Using Function Procedure Arguments

Keep in mind the following about function procedure arguments:

- Arguments can be variables (including arrays), constants, literals, or expressions.
- Some functions do not have arguments.
- Some functions have a fixed number of required arguments (from 1 to 60).
- Some functions have a combination of required and optional arguments.

Creating a function with no arguments

Most functions use arguments, but that's not a requirement. Excel, for example, has a few built-in worksheet functions that don't use arguments, such as RAND, TODAY, and NOW.

The following is a simple example of a function that has no arguments. This function returns the UserName property of the Application object, which is the name that appears in the Personalize section of the Excel Options dialog box. This function is simple, but it can be useful because there's no built-in function that returns the user's name:

```
Function User()
     ' Returns the name of the current user
     User = Application.UserName
End Function
```

When you enter the following formula into a worksheet cell, the cell displays the name of the current user:

```
=User()
```

As with Excel's built-in functions, when you use a function with no arguments, you must include a set of empty parentheses.

Creating a function with one argument

The function that follows takes a single argument and uses the Excel text-to-speech generator to "speak" the argument:

```
Function SayIt(txt)
     Application.Speech.Speak (txt)
End Function
```

For example, if you enter this formula, Excel will "speak" the contents of cell A1 whenever the worksheet is recalculated:

```
=SayIt(A1)
```

You can use this function in a slightly more complex formula, as shown here. In this example, the argument is a text string rather than a cell reference:

```
=IF(SUM(A:A)>1000,SayIt("Goal reached"),)
```

This formula calculates the sum of the values in Column A. If that sum exceeds 1,000, you will hear "Goal reached." When you use the SayIt function in a worksheet formula, the function always returns 0 because a value is not assigned to the function's name.

Creating another function with one argument

This section contains a more complex function that is designed for a sales manager who needs to calculate the commissions earned by the sales force. The commission rate is based on the amount sold—those who sell more earn a higher commission rate. The function returns the commission amount, based on the sales made (which is the function's only argument—a required argument). The calculations in this example are based on the following table:

Monthly Sales	Commission Rate
0–$9,999	8.0%
$10,000–$19,999	10.5%
$20,000–$39,999	12.0%
$40,000+	14.0%

You can use any of several different methods to calculate commissions for various sales amounts that are entered into a worksheet. You could write a formula such as the following:

```
=IF(AND(A1>=0,A1<=9999.99),A1*0.08,IF(AND(A1>=10000,
A1<=19999.99), A1*0.105,IF(AND(A1>=20000,
A1<=39999.99),A1*0.12,IF(A1>=40000,A1*0.14,0))))
```

This approach isn't the best for a couple of reasons. First, the formula is overly complex and difficult to understand. Second, the values are hard-coded into the formula, making the formula difficult to modify if the commission structure changes. A better solution is to use a lookup table function to compute the commissions; here's an example:

```
=VLOOKUP(A1,Table,2)*A1
```

Using the VLOOKUP function requires that you have a table of commission rates set up in your worksheet. Another option is to create a custom function, such as the following:

```
Function Commission(Sales)
     ' Calculates sales commissions
     Tier1 = 0.08
     Tier2 = 0.105
     Tier3 = 0.12
     Tier4 = 0.14
          Select Case Sales
          Case 0 To 9999.99
          Commission = Sales * Tier1
          Case 10000 To 19999.99
          Commission = Sales * Tier2
          Case 20000 To 39999.99
          Commission = Sales * Tier3
          Case Is >= 40000
          Commission = Sales * Tier4
     End Select
End Function
```

After you define the Commission function in a VBA module, you can use it in a worksheet formula. Entering the following formula into a cell produces a result of 3,000. (The amount, 25,000, qualifies for a commission rate of 12%.)

```
=Commission(25000)
```

If the sales amount is in cell D23, the function's argument would be a cell reference, like this:

```
=Commission(D23)
```

