

---

## VII VISUAL BASIC FOR APPLICATIONS (VBA)

Visual Basic for Applications (VBA) is the programming language attached to Excel. VBA is very functional and flexible. Because of its ready integration with Excel worksheets, VBA is widely used in the financial community. VBA incorporates many features that are part of standard programming languages, and it is not difficult to master if you have some programming experience.

You do not need to be proficient in VBA to understand Sections I–VI of *Financial Modeling*. These sections can be understood without anything more than the very rudimentary VBA principles incorporated in the preface to this book (or alternatively in the small file called “Adding Getformula to your Spreadsheet” that is part of the disk that comes with the book).

The four chapters of this section cover Visual Basic for Applications (VBA) topics for the reader interested in developing his or her own programs. Chapter 36 shows how to write functions that can be added in to Excel spreadsheets. *Financial Modeling* uses many of these “homemade” functions. Examples are the two-stage Gordon model (Chapter 3), Black-Scholes pricing of options (Chapter 17), and derivation of the Nelson-Siegel term structure (Chapter 22).

Chapter 37 discusses more advanced topics related to variables and arrays in VBA. We have used this topic in fixing the bugs in Excel’s **XNPV** and **XIRR** functions (Chapter 1). Chapter 38 shows how to build subroutines in VBA. A subroutine is not a function, but rather an automation of some repetitive action. *Financial Modeling* uses subroutines in a number of places—for example, in computing the efficient frontier without short sales (Chapter 12).

Finally, Chapter 39 discusses objects and add-ins. Among other topics discussed in this chapter is the creation of user-defined add-ins in Excel.



---

# 36

## User-Defined Functions with VBA

---

### 36.1 Overview

Chapters 36–39 discuss the uses of Excel’s programming language, Visual Basic for Applications (VBA). VBA provides a complete programming language and environment fully integrated with Excel and all other Microsoft Office applications. In this chapter we introduce user-defined functions, which are used in various places in this book.

The examples and screen shots depict the Excel 2013 working environment but are fully compatible (unless otherwise noted) with all versions of Excel using Visual Basic for Applications (Version 5 and above).

---

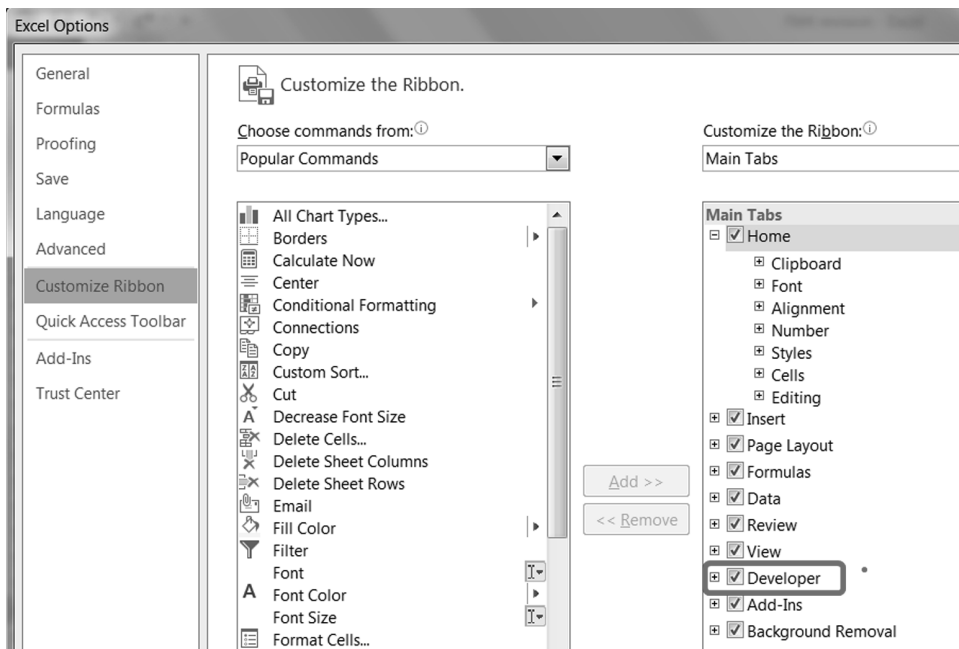
### 36.2 Using the VBA Editor to Build a User-Defined Function

Throughout this book we have used VBA to define functions that are not included in Excel. One example is the function **Getformula** that is attached to all the spreadsheets in this book; another example is the function that computes the Black-Scholes option value (Chapter 17). In this section we show you how to build a user-defined function. A user function is a saved list of instructions for Excel that produces a value. Once defined, a user function can be used inside an Excel worksheet like any other function.<sup>1</sup>

---

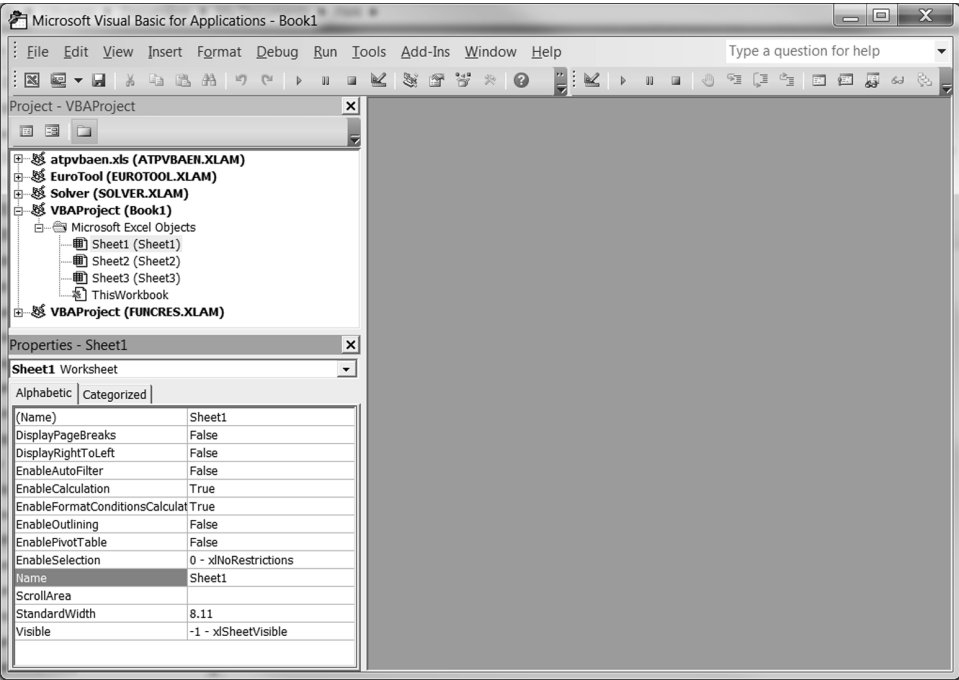
1. User-defined functions are usually attached to a specific workbook and are only available if that workbook is currently open in Excel. If you want a macro to be available whenever you use Excel on a specific computer save it in the **Personal Macro Workbook**; see Chapter 35, section 16. Another way of having access to a VBA function across worksheets is to put it in an add-in; see Chapter 39 for an introduction to add-ins in Excel.

In this section we will write our first user-defined function. Before you can do this, you need to activate the VBA editor. You can do this either by using the keyboard shortcut [Alt] + F11 or from the Excel ribbon (**Developer Tab**|**Visual Basic Editor**). By default, Excel doesn't display the **Developer** tab on the Excel ribbon. To show the **Developer** tab, go to **File**|**Options**|**Customize the Ribbon** and indicate **Developer**:

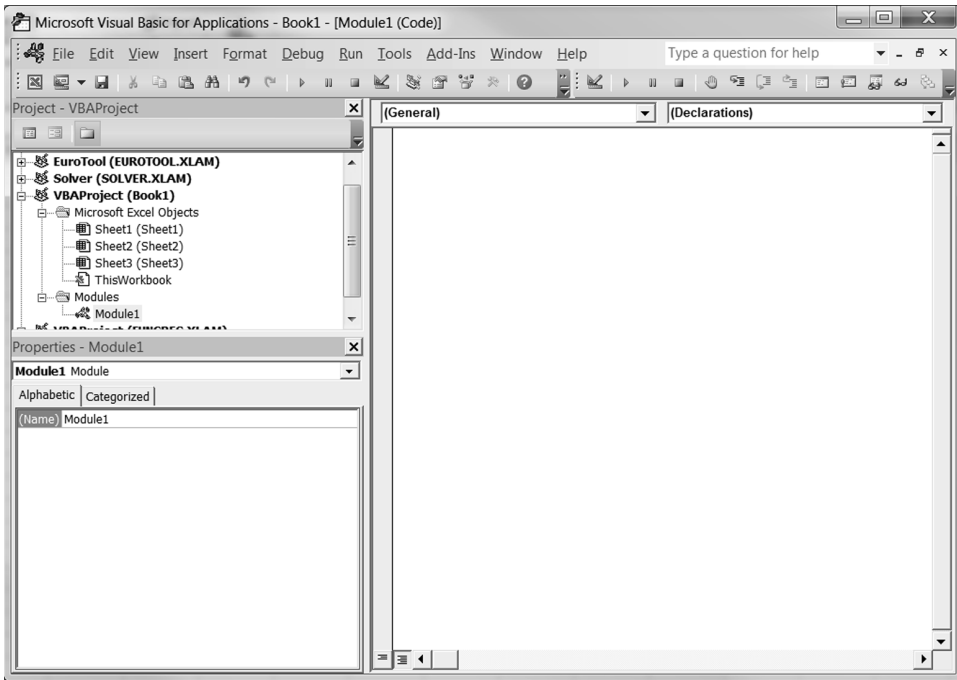




The result in both cases is a new window like the following screen shot (your window may look slightly different, but it will be functionally equivalent).



A user-defined function needs to be written in a module. To open a new module, select **Insert|Module** from the menu in the VBA editor environment. This will open a new window, as illustrated in the next screen shot:



We are now ready to write our first function. This function (named “**plus**”) will add together two numbers.

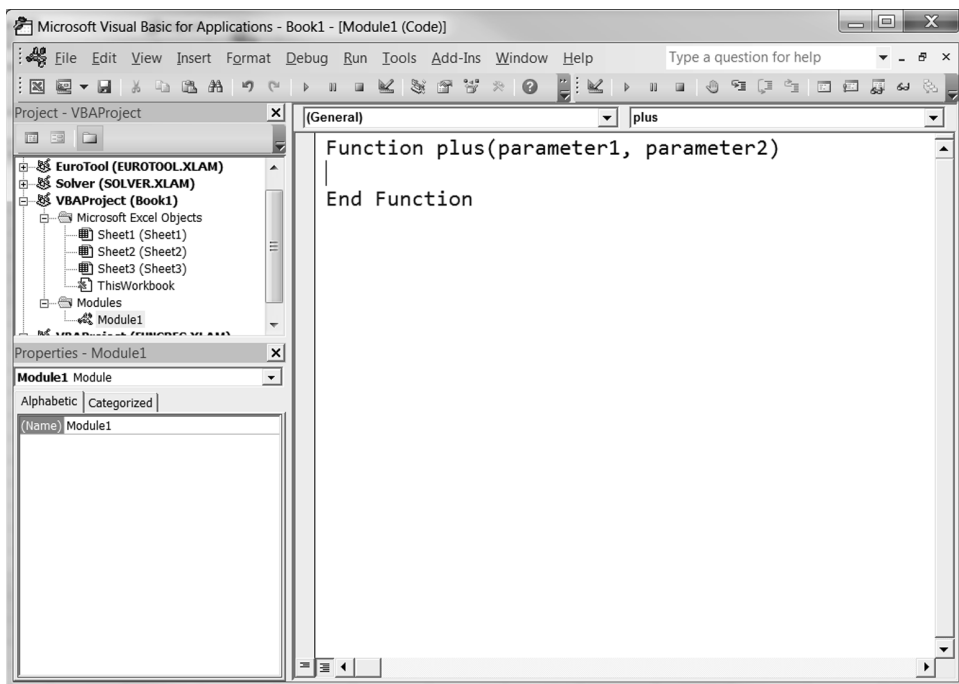
A user-defined function in Excel has three obligatory elements:

1. A header line with the name of the function and a list of parameters.
2. A closing line (usually inserted by VBA).
3. Some program lines between the header and the closing line.

