comma_posn = coords.IndexOf(""")
    left_paren_posn = coords.IndexOf("()")
    Return CDbl(coords.Substring(left_paren_posn + 1, comma_posn - left_paren_posn - 1))
End Function

Function Y_Coord(coords As String) As Double
    Dim comma_posn As Integer, right_paren_posn As Integer
    comma_posn = coords.IndexOf(""")
    right_paren_posn = coords.IndexOf("()")
    Return CDbl(coords.Substring(comma_posn + 1, right_paren_posn - comma_posn - 1))
End Function

Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
    Button1_Click(sender, e) 'Initializes the Textboxes to "(0, 0)"
End Sub

End Class