Creating a function with two arguments

This example builds on the previous one. Imagine that the sales manager implements a new policy: the total commission paid is increased by 1 percent for every year that the salesperson has been with the company. For this example, the custom Commission function (defined in the preceding section) has been modified so that it takes two arguments, both of which are required arguments. Call this new function Commission2:

```
Function Commission2(Sales, Years)
     ' Calculates sales commissions based on years in
     service
          Tier1 = 0.08
          Tier2 = 0.105
          Tier3 = 0.12
          Tier4 = 0.14
     Select Case Sales
     Case 0 To 9999.99
     Commission2 = Sales * Tier1
     Case 10000 To 19999.99
     Commission2 = Sales * Tier2
     Case 20000 To 39999.99
     Commission2 = Sales * Tier3
     Case Is >= 40000
     Commission2 = Sales * Tier4
     End Select
     Commission2 = Commission2 + (Commission2 * Years /
100)
End Function
```

The modification was quite simple. The second argument (Years) was added to the Function statement, and an additional computation was included that adjusts the commission before exiting the function. The following is an example of how you write a formula using this function. It assumes that the sales amount is in cell A1 and that the number of years that the salesperson has worked is in cell B1:

```
=Commission2(A1,B1)
```

Creating a function with a range argument

The example in this section demonstrates how to use a worksheet range as an argument. Actually, it's not at all tricky; Excel takes care of the details behind the scenes. Assume that you want to calculate the average of the five largest values in a range named Data. Excel doesn't have a function that can do this calculation, so you can write the following formula:

```
=(LARGE(Data,1)+LARGE(Data,2)+LARGE(Data,3)+
```

```
LARGE(Data,4)+LARGE(Data,5))/5
```

This formula uses Excel's LARGE function, which returns the nth largest value in a range. The preceding formula adds the five largest values in the range named Data and then divides the result by 5. The formula works fine, but it's rather unwieldy. Plus, what if you need to compute the average of the top six values? You'd need to rewrite the formula and make sure that all copies of the formula also get updated. Wouldn't it be easier if Excel had a function named TopAvg? For example, you could use the following (nonexistent) function to compute the average:

```
=TopAvg (Data,5)
```

This situation is an example of when a custom function can make things much easier for you. The following is a custom VBA function, named TopAvg, which returns the average of the top n values in a range:

```
Function TopAvg(Data, Num)
' Returns the average of the highest Num values in
Data
     Sum = 0
     For i = 1 To Num
     Sum = Sum + WorksheetFunction.Large(Data, i)
     Next i
     TopAvg = Sum / Num
End Function
```

This function takes two arguments: Data (which represents a range in a worksheet) and Num (the number of values to average). The code starts by initializing the Sum variable to 0. It then uses a For-Next loop to calculate the sum of the nth largest values in the range. (Note that Excel's LARGE function is used within the loop.) You can use an Excel worksheet function in VBA if you precede the function with WorksheetFunction and a period. Finally, TopAvg is assigned the value of Sum divided by Num.

You can use all Excel worksheet functions in your VBA procedures except those that have equivalents in VBA. For example, VBA has a Rnd function that returns a random number. Therefore, you can't use Excel's RAND function in a VBA procedure.

Creating a simple but useful function

Useful functions don't have to be complicated. The function in this section is essentially a wrapper for a built-in VBA function called Split. The Split function makes it easy to extract an element in a delimited string. The function is named ExtractElement:

```
Function ExtractElement(Txt, n, Separator)
     ' Returns the nth element of a text string, where the
     ' elements are separated by a specified separator
     character
     ExtractElement = Split(Application.Trim(Txt),
     Separator)(n - 1)
End Function
```

The function takes three arguments:

Txt: A delimited text string, or a reference to a cell that contains a delimited text string

n: The element number within the string

Separator: A single character that represents the separator

Here's a formula that uses the ExtractElement function:

```
=EXTRACTELEMENT("123-45-678",2,"-")
```

The formula returns 45, the second element in the string that's delimited by hyphens. The delimiter can also be a space character. Here's a formula that extracts the first name from the name in cell A1:

```
=EXTRACTELEMENT(A1,1," ")
```

Debugging Custom Functions

Debugging a function can be a bit more challenging than debugging a Sub procedure. If you develop a function to use in worksheet formulas, an error in the function simply results in an error display in the formula cell (usually #VALUE!). In other words, you don't receive the normal runtime error message that helps you locate the offending statement.