Start writing the first line of the function:

```
function plus (parameter1,parameter2)
```

As soon as you end the line with a tap on the Enter key, VBA will do a cleanup job. The color of all the words that VBA recognizes as part of its programming language (“reserved words”) will change. All reserved words will be capitalized. A space will be added after the comma separating the first parameter from the second parameter. The closing line for the function will be inserted, and the cursor will be in position between the header and the closing line ready for you to go on typing.



We are now ready to type our function line. This is the line that makes our function do something.<sup>2</sup> Our first function will take two variables and return their sum:

```
Function plus(parameter1, parameter2)
    plus = parameter1 + parameter2
End Function
```


You can now use this function in your spreadsheet:

	A	B	C
1	<b>PLUS IN ACTION</b>		
2	Parameter1	3.25	
3	Parameter2	1.5	
4	Plus	4.75	<-- =plus(B2,B3)

2. The indentation of lines in VBA code, which we added manually, is not required by VBA but makes reading the code much easier.

The fastest way to insert a function (assuming you know its name) is to start typing the name. When the suggested list of names narrows down, select the appropriate function name from the list:

	A	B	C
1	PLUS IN ACTION		
2	Parameter1	3.25	
3	Parameter2	1.5	
4	Plus	=p	
5		<div><div>PDURATION</div><div>PEARSON</div><div>PERCENTILE.EXC</div><div>PERCENTILE.INC</div><div>PERCENTRANK.EXC</div><div>PERCENTRANK.INC</div><div>PERMUT</div><div>PERMUTATIONA</div><div>PHI</div><div>PI</div><div>Plus</div><div>PMT</div></div>	
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			

You can also use the function in the Excel Function Wizard. Clicking on this  icon on the toolbar will produce the following screen:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

A

B

C

D

E

F

G

H

I

J

PLUS IN ACTION

Parameter13.25

Parameter21.5

Plus=

Insert Function

Search for a function:

Type a brief description of what you want to do and then click Go

Go

Or select a category:

Most Recently Used

Select a function:

T.TEST

PMT

SUM

AVERAGE

IF

HYPERLINK

COUNT

T.TEST(array1,array2,tails,type)

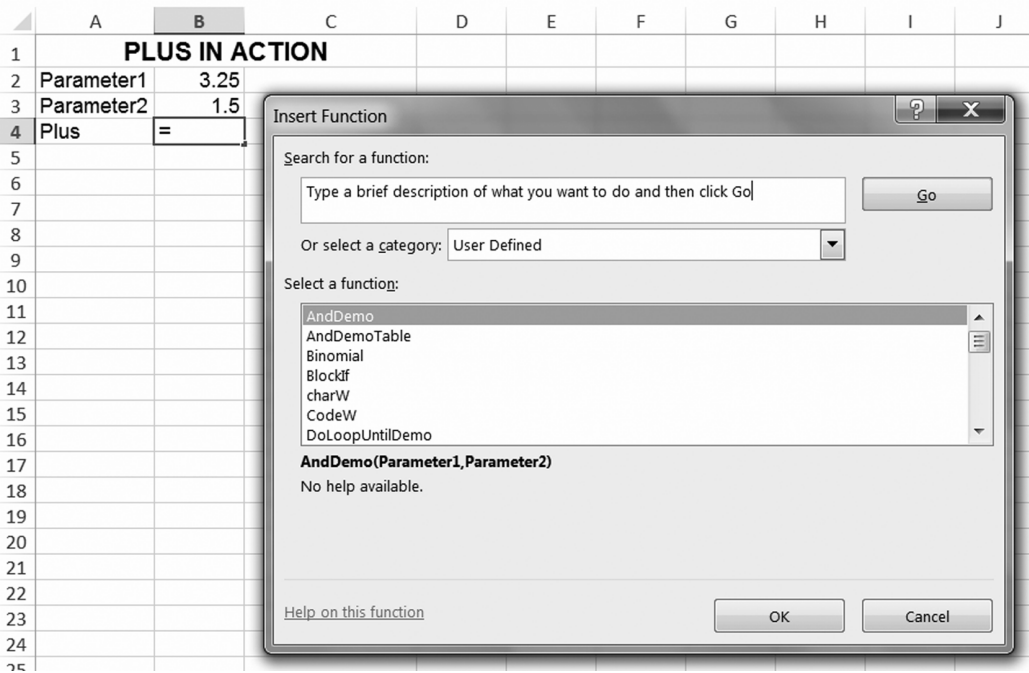
Returns the probability associated with a Student's t-Test.

Help on this function

OK

Cancel

Selecting **User Defined** from the pull-down menu will present the following screen listing all user-defined functions; one of them should be the function we have just added, **plus**:



When you select **plus** and click OK, you will see that Excel treats this like any other function, bringing up a dialogue box that asks for the location or value of **parameter1** and **parameter2**:

The image shows the 'Function Arguments' dialog box in Microsoft Excel. The title bar reads 'Function Arguments' with a question mark icon and a close button (X). The main area is titled 'plus'. It contains two input fields: 'Parameter1' and 'Parameter2'. Each field has a small grid icon to its right, followed by an equals sign (=). Below these fields, there is a text label 'No help available.' and the text 'Parameter1' centered. At the bottom left, it says 'Formula result =' followed by a link 'Help on this function'. At the bottom right, there are two buttons: 'OK' and 'Cancel'.

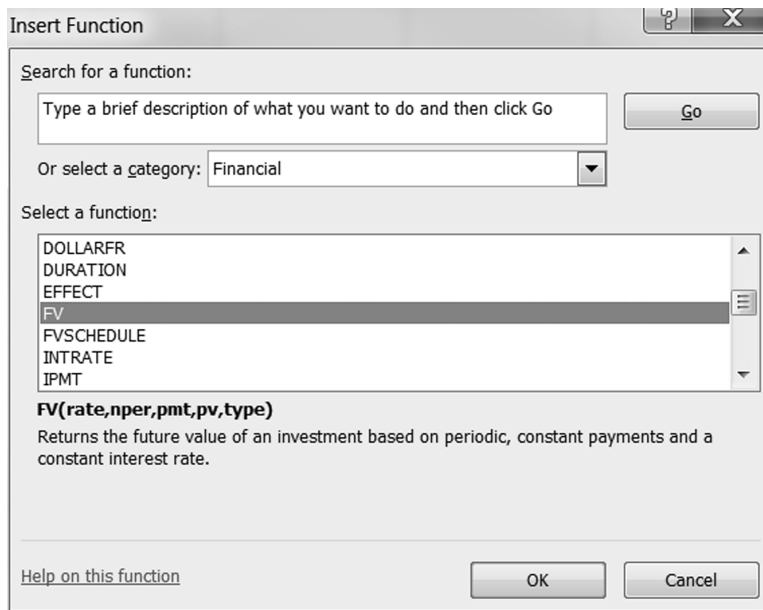
Notice that at this point there is no explanation or help for the function. The next section provides part of the remedy.



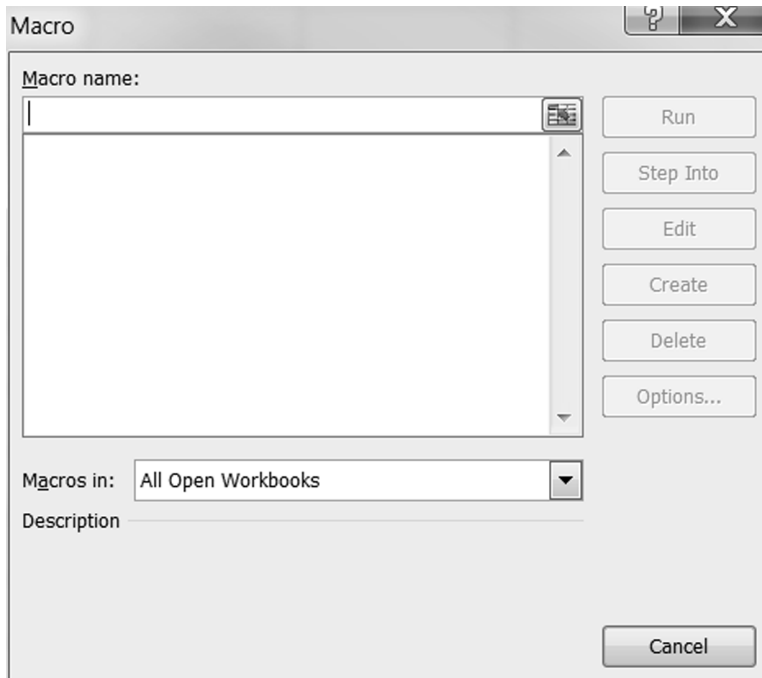
---

### 36.3 Providing Help for User-Defined Functions in the Function Wizard

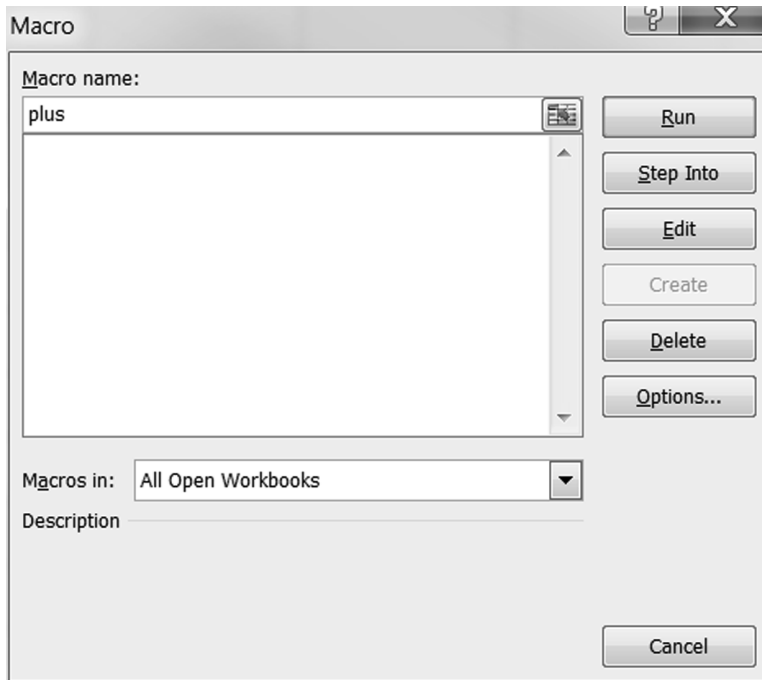
Excel's Function Wizard (shown below) provides a short help line (an explanation of what the function does). Here's how Excel explains its own functions in the Function Wizard:



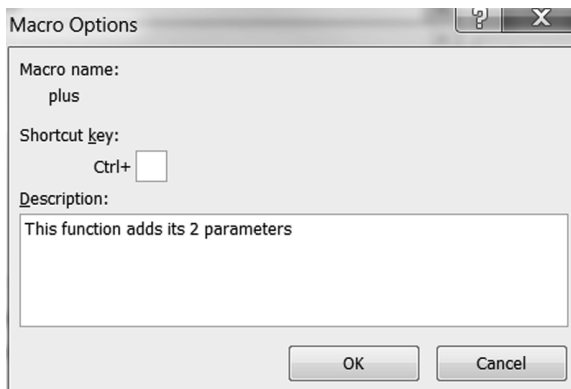
To attach a text description to our function, activate the macro selection box. You can do this either from the Excel ribbon (**Developer**|**Macros**) or by using the keyboard shortcut [Alt] + F8.



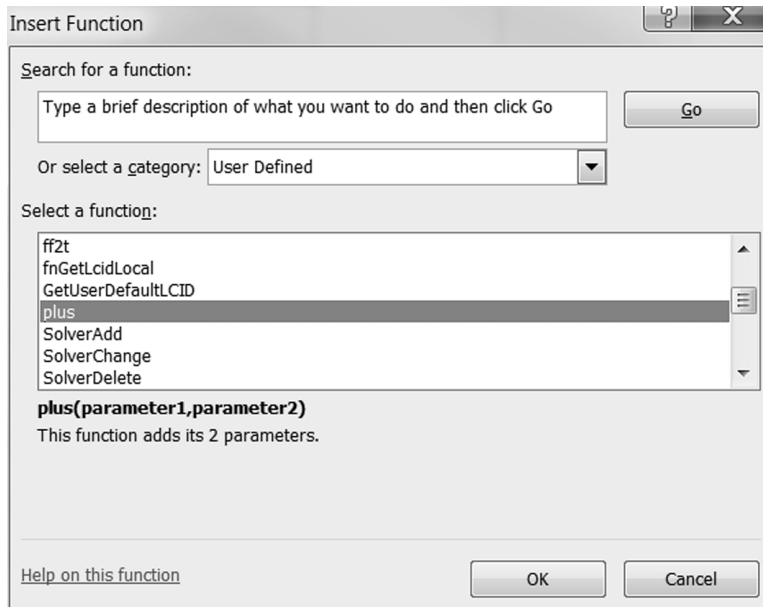
Click in the **Macro name** box, and type the name of the function (notice that you don't see the function name in the macro dialogue box above ... you have to type it in):



Click on the **Options** button:



Type the description in the **Description** box. Click **OK**, and close the macro selection box. Our function now has a help line.



Excel functions have help lines attached to each of the parameters and a help file entry. We can supply the same for our function; sadly, the subject is beyond the scope of this introduction.

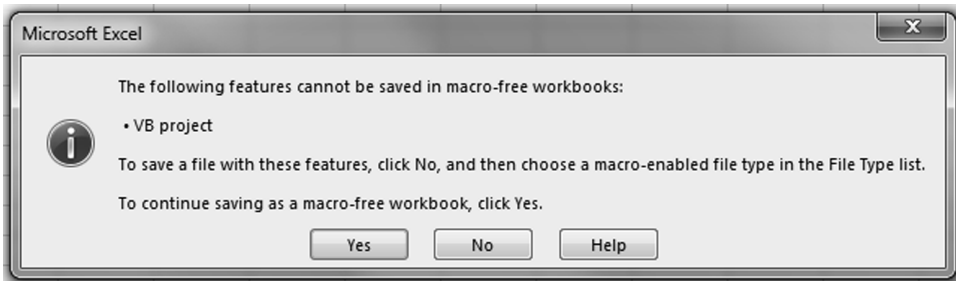
---

## 36.4 Saving Excel Workbook with VBA Content

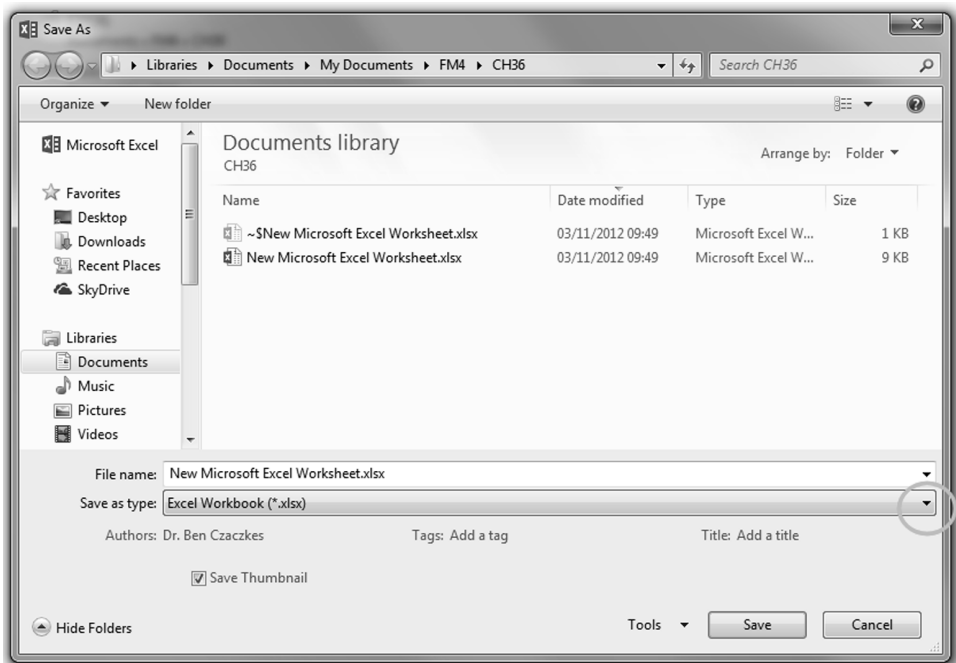
At some point in the process, you need to save your work.<sup>3</sup> Starting with Excel 2007, an Excel workbook with VBA content has to be saved as a “macro-enabled file.” When you first try to save a workbook with VBA content, Excel will present you with the following message:

---

3. We suggest soon and often.



You should choose **No** and get the **Save As** dialog to enable you to choose a new file type.



Now open the circled selection, select the second option, **xlsm**, and save the workbook. If you use VBA often, then you might consider changing the default Excel file type to **xlsm**.<sup>4</sup>

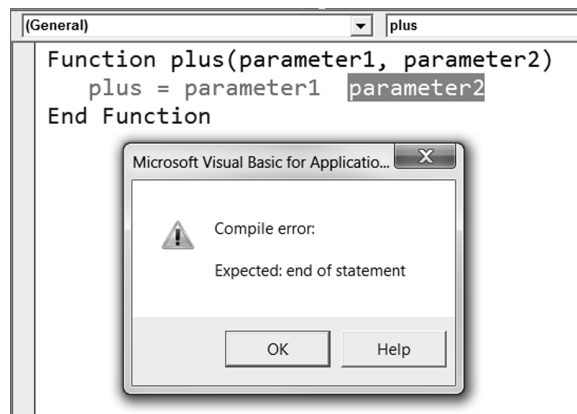
---

### 36.5 Fixing Mistakes in VBA

Once you start using VBA, you're sure to make mistakes. In this section we illustrate several typical mistakes and help you correct them. This list is not meant to be exhaustive—we have selected mistakes typically made by VBA beginners.

#### Mistake 1: Using the Wrong Syntax

Suppose that in writing **Plus** you forget the “+” between **parameter1** and **parameter2** (recall that the function is supposed to return **parameter1 + parameter2**). Once you hit the Enter key, you get the following error message:



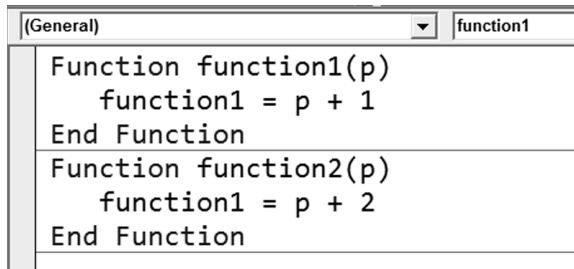
Clicking the OK button corrects this problem.

---

4. The command is **File|Options|Save|Save files in this format**.

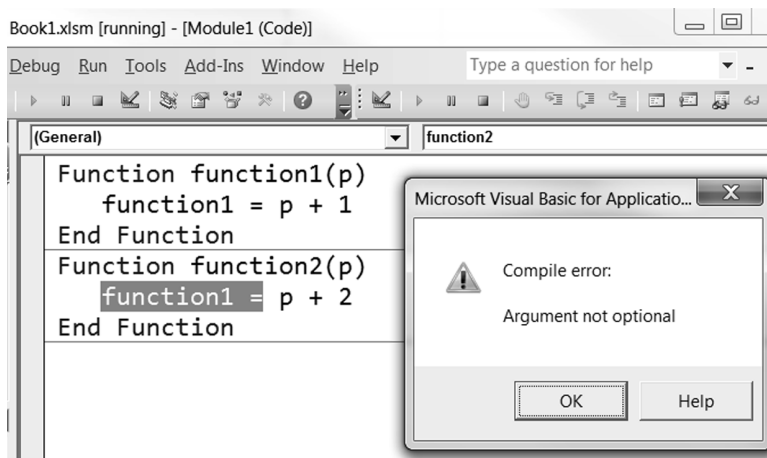
## Mistake 2: Right Syntax with a Typing Error

It's easy to make typing errors that will only be detected once you try to use the function. In the example below, we define two functions—**function1** and **function2**. Unfortunately, the program line for **function2** mistakenly calls the function “function1”:



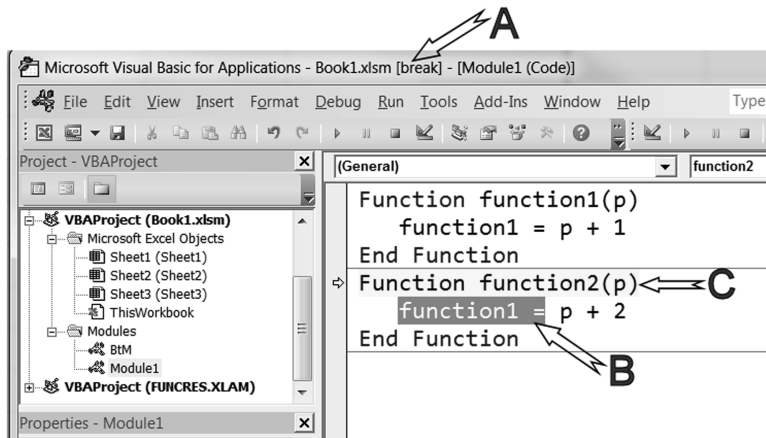
```
(General) function1
Function function1(p)
    function1 = p + 1
End Function
Function function2(p)
    function1 = p + 2
End Function
```

The VBA editor does not immediately recognize this mistake. The mistake will pop up when you try to use the function in a worksheet. Excel will notify you that you've made a mistake and take you to the VBA editor:




If you recognize your mistake, you can correct it. You can also try to go to the VBA help by clicking **Help** (in many cases this will lead to an incomprehensibly complicated explanation).

Suppose you recognize your mistake. You click **OK**, and get ready to correct the error by replacing the word “Function1” with “Function2.” At this point your screen looks like this:



Notice:

- A. The word [**break**] in the title bar.
- B. The offending symbol is selected.
- C. The function line is highlighted and pointed to by an arrow in the margin.

Because VBA found an error while trying to execute the function, it moved into a special execution mode called debug-break mode. For now all we need to do is get out of this special mode so we can get on with our work. We do this by clicking the  icon on the VBA toolbar. Now you can fix the function and use it.

We can (and should) have VBA check the module for errors before trying to use the functions in the module. From the VBA menu we select **Debug|Compile VBAproject**; this will find the first error in the module and point it out as before but without going into debug-break mode.



36.6 Conditional Execution: Using If Statements in VBA Functions

In this section we explore the **If** statements available to you in VBA. Not all things in life are linear, and sometimes decisions have to be made. **If** statements are one way of doing this in VBA

The One-Line If Statement

The one-line **If** statement is the simplest way to control the execution of a VBA function: One statement is executed if a condition is true and another is executed if a condition is not true. The complete condition and its statement should be on one line. Here’s an example:

```
Function OneLineIf(Parameter)
    If Parameter > 5 Then OneLineIf = 1
    Else OneLineIf = 15
End Function
```

We can now use the function **OneLineIf** in Excel. When **Parameter** is > 5, **OneLineIf** returns 1 and when **Parameter** is < 5, **OneLineIf** returns 15.

	A	B	C
1	ONELINEIF IN ACTION		
2	Parameter		
3	12	1	<-- =OneLineIf(A3)
4	3	15	<-- =OneLineIf(A4)

The one-line **If** statement doesn’t even need the **Else** part. The function below, **OneLineIf2**, returns 0 if the condition “Parameter > 5” is not fulfilled:

```
Function OneLineIf2(Parameter)
    If Parameter > 5 Then OneLineIf = 1
End Function
```

	A	B	C
6	ONELINEIF2 IN ACTION		
7	Parameter		
8	12	1	<-- =OneLineIf2(A8)
9	3	0	<-- =OneLineIf2(A9)

**Good Programming Practice: Assign a Value to Your Function *First***

In the above functions, it would be good programming practice to first assign a value to the function before introducing the **If** statement. This way we know that **OneLineIf3** defaults to -16 if the condition on **Parameter** is not fulfilled.

```
Function OneLineIf3(Parameter)
    OneLineIf3 = -16
    If Parameter > 5 Then OneLineIf3 = 1
End Function
```

To see the difference this makes, look at the spreadsheet below:

	A	B	C
11	ONELINEIF3 IN ACTION		
12	Parameter		
13	12	1	<-- =OneLineIf3(A13)
14	3	-16	<-- =OneLineIf3(A14)

**If ... ElseIf Statements**

If more than one statement is to be conditionally executed, the block **If...ElseIf** statement can be used. It uses the following syntax:

```
If Condition0 Then
    Statements
ElseIf Condition1 Then
    Statements
```

[... More Elselfs ...]