When you're debugging a worksheet formula, using only one instance of the function in your worksheet is the best technique. The following are three methods you may want to use in your debugging:

- Place MsgBox functions at strategic locations to monitor the value of specific variables. Fortunately, message boxes in function procedures pop up when the procedure is executed. But make sure you have only one formula in the worksheet that uses your function; otherwise, the message boxes appear for each formula that's evaluated.
- Test the procedure by calling it from a Sub procedure. Runtime errors display normally, and you can either fix the problem (if you know what it is) or jump right into the debugger.
- Set a breakpoint in the function, and then use the Excel debugger to step through the function. Press F9, and the statement at the cursor becomes a breakpoint. The code will stop executing, and you can step through the code line by line (by pressing F8). Consult the Help system for more information about using VBA debugging tools.

UserForm Example

The example in this section is an enhanced version of the ChangeCase procedure presented at the beginning of the chapter. Recall that the original version of this macro changes the text in the selected cells to uppercase characters. This modified version asks the user what type of case change to make: uppercase, lowercase, or proper case (initial capitals).

Creating the UserForm

This UserForm needs one piece of information from the user: the type of change to make to the text. Because only one option can be selected, OptionButton controls are appropriate. Start with an empty workbook and follow these steps to create the UserForm:

1. Press Alt+F11 to activate the VBE.
2. In the VBE, choose Insert ⇨ UserForm. The VB Editor adds an empty form named UserForm1 and displays the Toolbox.
3. Press F4 to display the Properties window and then change the following properties of the UserForm object:

Property	Change to
Name	UChangeCase
Caption	Change Case

4. Add a CommandButton object to the UserForm and then change the following properties for the CommandButton:

Property	Change to
Name	OKButton
Caption	OK
Default	True

5. Add another CommandButton object and then change the following properties:

Property	Change to
Name	CancelButton
Caption	Cancel
Cancel	True

6. Add an OptionButton control and then change the following properties. (This option is the default, so its Value property should be set to True.)

Property	Change to
Name	OptionUpper
Caption	Upper Case
Value	True

7. Add a second OptionButton control and then change the following properties:

Property	Change to
Name	OptionLower
Caption	Lower Case

8. Add a third OptionButton control and then change the following properties:

Property	Change to
Name	OptionProper
Caption	Proper Case

9. Adjust the size and position of the controls and the form until your UserForm resembles the one shown in Figure 44.10. Make sure that the controls do not overlap.

Creating event handler procedures

The next step is to create two event handler procedures: one to handle the Click event for the CancelButton CommandButton and the other to handle the Click event for the OKButton CommandButton. Event handlers for the OptionButton controls are not necessary. The VBA code can determine which of the three OptionButton controls is selected, but it does not need to react when the choice is changed—only when OK or Cancel is clicked.

Event handler procedures are stored in the UserForm code module. To create the procedure to handle the Click event for the CancelButton, follow these steps:

1. Activate the UserForm1 form by double-clicking its name in the Project window.
2. Double-click the CancelButton control. The VBE activates the code module for the UserForm and inserts an empty procedure.
3. Insert the following statement before the End Sub statement: Unload Me

That's all there is to it. The following is a listing of the entire procedure that's attached to the Click event for the CancelButton:

```
Private Sub CancelButton_Click()
     Unload Me
End Sub
```

This procedure is executed when the CancelButton is clicked. It consists of a single statement that unloads the form. Next, add the code to handle the Click event for the OKButton control. Follow these steps:

1. Select OKButton from the drop-down list at the top of the module or reactivate the UserForm and double-click the OKButton control. The VBE creates a new procedure called OKButton_Click.
2. Enter the following code. The VBE has already entered the first and last statements for you:

```
Private Sub OKButton_Click()
     ' Exit if a range is not selected
     If TypeName(Selection) <> "Range" Then Exit Sub
     ' Upper case
     If Me.OptionUpper.Value Then
     For Each cell In Selection
     If Not cell.HasFormula Then
          cell.Value = StrConv(cell.Value, vbUpperCase)
     End If
     Next cell
     End If
     ' Lower case
     If Me.OptionLower.Value Then
     For Each cell In Selection
     If Not cell.HasFormula Then
          cell.Value = StrConv(cell.Value, vbLowerCase)
     End If
     Next cell
     End If
     ' Proper case
     If Me.OptionProper.Value Then
     For Each cell In Selection
     If Not cell.HasFormula Then
          cell.Value = StrConv(cell.Value, vbProperCase)
     End If
     Next cell
     End If
     Unload Me
End Sub
```

The macro starts by checking the type of selection. If a range is not selected, the procedure ends. The remainder of the procedure consists of three separate blocks. Only one block is executed, determined by which OptionButton is selected. The selected OptionButton has a Value of True. Finally, the UserForm is unloaded (dismissed).

Showing the UserForm

At this point, the UserForm has all of the necessary controls and event procedures. All that's left is a way to display the form. This section explains how to write a VBA procedure to display the UserForm:

1. Make sure the VBE window is activated.
2. Insert a module by choosing Insert ➪ Module.
3. In the empty module, enter the following code:

```
Sub ShowUserForm()
     UChangeCase.Show
End Sub
```

4. Choose Run ➪ Run Sub/UserForm (or press F5). The Excel window is activated, and the new UserForm is displayed.

Testing the UserForm

To try the UserForm from Excel, follow these steps:

1. Activate Excel.
2. Enter some text into a range of cells.
3. Select the range with the text.
4. Choose Developer ➪ Code ➪ Macros (or press Alt+F8). The Macro dialog box appears.
5. Select ShowUserForm from the list of macros and then click Run. The UserForm appears.
6. Make your choice, and click OK.

Try it with a few more selections, including noncontiguous cells. Notice that if you click Cancel, the UserForm is dismissed, and no changes are made. The code does have a problem, though: if you select one or more entire columns, the procedure processes every cell, which can take a long time. The version of the workbook on the website corrects this problem by working with a subset of the selection that intersects with the workbook's used range.

Enhancing UserForms

Creating UserForms can make your macros much more versatile. You can create custom commands that display dialog boxes that look exactly like those that Excel uses. This section contains some additional information to help you develop custom dialog boxes that work like those that are built in to Excel.

Adding accelerator keys

All Excel dialog boxes work well with a mouse and a keyboard because each control has an associated accelerator key. The user can press Alt plus the accelerator key to work with a specific dialog box control.

Your custom dialog boxes should also have accelerator keys for all controls. You add accelerator keys in the Properties window by entering a character for the Accelerator property.

Your accelerator key can be any letter, number, or punctuation, regardless of whether that character appears in the control's caption. It's a good practice to use a letter that is in the control's caption, though, because that letter will be underlined—a visual cue for the user.

Another common convention is to use the first letter of the control's caption. But don't duplicate accelerator keys. If the first letter is already taken, use a different letter, preferably

one that is easy to associate with the word (like a hard consonant). If you have duplicate accelerator keys, the accelerator key acts on the next control in the tab order of the UserForm. Then, pressing the accelerator key again takes you to the second control with that accelerator. Some controls (such as textboxes) don't have a Caption property and other controls (such as labels) can't have the focus. You can assign an accelerator key to a label that describes the control and put that label right before your target control in the tab order. Pressing the accelerator key for a control that can't take the focus activates the next control in the tab order.

Controlling tab order

The previous section refers to a UserForm's tab order. When you're working with a UserForm, pressing Tab and Shift+Tab cycles through the dialog box's controls. When you create a UserForm, you should make sure that the tab order is correct. Usually, it means that tabbing should move through the controls in a logical sequence.

To view or change the tab order in a UserForm, choose View ➪ Tab Order to display the Tab Order dialog box. You can then select a control from the list; use the Move Up and Move Down buttons to change the tab order for the selected control.