**Else**  
Statements  
**End If**

The **Else** and **Elself** clauses are both optional. You may have as many **Elself** clauses as you want following an If, but none can appear after an **Else** clause. **If** statements can be contained within one another. Here's an example:

```
Function BlockIf(Parameter)
    If Parameter < 0 Then
        BlockIf = -1
    ElseIf Parameter = 0 Then
        BlockIf = 0
    Else
        BlockIf = 1
    End If
End Function
```

Here's how this function works in Excel:

	A	B	C
23	<b>BLOCKIF IN ACTION</b>		
24	Parameter		
25	-3	-1	=BlockIf(A25)
26	0	0	=BlockIf(A26)
27	13	1	=BlockIf(A27)

**Nested If Structures**

As stated in the previous section, **If** statements can be used as part of the statements used in another **If** statement. A program structure that has some **If** statements inside others is called a *nested If* structure. Each **If** statement in the structure must be a complete **If** statement. Either the one-line or the block version can be used.

The following function demonstrates the use of the **NestedIf** structure:

```
Function NestedIf(P1, P2)
  If P1 > 10 Then
    If P2 > 5 Then NestedIf = 1 Else NestedIf
    = 2
  ElseIf P1 < -10 Then
    If P2 > 5 Then
      NestedIf = 3
    Else
      NestedIf = 4
    End If
  Else
    If P2 > 5 Then
      If P1 = P2 Then NestedIf = 5 Else
      NestedIf = 6
    Else
      NestedIf = 7
    End If
  End If
End Function
```

This is how it looks in Excel:

	A	B	C	D
30	<b>NESTEDIF IN ACTION</b>			
31	11	6	1	<-- =NestedIf(A31,B31)
32	22	3	2	<-- =NestedIf(A32,B32)
33	-22	6	3	<-- =NestedIf(A33,B33)
34	-57.3	4	4	<-- =NestedIf(A34,B34)
35	6	6	5	<-- =NestedIf(A35,B35)
36	-5	7	6	<-- =NestedIf(A36,B36)
37	4	3	7	<-- =NestedIf(A37,B37)

---

### 36.7 The Boolean and Comparison Operators

The expressions used as conditions in an **If** statement are also known as Boolean expressions. Boolean expressions can have one of two values: **TRUE** when the condition holds, and **FALSE** when the condition is violated. Usually Boolean expressions are constructed using the Comparison and/or Boolean operators. The following is a list of the most common Comparison operators.

Operator	Meaning
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
=	Equal to
<>	Not equal to

#### The And Boolean Operator

The next function uses a Boolean operator to check whether two conditions hold at the same time.

```
Function AndDemo(parameter1, parameter2)
    If (parameter1 < 10) And (parameter2 > 15) _
    Then
        AndDemo = 3
    Else
        AndDemo = 12
    End If
End Function
```

Here are some illustrations:

	A	B	C	D
1	ANDDEMO IN ACTION			
2	parameter1	parameter2		
3	9	14	12	<-- =AndDemo(A3,B3)
4	9	16	3	<-- =AndDemo(A4,B4)
5	11	14	12	<-- =AndDemo(A5,B5)
6	11	16	12	<-- =AndDemo(A6,B6)

Notice what **AndDemo** does: It checks **both** conditions (parameter1 < 10) **and** (parameter2 > 15). If both conditions hold, then the combined conditions hold and the function returns a value of 3. Otherwise (i.e., if either one of the conditions is violated) it returns 12. (Note that both conditions are in parentheses.)

The following function and screen shot demonstrate all four possible combinations of two conditions and the resulting combined condition:

```
Function AndTable(parameter1, parameter2)
    AndDemoTable = parameter1 And parameter2
End Function
```

	A	B	C	D
1	ANDDEMO IN ACTION			
2	parameter1	parameter2		
3	9	14	12	<-- =AndDemo(A3,B3)
4	9	16	3	<-- =AndDemo(A4,B4)
5	11	14	12	<-- =AndDemo(A5,B5)
6	11	16	12	<-- =AndDemo(A6,B6)

The Or Boolean Operator

The function **OrDemo**, illustrated below, checks whether at least one of two conditions holds:

```
Function OrDemo(parameter1, parameter2)
    If (parameter1 < 10) Or (parameter2 > 15) _
    Then
        OrDemo = 3
    Else
        OrDemo = 12
    End If
End Function
```

	A	B	C	D
18	ORDEMO IN ACTION			
19	parameter1	parameter2		
20	9	14	3	<-- =OrDemo(A20,B20)
21	9	16	3	<-- =OrDemo(A21,B21)
22	11	14	12	<-- =OrDemo(A22,B22)
23	11	16	3	<-- =OrDemo(A23,B23)

Notice what **OrDemo** does: It checks whether **either** the first condition (Parameter1 < 10) **or** the second condition (Parameter2 > 15) **or both** conditions hold. Only if both conditions are violated will the function return a value of 12. Otherwise (i.e., if either one or both of the conditions hold) it returns 3. (Note that both conditions are in parentheses.)

The following function and the screen shot demonstrate all four possible combinations of two conditions and the resulting combined condition:

```
Function OrDemoTable(parameter1, parameter2)
    OrDemoTable = parameter1 Or parameter2
End Function
```

	A	B	C	D
1	<b>ORTABLE IN ACTION</b>			
2	parameter1	parameter2		
3	FALSE	FALSE	FALSE	<-- =ORDemoTable(A3,B3)
4	FALSE	TRUE	TRUE	<-- =ORDemoTable(A4,B4)
5	TRUE	FALSE	TRUE	<-- =ORDemoTable(A5,B5)
6	TRUE	TRUE	TRUE	<-- =ORDemoTable(A6,B6)

---

## 36.8 Loops

Looping structures are used when you need to do something repeatedly. As always there is more than one way to achieve the desired effect. In general there are two major looping constructs:

- *A top-checking loop:* The loop condition is checked before anything else gets done. The something to be done can be left undone if the condition is not fulfilled on entry to the loop.
- *A bottom-checking loop:* The loop condition is checked after the something to be done is done. The something to be done will always be done at least once.

VBA has the two major looping structures covered from all possible angles by the **Do** statement and its variations. All the following subsections will use a version of the factorial function for demonstration purposes. The function used is defined as:

$$f(0) = 1 \quad f(1) = 1 \quad f(2) = 2 * f(1) = 2 \dots f(n) = n * f(n-1)$$



The Do While Statement

The **Do While** statement is a member of the top-checking loops family. It makes VBA execute one or more statements **zero** or more times, while a condition is true. The following function demonstrates this behavior:

```
Function DoWhileDemo (N)
  If N < 2 Then
    DoWhileDemo = 1
  Else
    i = 1
    j = 1
    Do While i <= N
      j = j * i
      i = i + 1
    Loop
    DoWhileDemo = j
  End If
End Function
```

	A	B	C
1	<b>DOWHILEDEMO IN ACTION</b>		
2	5	120	<-- =DoWhileDemo(A2)
3	9	362880	<-- =DoWhileDemo(A3)
4	13	6227020800	<-- =DoWhileDemo(A4)

The Do ... Loop While Statement

The **Do ... Loop While** statement is a member of the bottom-checking loops family. It makes VBA execute one or more statements **one** or more times, while a condition is true. The following function demonstrates this behavior:

```
Function DoLoopWhileDemo (N)
    If N < 2 Then
        DoLoopWhileDemo = 1
    Else
        i = 1
        j = 1
        Do
            j = j * i
            i = i + 1
        Loop While i <= N
        DoLoopWhileDemo = j
    End If
End Function
```

	A	B	C
1	<b>DOLOOPWHILEDEMO IN ACTION</b>		
2	5	120	<-- =DoLoopWhileDemo(A2)
3	9	362880	<-- =DoLoopWhileDemo(A3)
4	13	6227020800	<-- =DoLoopWhileDemo(A4)

The Do Until Statement

The **Do Until** statement is a member of the top-checking loops family. It makes VBA execute one or more statements **zero** or more times, until a condition is met. The following function demonstrates this behavior:

```
Function DoUntilDemo (N)
    If N < 2 Then
        DoUntilDemo = 1
    Else
        i = 1
        j = 1
        Do Until i > N
            j = j * i
            i = i + 1
        Loop
        DoUntilDemo = j
    End If
End Function
```

	A	B	C
1	<b>DOUNTILDEMO IN ACTION</b>		
2	5	120	<-- =DoUntilDemo(A2)
3	9	362880	<-- =DoUntilDemo(A3)
4	13	6227020800	<-- =DoUntilDemo(A4)

The Do ... Loop Until Statement

The **Do ... Loop Until** statement is a member of the bottom-checking loops family. It makes VBA execute one or more statements **one** or more times, until a condition becomes true. The following function demonstrates this behavior:

```
Function DoLoopUntilDemo (N)
  If N < 2 Then
    DoLoopUntilDemo = 1
  Else
    i = 1
    j = 1
    Do
      j = j * i
      i = i + 1
    Loop Until i > N
    DoLoopUntilDemo = j
  End If
End Function
```

	A	B	C
1	<b>DOLOOPUNTILDEMO IN ACTION</b>		
2	5	120	<-- =DoLoopUntilDemo(A2)
3	9	362880	<-- =DoLoopUntilDemo(A3)
4	13	6227020800	<-- =DoLoopUntilDemo(A4)

The For Loop

One last (for now) variation on the loopy theme, the **For** loop, is used mainly for loops where the number of times the action is repeated is known in advance. The following functions demonstrate its use and variations:

```
Function ForDemo1(N)
    If N <= 1 Then
        ForDemo1 = 1
    Else
        j = 1
        For i = 1 To N Step 1
            j = j * i
        Next i
        ForDemo1 = j
    End If
End Function
```

	A	B	C
1	<b>FORDEMO1 IN ACTION</b>		
2	5	120	<-- =ForDemo1(A2)
3	9	362880	<-- =ForDemo1(A3)
4	13	6227020800	<-- =ForDemo1(A4)

The **Step** part of the statement can be dropped if (as is in our case) the increment is 1. For example:

```
For i = 1 To N
    j = j * i
Next i
```

If you want the loop to count down, the **Step** argument can be negative, as demonstrated in the next function:

```
Function ForDemo2(N)
  If N <= 1 Then
    ForDemo2 = 1
  Else
    j = 1
    For i = N To 1 Step -1
      j = j * i
    Next i
    ForDemo2 = j
  End If
End Function
```

	A	B	C
1	<b>FORDEMO2 IN ACTION</b>		
2	5	120	<-- =ForDemo2(A2)
3	9	362880	<-- =ForDemo2(A3)
4	13	6227020800	<-- =ForDemo2(A4)

The **For** loop can be exited early by using the **Exit For** statement as demonstrated in the next function (not the factorial function).

```
Function ExitForDemo(Parameter1, Parameter2)
  Sum = 0
  For i = 1 To Parameter1
    Sum = Sum + i
    If Sum > Parameter2 Then Exit For
  Next i
  ExitForDemo = Sum
End Function
```

	A	B	C	D
1	<b>EXITFORDEMO IN ACTION</b>			
2	Parameter1	Parameter2		
3	5	22	15	<-- =ExitForDemo(A3,B3)
4	6	22	21	<-- =ExitForDemo(A4,B4)
5	7	22	28	<-- =ExitForDemo(A5,B5)
6	8	22	28	<-- =ExitForDemo(A6,B6)

### 36.9 Using Excel Functions in VBA

VBA can use most of Excel's worksheet functions. We illustrate by showing how to define the binomial distribution (even though this, itself, is an Excel function). The probability distribution of a binomial random variable is defined

as  $Binom(p, n, x) = \binom{n}{x} p^x (1-p)^{n-x}$  where  $p$  is the probability of success;  $x$  is the number of successes, and  $n$  is the number of trials.  $\binom{n}{x} = \frac{n!}{(n-x)!x!}$

is the binomial coefficient, which gives the number of ways of choosing  $x$  elements from among  $n$  elements. For example, suppose you want to form a two-person team from eight candidates and you want to know how many possible teams can be formed. The answer is given by  $\binom{8}{2} = \frac{8!}{6!2!} = \frac{8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}{6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 2 \cdot 1} = 28$ . The Excel function **Combin**(8, 2) does this calculation.

We use this Excel function in the following VBA function:

```
Function Binomial(p, n, x)
    Binomial = Application.WorksheetFunction. _
        Combin(n, x) * p ^ x * (1 - p) ^ (n - x)
End Function
```

As usual, this can be applied inside a spreadsheet:

	A	B	C
1	<b>BINOMIAL IN ACTION</b>		
2	p	0.5	
3	n	10	
4	x	6	
5	Binomial	0.20507813	<-- =Binomial(B2,B3,B4)

Note that we used **Application.WorksheetFunction.Combin(n, x)** to compute  $\binom{n}{x}$  in our function. As you might guess from its name (**Application.WorksheetFunction.Something**), this function is the Excel Worksheet function **Combin( )**. Most, but not all,<sup>5</sup> Excel worksheet functions can be used in VBA in exactly the same way. For a complete list see the Help file.

One more thing to notice is the underscore ( ) preceded by a space at the end of line 2. If a line gets too long to deal with, it can be continued on the next line using this contraption (the second and third lines of **Binomial** are one line as far as VBA is concerned).<sup>6</sup>

Suppose we try to use our **Binomial** function to calculate **Binomial(0.5,10,15)**. This won't work:

	A	B	C
1	<b>BINOMIAL IN ACTION</b>		
2	p	0.5	
3	n	10	
4	x	15	
5	Binomial	#VALUE!	<-- =Binomial(B2,B3,B4)

5. When an equivalent function is available as a native VBA function, the corresponding Excel function is not available in VBA. For example, in VBA use **rnd( )** and not **Application.WorksheetFunction.Rand( )** and **sqr( )** and not **Application.WorksheetFunction.Sqrt( )**.

6. What's too long? This is a matter of programming taste, but for our purposes any line over 70–80 characters is considered too long



The reason for the problem is that in the computation  $\binom{n}{x}$  used in **Binomial**, we have to have  $x < n$ . In this case, VBA causes Excel to return the error message **#VALUE!**. The subject of Excel error values is somewhat obscure and is discussed in the Appendix to this chapter.

---

### 36.10 Using User-Defined Functions in User-Defined Functions

User-defined functions can be used in other user-defined functions, just like Excel functions. The next function is a replacement for the COMBIN worksheet function. COMBIN is defined as  $c(n, x) = \frac{n!}{(n-x)!x!}$  where  $!$  stands for the factorial function. (Recall that the factorial function  $n!$  is defined for any  $n > 0$ :  $0! = 1$ , and for  $n > 0$ ,  $n! = n*(n-1)*(n-2) \dots 1$ .)

We will now write our VBA version of the two functions: the factorial function and the COMBIN function.

```

1 Function HomeFactorial(n)
2   If Int(n) <> n Then
3     HomeFactorial = CVErr(xlErrValue)
4   ElseIf n < 0 Then
5     HomeFactorial = CVErr(xlErrNum)
6   ElseIf n = 0 Then
7     HomeFactorial = 1
8   Else
9     HomeFactorial = HomeFactorial(n - 1) * n
10  End If
11 End Function

```

Line 2 checks if the input is an integer by comparing the integer part of “n” to “n.” The function “Int” is a part of VBA. If we have erred, for example, by

asking for **HomeFactorial(3.3)**, then line 3 of the program will cause Excel to return **#VALUE!**. Similarly, lines 4 and 5 check if we have improperly asked for **HomeFactorial** of a negative number; if this is the case, then line 5 causes Excel to return **#NUM!**. For a fuller explanation of the use of error values, see the Appendix to this chapter.

Line 9 introduces a new concept; the function uses itself to calculate the value it should return. This is called recursion. Here’s an illustration of the function in action:

	A	B	C	D	E
1	RECURSION IN ACTION				
2	1	1	<-- 1	1	<-- =HomeFactorial(A2)
3	2	2	<-- =B2*A3	2	<-- =HomeFactorial(A3)
4	3	6	<-- =B3*A4	6	<-- =HomeFactorial(A4)
5	4	24	<-- =B4*A5	24	<-- =HomeFactorial(A5)
6	5	120	<-- =B5*A6	120	<-- =HomeFactorial(A6)

We can now use **HomeFactorial** to create our VBA version of **Combin** (which we will call **HomeCombin**):

```
Function HomeCombin(n, x)
    HomeCombin = HomeFactorial(n) / _
        (HomeFactorial(n - x) * HomeFactorial(x))
End Function
```

Finally, we can use **HomeCombin** to create a VBA version of the binomial function:

```
Function HomeBinom(p, n, x)
    If n < 0 Then
        HomeBinom = CVErr(xlErrValue) 'Make the function
                                         'return #VALUE!

    ElseIf x > n Or x < 0 Then
        HomeBinom = CVErr(xlErrNum)    'Make the function
                                         'return #NUM!

    Else
        HomeBinom = HomeCombin(n, x) _
                    * p ^ x * p ^ (n - x)

    End If
End Function
```

**Putting Comments in VBA Code**

As illustrated above, VBA will ignore anything on a line which follows an apostrophe (note that each new line of comments has to begin with an apostrophe).

---

**Exercises**

- 1. Write a VBA function for  $f(x) = x^2 - 3$ .

	A	B	C
1	<b>Exercise 1</b>		
2			
3	X		
4	1	-2	<-- =Exercise1(A4)
5	2	1	<-- =Exercise1(A5)
6	3	6	<-- =Exercise1(A6)

2. Write a VBA function for  $f(x) = \sqrt{2x^2} + 2x$ . Note that there are two ways to do this: The first is to use the VBA function **Sqr**. The second is to use the VBA operator “^”. We suggest you try both.

	A	B	C
8	<b>Exercise 2</b>		
9			
10	X	Exercise2	
11	1	3.414213562	<-- =Exercise2(A10)
12	2	6.828427125	<-- =Exercise2(A10)
13	1	3.414213562	<-- =Exercise2a(A12)
14	2	6.828427125	<-- =Exercise2a(A13)

3. Suppose a share was priced at price  $P_0$  at time 0, and suppose that at time 1 it will be priced  $P_1$ . Then the continuously compounded return is defined as  $return = \ln\left(\frac{P_1}{P_0}\right)$ . Implement this function in VBA. There are two ways to do this: You can use **Worksheetfunction.Ln** or the VBA function **Log**.

	A	B	C	D
18	<b>Exercise 3</b>			
19				
20	P0	P1		
21	100	110	0.09531018	<-- =Exercise3(A21,B21)
22	100	200	0.693147181	<-- =Exercise3(A22,B22)
23	100	110	0.09531018	<-- =Exercise3a(A23,B23)
24	100	200	0.693147181	<-- =Exercise3a(A24,B24)

4. A bank offers different yearly interest rates to its customers based on the size of the deposit in the following way:

- For deposits up to 1,000, the interest rate is 5.5%
- For deposits from 1,000 and up to 10,000, the interest rate is 6.3%
- For deposits from 10,000 and up to 100,000, the interest rate is 7.3%
- For all other deposits the interest rate is 7.8%

Implement the function **Interest(Deposit)** in VBA. Note that you can use the **BlockIf** structure.

	A	B	C
1	<b>Exercise 4</b>		
2	Deposit		
3	-1	#VALUE!	<-- =Interest(A3)
4	100	5.50%	<-- =Interest(A4)
5	1100	6.30%	<-- =Interest(A5)
6	9999.99	6.30%	<-- =Interest(A6)
7	10000	6.30%	<-- =Interest(A7)
8	10000.001	7.30%	<-- =Interest(A8)
9	100000.001	7.80%	<-- =Interest(A9)

5. Using the function in exercise 4, implement a function **NewDFV(Deposit, Years)**. The function will return the future value of a deposit with the bank assuming the deposit and accrued interest is reinvested for a given number of years. Thus, for example, **NewDFV(10000,10)** will return  $10000 \times (1.063)^{10}$ .

	A	B	C	D
1	<b>Exercise 5</b>			
2	Deposit	Years		
3	10000	10	18421.82	<-- =NewDFV(A3,B3)
4	10000.001	10	20230.06	<-- =NewDFV(A4,B4)

6. An investment company offers a bond linked to the FT100 index. On redemption the bond pays the face value plus the largest of A: the face value times the change in the index. Or B: 5% yearly interest compounded monthly. Thus, for example, 100 invested when the index was 110 and redeemed a year later when the index was 125 will pay A:  $100 + 100 \times (125 - 110)/110 = 113.636$  and not B:  $100 \times (1 + 0.05/12)^{12} = 105.116$ . Implement a VBA function **Bond(Deposit, Years, FT0, FT1)**.

	A	B	C	D	E	F
64	<b>Exercise 6</b>					
65						
66	Deposit	Years	FT0	FT1		
67	100	1	110	125	113.636	<-- =Bond(A67,B67,C67,D67)
68	100	1	110	100	105.116	<-- =Bond(A68,B68,C68,D68)
69	100	12	110	125	1,261.394	<-- =Bond(A69,B69,C69,D69)
70	100	12	110	1387.53	1,261.394	<-- =Bond(A70,B70,C70,D70)
71	100	12	110	1387.535	1,261.395	<-- =Bond(A71,B71,C71,D71)

7. Implement a VBA function **ChooseBond(Deposit, Years, FT0, FT1)**. The function will return the value 1 if the superior investment is the bank in exercise 5 or the value 2 if it is the company in exercise 6.

	A	B	C	D	E	F
76	<b>Exercise 7</b>					
77						
78	Deposit	Years	FT0	FT1		
79	100	1	110	125	2	<-- =ChooseBond(A79,B79,C79,D79)
80	100	1	110	110	1	
81	100	1	110	116.04	1	
82	100	1	110	116.05	2	
83	100000	1	110	125	2	<-- =ChooseBond(A83,B83,C83,D83)
84	100000	1	110	110	1	
85	100000	1	110	118.02	1	
86	100000	1	110	118.03	2	

8. A bank offers the following saving scheme: Invest a fixed amount on the first of each month for a set number of years. On the first of the month after your last installment, you get your money plus the accrued interest. The bank quotes a yearly interest rate but interest is calculated and compounded on a monthly basis. Eight different interest rates are offered depending on the monthly deposit and the number of years the program is to run. The following table lists the interest rates offered.

	For sums ≤ 100 a month	For sums > 100 a month
For a period of 2 years	3.5%	3.9%
For a period of 3 years	3.7%	4.5%
For a period of 4 years	4.2%	5.1%
For a period of 5 years	4.6%	5.6%

Write a two-argument function **DFV(Deposit, Years)**, returning the future value of such an investment.

	A	B	C	D
1	<b>Exercise 8</b>			
2	Deposit	Years	DFV	
3	10	5	675.7458	<-- =DFV(A3,B3)
4	10	4	523.5107	<-- =DFV(A4,B4)
5	10	3	381.2934	<-- =DFV(A5,B5)
6	10	2	248.9488	<-- =DFV(A6,B6)
7	10	1	120	<-- =DFV(A7,B7)

9. Using the information provided in exercise 8 write a two-argument function **DEP(DFV, Years)** that will return the monthly contribution necessary to get a certain sum in the future (2, 3, 4, or 5 years). Note: This problem is more interesting; remember that the interest rate depends on the monthly contribution.

	A	B	C	D
1	<b>Exercise 9</b>			
2	DFV	Years	DEP	
3	-100	2	-4.01689	<-- =DEP(A3,B3)
4	100	2	4.01689	<-- =DEP(A4,B4)
5	1000	4	19.10181	<-- =DEP(A5,B5)
6	2499	2	99.96106	<-- =DEP(A6,B6)
7	2500	2	100.0011	<-- =DEP(A7,B7)

10. Fibonacci numbers are named after Leonardo Fibonacci (1170–1230), an outstanding European mathematician of the medieval period. Fibonacci numbers are defined as follows:

$F(0)=0$

$F(1)=1$

$F(2)=F(0)+F(1)=1$

$F(3)=F(1)+F(2)=2$

$F(4)=F(2)+F(3)=3$

...

In general  $F(n)=F(n-2)+F(n-1)$ .

Write a **recursive** VBA function that computes the  $n$ th number in the Fibonacci series.

	A	B	C
1	<b>Exercise 10</b>		
2	n	Fibonacci	
3	0	0	<-- =Fibonacci(A3)
4	1	1	<-- =Fibonacci(A4)
5	2	1	<-- =Fibonacci(A5)
6	3	2	<-- =Fibonacci(A6)
7	4	3	<-- =Fibonacci(A7)
8	5	5	<-- =Fibonacci(A8)
9	6	8	<-- =Fibonacci(A9)
10	7	13	<-- =Fibonacci(A10)

11. Write a VBA function that computes the *n*th number in the Fibonacci series; do not use recursion.

	A	B	C
1	Exercise 11		
2	n	LoopFibonacci	
3	0	0	<-- =LoopFibonacci(A3)
4	1	1	<-- =LoopFibonacci(A4)
5	2	1	<-- =LoopFibonacci(A5)
6	3	2	<-- =LoopFibonacci(A6)
7	4	3	<-- =LoopFibonacci(A7)
8	5	5	<-- =LoopFibonacci(A8)
9	6	8	<-- =LoopFibonacci(A9)
10	7	13	<-- =LoopFibonacci(A10)

Appendix: Cell Errors in Excel and VBA

Excel uses a special kind of value to report errors. The **CVErr( )** function is part of VBA. It converts a value, supplied by you, to the special kind of value used for errors in Excel. Excel has a number of error values that a function can return to signal that something went wrong. Here’s an example: The function **NewMistake(x,y)** returns the result *x/y*. However, if *y* = 0, the function outputs the (cryptic) error message #DIV0!.

```
Function NewMistake(x, y)
    If y <> 0 Then NewMistake = x / y Else _
        NewMistake = CVErr(xlErrDiv0)
End Function
```

**To Anticipate Future Confusion**

All the VBA error values are written “xlErr ... .” Because the typed alphabet letter “l” also looks like the number one, it would have been easier had Microsoft used capital letters “XLErr ... .” But ...



This is **NewMistake** in Excel:

	A	B	C	D
1	NewMistake In Action			
2				
3	X	Y	NewMistake	
4	1	2	0.5	<-- =NewMistake(A4,B4)
5	2	1	2	<-- =NewMistake(A5,B5)
6	0	1	0	<-- =NewMistake(A6,B6)
7	1	0	#DIV/0!	<-- =NewMistake(A7,B7)

Error values and their explanation are listed below.

Error Value	VBA Name	Possible causes
#NULL!	XIErrNull	The #NULL! error value occurs when you specify an intersection of two areas that do not intersect.
#DIV/0!	XIErrDiv0	The #DIV/0! error value occurs when a formula divides by 0 (zero).
#VALUE!	XIErrValue	The #VALUE! error value occurs when the wrong type of argument is used.
#REF!	XIErrRef	The #REF! error value occurs when a cell reference is not valid.
#NAME?	XIErrName	The #NAME? error value occurs when Microsoft Excel doesn't recognize text in a formula.
#NUM!	XIErrNum	The #NUM! error value occurs when a problem occurs with a number in a formula or function.
#N/A	XIErrNA	The #N/A error value occurs when a value is not available to a function or formula.



---

# 37 Variables and Arrays

---

## 37.1 Overview

In the first part of this chapter we introduce function variable definitions. The second part of the chapter introduces arrays. An array is a group of variables of the same type sharing the same name and referenced individually using an index. Vectors and matrices are good examples of one- and two-dimensional arrays. The relationship between arrays and worksheet ranges opens the discussion, followed by sections describing simple and dynamic arrays (whose size can be changed at run time). The chapter concludes with sections on the use of arrays as parameters and a short discussion of typed variables.

---

## 37.2 Defining Function Variables

Function variables are used to store values. Function variables can be either parameter or simple variables. Parameters are defined when the function is defined by listing them within parentheses after the function's name. Up until now we used simple variables as and when needed, relying on VBA to define the variable for us when it was first used. In most scenarios encountered in this book, this practice is good enough, and it has the advantage of being quick.

The first time we encountered both flavors of function variables was in the function **DoWhileDemo**:

```
Function DoWhileDemo (N)
    If N < 2 Then
        DoWhileDemo = 1
    Else
        i = 1 ' A Loop counter
        j = 1 ' An accumulator for the series
        Do While i <= N
            j = j * i
            i = i + 1
        Loop
        DoWhileDemo = j
    End If
End Function
```

The variable **N** is a parameter that gets its value from the application that activates the function (either Excel or another function). The variables **i**, **j** are simple variables. Function variables (aka internal or local variables) of both types are recognized only in the function in which they were defined (implicitly or explicitly) and are not recognized by Excel or by other VBA functions.

As this is a very short function, there really is no reason to define the variables explicitly and the addition of comments makes everything clear enough. Longer functions with more variables might benefit by defining the variables at the top of the function, as it makes for more maintainable, and clear programming. Simple variables are defined using the **Dim** statement as demonstrated by the following function:

```
Function NewDoWhileDemo (N)
Dim i ' a Loop counter
Dim j ' An accumulator for the series
    If N < 2 Then
        NewDoWhileDemo = 1
    Else
        i = 1
        j = 1
        Do While i <= N
            j = j * i
            i = i + 1
        Loop
        NewDoWhileDemo = j
    End If
End Function
```

### The Option Explicit Statement

We can make VBA alert us if we use an undeclared variable by inserting the **Option Explicit** statement as the first line in the module. With this statement any use of an undeclared variable will result in an error and not the creation of a new variable. The **Option Explicit** statement holds for all the routines in the module.

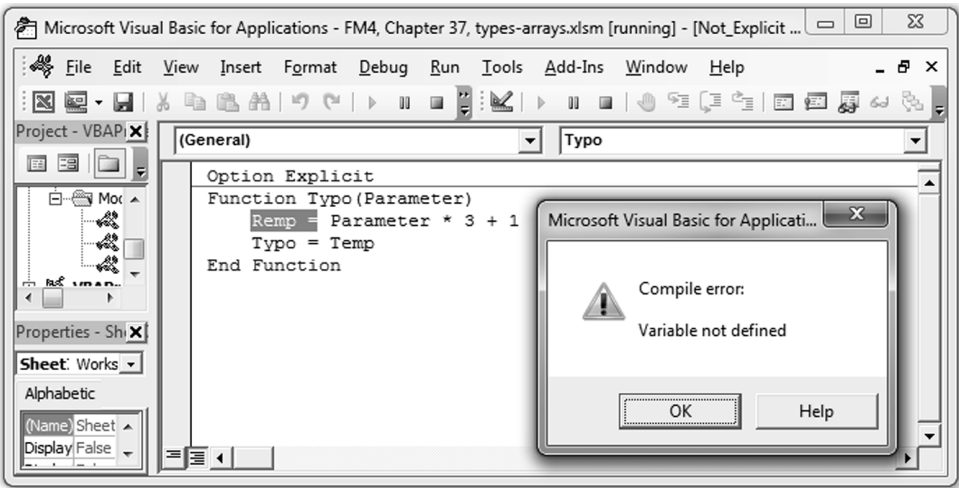
Forcing the definition of variables can help prevent errors from creeping into your functions. Here is an (slightly forced) example: The following function contains a typing error (“Temp” is spelled “Remp”):


```
Function Typo(Parameter)
    Remp = Parameter * 3 + 1
    Typo = Temp
End Function
```

Without the **Option Explicit** statement, Excel merrily displays the following result:

	A	B	C
1	TYPO IN ACTION		
2	5	0	=Typo(A2)

However, inserting the **Option Explicit** statement before the VBA code and recalculating the worksheet results in the following “Run Time Error”:



Once we are alerted to the problem, we can click the OK button, stop VBA from running, and fix the problem by replacing “Remp” with “Temp.” (Recall from Chapter 36 that after you fix the mistake in VBA, you have to press the  button on the VBA editor toolbar.)

37.3 Arrays and Excel Ranges

A VBA array is a group of variables of the same type sharing the same name and referenced individually using an index (or indices). VBA has its own version of arrays and we shall deal with this type of array in the following sections. For now let us demonstrate the **Variant**. If we want a function to accept an Excel range as a parameter we have to leave the parameter type-less, or declare the parameter as Variant (which amounts to the same thing). From inside the function the variable looks like an array. To demonstrate, we shall now write a small function, **SumRange**, that sums the value in the first four elements of its parameter.

```
Function SumRange(R)
    S = 0
    For i = 1 To 4
        S = R(i) + S
    Next i
    SumRange = S
End Function
```

	A	B
1	SUMRANGE IN ACTION	
2	1	<-- 1
3	2	<-- 2
4	3	<-- 3
5	4	<-- 4
6	10	<-- =SumRange(A2:A5)

	A	B	C	D
1	SUMRANGE IN ACTION			
2	1	2	3	4
3	10	<-- =SumRange(A2:D2)		

In both cases the variable *R* can be treated as an array, with the first element being **R(1)** and the last element **R(4)**. Each of the elements can be treated as a single variable, that is, **R(2)** is a variable and so is **R(i-3)** (assuming that *i*-3 has an integer value  $\geq 1$  and  $\leq 4$ ). Ranges treated as arrays always start with index 1.

What happens if the range passed to our function is rectangular? To demonstrate, we introduce a modified version of **SumRange**, inserting a second parameter that tells the function how many elements to sum.

```
Function SumRange1(R, N)
    S = 0
    For i = 1 To N
        S = R(i) + S
    Next i
    SumRange = S
End Function
```

	A	B	C	D
1	<b>SUMRANGE1 IN ACTION</b>			
2	3	4	5	
3	6	7	8	
4	9	10	11	
5	18	=<SumRange1(A2:C4,4)		
6	25	=<SumRange1(A2:C4,5)		
7	33	=<SumRange1(A2:C4,6)		

As we can see, VBA treats the rectangular array as a linear array composed of the rows of the original range. The second parameter of the function **SumRange1** indicates how many elements should be summed. Thus, for example, Sumrange1(A2:C4,5) sums the first row plus two cells in the second row.

### A Payback Period Function

A slightly more complex use of ranges can be shown with a simple Payback Period function. Recall that the payback period in capital budgeting refers to

the period of time required for the return on an investment to “repay” the sum of the original investment. For example, a \$1,000 investment which pays cash flows of \$500 per year has a 2-year payback period. To simplify matters, the function **PayBack** defined below gives a whole year solution. If the sum of the cash flows for 5 years is  $< 0$  and for 6 years is  $> 0$ , then the function will return 6. We also assume that the first cash flow is the initial investment (negative) and that no other cash flows are negative.

```
Function PayBack(R, N)
    Temp = 0
    For i = 1 To N
        Temp = Temp + R(i)
        If Temp >= 0 Then Exit For
    Next i
    PayBack = i - 1
End Function
```

	A	B	C	D	E	F
1	<b>PAYBACK IN ACTION</b>					
2	Period	1	2	3	4	5
3	Cash-flow	-1500	400	600	600	300
4	PayBack	3 <-- =PayBack(B3:F3,5)				

There are a few problems with this function as currently defined. One is that the function returns a wrong answer if the investment does not pay back its initial outlay, as demonstrated by the next screen shot.

	A	B	C	D	E	F
1	<b>PAYBACK IN ACTION</b>					
2	Period	1	2	3	4	5
3	Cash-flow	-4000	400	600	600	300
4	PayBack	5 <-- =PayBack(B3:F3,5)				



The problem is solved by inserting a check before returning the payback period.

```
Function PayBack1(r, n)
    Temp = 0
    For i = 1 To n
        Temp = Temp + r(i)
        If Temp >= 0 Then Exit For
    Next i
    If Temp >= 0 Then
        PayBack1 = i - 1
    Else
        PayBack1 = "No Payback"
    End If
End Function
```

	A	B	C	D	E	F
1	<b>PAYBACK1 IN ACTION</b>					
2	Period	1	2	3	4	5
3	Cash-flow	-4000	400	600	600	300
4	PayBack	No Payback	<-- =PayBack1(B3:F3,5)			

### 37.4 Simple VBA Arrays

There are several ways to declare VBA arrays, all using the **Dim** statement. The simplest way to declare an array is simply to tell VBA the largest value the array index can take. Unless you indicate otherwise, VBA arrays always start with index 0. In the function below, **MyArray** has 6 elements numbered 0, 1, 2, ... , 5.

```

Function ArrayDemo1()
    Dim MyArray(5)
    For i = 0 To 5
        MyArray(i) = i * i
    Next i
    S = ""
    For i = 0 To 5
        S = S & " # " & MyArray(i)
    Next i
    ArrayDemo1 = S
End Function

```

If you use **ArrayDemo1** in a spreadsheet, here is the result:

	A	B
1	<b>ARRAYDEMO1 IN ACTION</b>	
2	# 0 # 1 # 4 # 9 # 16 # 25	<-- =ArrayDemo1()

Notice:

- **MyArray** has six elements (variables), the first being **MyArray(0)** and the last **MyArray(5)**. All VBA arrays start from 0, unless you specify otherwise (see discussion of **Option Base** below).
- An array element is treated just like a variable. **MyArray(2)** is a variable and so is **MyArray(i-3)** (assuming that i-3 has an integer value  $\geq 0$  and  $\leq 5$ ).
- The use of the concatenation operator **&**. This operator concatenates (combines) its two operands to create a string. If an operand to the concatenation operator is not a string, it is converted to a string, and then the concatenation takes place.

If you try and access an array element that is not part of the array, VBA will return an error value, as demonstrated by the following function:

```

Function ArrayDemo2 (N)
    Dim MyArray(5)
    Dim i As Integer
    For i = 0 To 5
        MyArray(i) = i * i
    Next i
    ArrayDemo2 = MyArray(N)
End Function

```

	A	B	C
1	<b>ARRAYDEMO2 IN ACTION</b>		
2	0	0	0 <-- =ArrayDemo2(A2)
3	1	1	1 <-- =ArrayDemo2(A3)
4	2	4	4 <-- =ArrayDemo2(A4)
5	3	9	9 <-- =ArrayDemo2(A5)
6	4	16	16 <-- =ArrayDemo2(A6)
7	5	25	25 <-- =ArrayDemo2(A7)
8	6	#VALUE!	<-- =ArrayDemo2(A8)

### LBound and UBound

**LBound** and **UBound** are two internal VBA functions that are very useful when dealing with arrays. These functions return the minimum and maximum value that an array index can have. The following function demonstrates their use on a one-dimensional array:

```

Function ArrayDemo3 (N)
    Dim MyArray(5)
    If N = "LB" Then
        ArrayDemo3 = LBound(MyArray)
    ElseIf N = "UB" Then
        ArrayDemo3 = UBound(MyArray)
    End If
End Function

```

	A	B	C
1	ARRAYDEMO3 IN ACTION		
2	LB	0	<-- =ArrayDemo3(A2)
3	UB	5	<-- =ArrayDemo3(A3)

Note that the array **MyArray** has six elements, the first being **MyArray(0)** as indicated by **LBound**, and the last being **MyArray(5)** as indicated by **UBound**.

When used on a multidimensional array, a second parameter should be supplied indicating the dimension in whose bounds we are interested, as the next function demonstrates.

```
Function ArrayDemo4(Dimension, Bound)
  Dim MyArray(2, 3, 4)
  If Bound = "LB" Then
    ArrayDemo4 = LBound(MyArray, Dimension)
  ElseIf Bound = "UB" Then
    ArrayDemo4 = UBound(MyArray, Dimension)
  End If
End Function
```

	A	B	C	D
1	ARRAYDEMO4 IN ACTION			
2	LB	1	0	<-- =ArrayDemo4(B2,A2)
3	UB	1	2	<-- =ArrayDemo4(B3,A3)
4	LB	2	0	<-- =ArrayDemo4(B4,A4)
5	UB	2	3	<-- =ArrayDemo4(B5,A5)
6	LB	3	0	<-- =ArrayDemo4(B6,A6)
7	UB	3	4	<-- =ArrayDemo4(B7,A7)

How to Get the Bound of an Excel Range in a Function

Sadly the internal functions **UBound** and **LBound** do not work for a range passed to a function. We can make use of the fact that the parameter is actually a range and use some of its properties to get the result we need. The following function demonstrates this:

```
Function RangeBound(R, What)
    If What = "C" Then
        RangeBound = R.Columns.Count
    ElseIf What = "R" Then
        RangeBound = R.Rows.Count
    End If
End Function
```

	A	B	C
1	RANGEBOUND IN ACTION		
2	C	2	<-- =rangebound(D1:E5,A2)
3	R	5	<-- =rangebound(D2:E6,A3)
4	c	0	<-- =rangebound(D3:E7,A4)
5	r	0	<-- =rangebound(D4:E8,A5)

Did you notice that the function does not work with lowercase characters? If we want to be case agnostic, as one usually does, we can use the VBA function **UCase** to convert “what” to uppercase.

```
Function RangeBound1(R, What)
    If UCase(What) = "C" Then
        RangeBound1 = R.Columns.Count
    ElseIf UCase(What) = "R" Then
        RangeBound1 = R.Rows.Count
    End If
End Function
```

	A	B	C
1	<b>RANGEBOUND1 IN ACTION</b>		
2	C	2	<-- =rangebound1(D1:E5,A2)
3	R	5	<-- =rangebound1(D2:E6,A3)
4	c	2	<-- =rangebound1(D3:E7,A4)
5	r	5	<-- =rangebound1(D4:E8,A5)
6	1	0	<-- =rangebound1(D5:E9,A6)

### Fixing Excel's NPV Function

Recall from Chapter 1 that

Excel's language about discounted cash flows differs somewhat from the standard finance nomenclature. Excel uses the letters NPV to denote the present value (**not** the net present value) of a series of cash flows.

To calculate the finance net present value of a series of cash flows using Excel, we have to calculate the present value of the future cash flows (using the Excel NPV function) and subtract from this present value the time-zero cash flow. (This is often the cost of the asset.)

Let us try and write a function **nNPV** that addresses this shortcoming. In the process we shall learn a few things about Excel Ranges in VBA. In order to make the function simple, it will only work on a row of cash flows.

```
Function nNPV(Rate, R)
    nNPV = R(1) + Application.WorksheetFunction _
        .npv(Rate, R.Range("B1", R.End(xlToRight)))
End Function
```

**R.Range(CellTopLeft,CellBottomRight)** returns a range defined by its parameters. Note that the cell addresses are relative to **R** and not the worksheet.

**R.End(Direction)** returns one of the four possible last cells in the **R** according to **Direction**. Possible values for **Direction** are xlDown, xlToLeft, xlToRight, xlUp.

Assuming **R** is a row of cells, **R.Range(“B1”, R.End(xlToRight))** returns a range containing all the cells in **R** excluding the first one.

	A	B	C	D	E	F
1	<b>NNPV IN ACTION</b>					
2	Cash Flows ►	-400	100	100	100	100
3	Rate ►	10%				
4	Excel NPV ►	-75.4667776	<-- =NPV(\$B\$3,\$B\$2:\$F\$2)			
5	Excel C <sub>0</sub> +NPV ►	-83.01345537	<-- =B\$2+NPV(\$B\$3,\$C\$2:\$F\$2)			
6	nNPV ►	-83.01345537	<-- =nNPV(\$B\$3,\$B\$2:\$F\$2)			

### A New IRR Function

Another useful function we can write using our newly acquired tools is **nIRR**. Recall from Chapter 1 that the internal rate of return (IRR) is defined as the compound rate of return  $r$  that makes the NPV equal to zero:

$$CF_0 + \sum_{t=0}^N \frac{CF_t}{(1+r)^t} = 0$$

We now use a technique called successive refinement to calculate IRR:

1. If we were given an initial guess for  $r$  we use it and if not we use 50% to calculate NPV.
2. If the calculated NPV is zero (or sufficiently near) we return the current guess.
3. If the calculated NPV is negative we set our guess to  $r = r + r/2$ .
4. If the calculated NPV is positive we set our guess to  $r = r - r/2$ .
5. We shall now recalculate NPV.
6. Repeat steps 2–5.

We assume that the first cash flow is negative and that all others are positive.

Here is the function:

```
Function nIRR(R, Optional guess = 0.5)
  n = nNPV(guess, R)
  Do While Abs(npv) > 0.0001
    If n < 0 Then
      guess = guess - guess / 2
    Else
      guess = guess + guess / 2
    End If
    n = nNPV(guess, R)
  Loop
  nIRR = guess
End Function
```

### Optional Parameters

Note that the use of **Optional guess = 0.5** to declare the last parameter as optional and give it a default value if the user did not supply one. Once a parameter is declared as optional, all the following parameters have to be declared optional as well. For example, this declaration is fine:

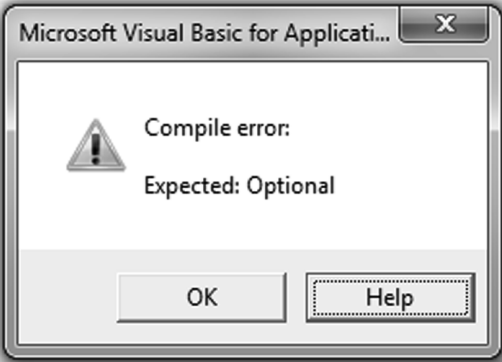
```
Function WillWork(a, Optional b = 5, Optional c = 4)
```



Whereas this will result in an error:

```
Function WillNotWork(a, Optional b = 5, c)
```

```
End Function
```



As noted, this function is very slow so it might take a few seconds to calculate its results.

	A	B	C	D	E	F
1	<b>NIRR IN ACTION</b>					
2	Cash Flows ►	-375	100	100	100	100
3	Guess ►	5%				
4	IRR ►	2.63247%	<-- =IRR(\$B\$2:\$F\$2,\$B\$3)			
5	nIRR ►	2.63247%	<-- =nIRR(\$B\$2:\$F\$2,\$B\$3)			
6	nIRR ►	2.63248%	<-- =nIRR(\$B\$2:\$F\$2)			
7	nNPV ►	7.708E-11	<-- =nNPV(\$B\$4,\$B\$2:\$F\$2)			
8	nNPV ►	9.795E-06	<-- =nNPV(\$B\$5,\$B\$2:\$F\$2)			
9	nNPV ►	-9.55E-05	<-- =nNPV(\$B\$6,\$B\$2:\$F\$2)			

The Option Base Statement

Excel arrays start at 1, whereas VBA arrays start at 0, unless otherwise defined. We can use a module option to make all not specifically declared array indices start at 1. We use **ArrayDemo3** to demonstrate. We open a new VBA module with first line “OptionBase1.” We rename our previous function to reflect this change.

```
Option Base 1
Function ArrayDemo3OptionBase1(N)
    Dim MyArray(5)
    If N = "LB" Then
        ArrayDemo3OptionBase1 = LBound(MyArray)
    ElseIf N = "UB" Then
        ArrayDemo3OptionBase1 = UBound(MyArray)
    End If
End Function
```

If we insert **Option Base 1** as the first line of the module to get (the only change is in cell B2 where we get 1 and not 0).

	A	B	C
1	ARRAYDEMO3OPTIONBASE1 IN ACTION		
2	LB	1	<-- =ArrayDemo3Optionbase1(A2)
3	UB	5	<-- =ArrayDemo3Optionbase1(A3)

The **Option Base 1** statement, like all option statements, should be inserted before all functions and subroutines in a module. Like all option statements, its effect is limited to all routines in the current module.

37.5 Multidimensional Arrays

Arrays can have more than one index. In a two-dimensional array the first index refers to the rows and the second to the columns. There is no formal limit to the number of indices you can declare in an array. The syntax for declaring a multidimensional array is demonstrated in the following functions:

```
Function Matrix1(R, C)
  Dim MyMat(2, 1)
  For i = 0 To 2
    For j = 0 To 1
      MyMat(i, j) = i * j
    Next j
  Next i
  If R >= 0 And R <= 2 And C >= 0 And C <= 1 _
Then
    Matrix1 = MyMat(R, C)
  End If
End Function
```

	A	B	C	D
1	MATRIX1 IN ACTION			
2	R	C	Matrix1(R,C)	
3	0	0	0	<-- =Matrix1(A3,B3)
4	1	0	0	<-- =Matrix1(A4,B4)
5	2	0	0	<-- =Matrix1(A5,B5)
6	0	1	0	<-- =Matrix1(A6,B6)
7	1	1	1	<-- =Matrix1(A7,B7)
8	2	1	2	<-- =Matrix1(A8,B8)
9	3	1	0	<-- =Matrix1(A9,B9)
10	1	3	0	<-- =Matrix1(A10,B10)

The following function demonstrates the use of **LBound** and **UBound** with multidimensional arrays:

```
Function Matrix2(R, C)
    Dim MyMat(1, 1)
    For i = LBound(MyMat, 1) To UBound(MyMat, 1)
        For j = LBound(MyMat, 2) To _
            UBound(MyMat, 2)
            MyMat(i, j) = i * j
        Next j
    Next i
    If R >= LBound(MyMat, 1) And _
        R <= UBound(MyMat, 1) And _
        C >= LBound(MyMat, 2) And _
        C <= UBound(MyMat, 2) Then
        Matrix2 = MyMat(R, C)
    End If
End Function
```

	A	B	C	D
1	<b>MATRIX2 IN ACTION</b>			
2	R	C	Matrix2(R,C)	
3	0	0	0 <-- =Matrix2(A3,B3)	
4	1	0	0 <-- =Matrix2(A4,B4)	
5	2	0	0 <-- =Matrix2(A5,B5)	
6	0	1	0 <-- =Matrix2(A6,B6)	
7	1	1	1 <-- =Matrix2(A7,B7)	
8	1	2	0 <-- =Matrix2(A8,B8)	

Note the use of the second argument to **LBound** and **UBound**. If used with only one argument, both functions return the largest index value the first dimension of the array can have; if the array has more than one dimension (as in this case), we can use a second argument to the function to specify the dimension we are interested in.

---

### 37.6 Dynamic Arrays and the ReDim Statement

Every so often it can be handy to have the size of an array set (and reset) when the program is running. Dynamic arrays are arrays that can have their size changed at run time. You declare dynamic arrays using the **Dim** statement but with nothing in the parentheses, as in:

```
Dim SomeName()
```

Before you can use the array you need to set its size using the **ReDim** statement, as in:

```
ReDim ArrayName(SomeIntegerExpression)
```

For example, you might type

```
ReDim Prices(12)
```

To set the size of the dynamic array **Prices** to 12 elements, a more typical case would involve the use of a variable for the size as in:

```
ReDim Prices(I)
```

This will set the size of **Prices** to the value of **I**.

The **ReDim** statement can also be used to change the size of a dynamic array (or indeed any VBA array). If you change the size of an array, all the data in the array are lost. Use **ReDim Preserve** to keep the old data, as in:

```
ReDim Preserve ArrayName (SomeIntegerExpression)
```

The following function calculates the present value of a series of future cash flows. To simplify the function, the interest rate is fixed at 5% per period. The function illustrates the use of a dynamic array (the variable **CF**) that derives its size from the size of the original input (the variable **n**):

```
Function DynPV(r As Range)
    ' n is number of periods
    ' cf() is dynamic array for cash
    ' flows
    Dim n
    Dim cf()
    Dim Temp
    Dim i
    'Below we distinguish if the data
    'is in a column or in a row
    If r.Columns.Count = 1 Then
        n = r.Rows.Count
    ElseIf r.Rows.Count = 1 Then
        n = r.Columns.Count
    Else
        Exit Function
    End If
    ' re-dimension the array
    ReDim cf(1 To n)
    For i = 1 To n
        cf(i) = r(i)
    Next i
    Temp = 0
    For i = 1 To n
        Temp = Temp + cf(i) / 1.05 ^ i
    Next i
    DynPV = Temp
End Function
```

Running the function produces the following:

	A	B	C
1	<b>DYNPV IN VERTICAL ACTION</b>		
2	Cash Flows		
3	100		
4	200		
5	300		
6	DynPV ►	535.79527	<-- =DynPV(A3:A5)

	A	B	C	D
1	<b>DYNPV IN HORIZONTAL ACTION</b>			
2	Cash Flows ►	100	200	300
3	DynPV ►	535.79527	<-- =DynPV(B2:D2)	

Using the ReDim Preserve Statement

As stated previously the **Preserve** part of the **ReDim** statement prevents the loss of data from the re-dimensioned array. The use of **Preserve** imposes two major limitations on the use of **ReDim**.

- The inability to change the lower boundary of the index.
- The inability to change the number of dimensions.

The main use of the **ReDim Preserve** is in interactive programs and as such it will be demonstrated in a later chapter dealing with user interaction.

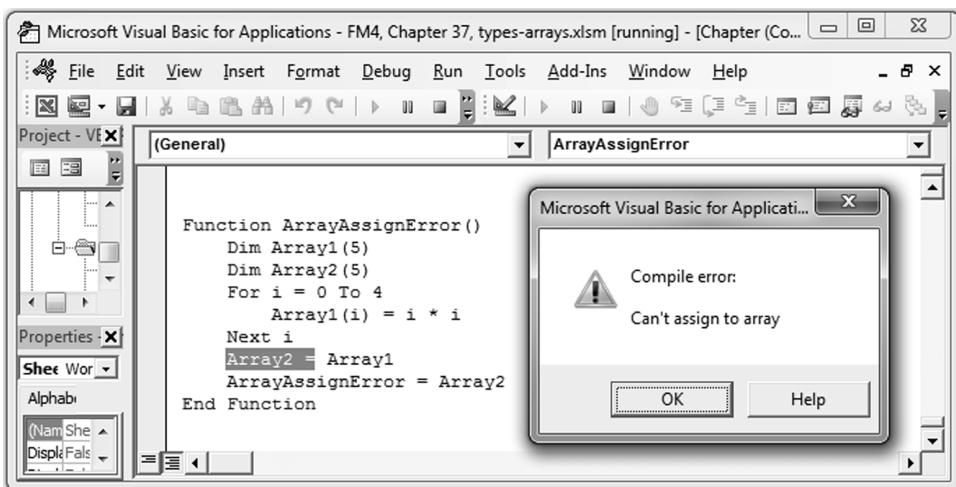
---

37.7 Array Assignment

Here’s an error that’s easy to make: In the following example we want to tell VBA that **Array2** is equal to **Array1**:

```
Function ArrayAssignError()  
    Dim Array1(5)  
    Dim Array2(5)  
    For i = 0 To 4  
        Array1(i) = i * i  
    Next i  
    Array2 = Array1  
    ArrayAssignError = Array2  
End Function
```

VBA doesn't allow this, as you can see on the next screen shot.



Obviously one way to assign arrays is to assign each element separately using a **For** loop.

```
For I = 0 To 4: Array2(I) = Array1(I):  
Next I
```



### The : Operator

Note the use of the “:” operator to signal the end of a statement. This way we can put two or more short statements on the same line. Another, much shorter, way of assigning arrays is discussed in the next section.

---

## 37.8 Variants Containing an Array

A **Variant** type variable can contain an array. The procedure is somewhat more complicated than the declaration of a normal array, but the reward in terms of assignment is sometimes worth the inconvenience. The following function demonstrates the use of a **Variant** containing an array:

```

01 Function ArrayAssign(r, j)
02     Dim Array1                                'This is a variant
03     Dim Array2                                'This is a variant
04     Dim n                                    'number of elements
                                                'in R
05     Array1 = Array()
06     If r.Columns.Count = 1 Then 'data in column
07         n = r.Rows.Count
08     ElseIf r.Rows.Count = 1 Then 'data in row
09         n = r.Columns.Count
10     Else                                     'invalid data
11         Exit Function
12     End If
13     ReDim Array1(1 To n)
14     For i = 1 To n
15         Array1(i) = r(i)
16     Next i
17     '*****Watch this spot
18     Array2 = Array1                          'Watch this spot
19     '*****Watch this spot
20     If j >= 1 And j <= n Then
21         ArrayAssign = Array2(j)
22     End If
23 End Function

```

The **Array()** function (on the fifth line) returns a **Variant** containing an array. The assignment on the same line makes **Array1** into an array (not initialized at the moment). The **ReDim** statement on line 15 makes **Array1** into an **n** element array. The reward for all our trouble is illustrated on line 18. Here is the function in a worksheet context:

	A	B	C	D	E
1	<b>ARRAYASSIGN IN HORIZONTAL ACTION</b>				
2	55	88	77	12	99
3	1	55	<-- =ArrayAssign(\$A\$2:\$E\$2,A3)		
4	2	88	<-- =ArrayAssign(\$A\$2:\$E\$2,A4)		
5	3	77	<-- =ArrayAssign(\$A\$2:\$E\$2,A5)		
6	4	12	<-- =ArrayAssign(\$A\$2:\$E\$2,A6)		
7	5	99	<-- =ArrayAssign(\$A\$2:\$E\$2,A7)		
8	6	0	<-- =ArrayAssign(\$A\$2:\$E\$2,A8)		

37.9 Arrays as Parameters to Functions

Arrays can be used as parameters to functions. The following set of functions presents an improved version of **DynPV** discussed in section 37.6. Notice how much easier it is to read the main function **NewDynPV**, when all the auxiliary tasks are relegated to separate functions.

A function **ComputePV(CF())** is used to compute the present value of a series of cash flows contained in an array of **Doubles**.

```
Function ComputePV(CF())
    Temp = 0
    For i = LBound(CF) To UBound(CF)
        Temp = Temp + CF(i) / 1.05 ^ i
    Next i
    ComputePV = Temp
End Function
```

Note the fact that in **ComputePV(CF())**, **CF()** has to be declared without index information. Consequently, we use **LBound** and **UBound** to get index information.

The function **GetN(R As Range)** returns the number of elements in R:

```
Function GetN(R As Range)
    If R.Columns.Count = 1 Then 'data in column
        GetN = R.Rows.Count
    ElseIf R.Rows.Count = 1 Then 'data in row
        GetN = R.Columns.Count
    Else
        GetN=0
    End If
End Function
```

Here is the main function:

```
Function NewDynPV(R As Range)
    Dim n As Integer ' Number of periods
    Dim CF() As Double ' Dynamic array for cash flows
    n = GetN(R)
    If (n=0) Then
        NewDynPV = n
        Exit Function
    End If
    ReDim CF(1 To n) ' re-dimension the array
    For i = 1 To n
        CF(i) = R(i)
    Next i
    NewDynPV = ComputePV(CF)
End Function
```

	A	B	C	D
1	<b>NEWDYNPV IN ACTION</b>			
2	<b>Cash Flows ►</b>	100	200	300
3	<b>NewDynPV ►</b>	535.79527	<-- =newDynPV(B2:D2)	

### Better IRR and NPV Functions

We can now revisit **nIRR** and **nNPV** from section 37.4, and try and make them faster using internal arrays.

```

Function fNPV(Rate, cf)
    Temp = 0
    For i = LBound(cf, 2) + 1 To UBound(cf, 2)
        Temp = Temp + cf(1, i) / (1 + Rate) ^ _
            (i - 1)
    Next i
    fNPV = Temp + cf(1, LBound(cf, 2))
End Function

Function fIRR(R, Optional guess = 0.5)
    cf = R.Value
    n = fNPV(guess, cf)
    Do While Abs(npv) > 0.0001
        If n < 0 Then
            guess = guess - guess / 2
        Else
            guess = guess + guess / 2
        End If
        n = fNPV(guess, cf)
    Loop
    fIRR = guess
End Function

```

	A	B	C	D	E	F
1	<b>FIRR IN ACTION</b>					
2	Cash Flows ►	-375	100	100	100	100
3	Guess ►	5%				
4	IRR ►	2.63247%	<-- =IRR(\$B\$2:\$F\$2,\$B\$3)			
5	fIRR ►	2.63247%	<-- =fIRR(\$B\$2:\$F\$2,\$B\$3)			
6	fIRR ►	2.63248%	<-- =fIRR(\$B\$2:\$F\$2)			
7	nNPV ►	7.70797E-11	<-- =nNPV(\$B\$4,\$B\$2:\$F\$2)			
8	nNPV ►	9.79546E-06	<-- =nNPV(\$B\$5,\$B\$2:\$F\$2)			
9	nNPV ►	-9.54928E-05	<-- =nNPV(\$B\$6,\$B\$2:\$F\$2)			

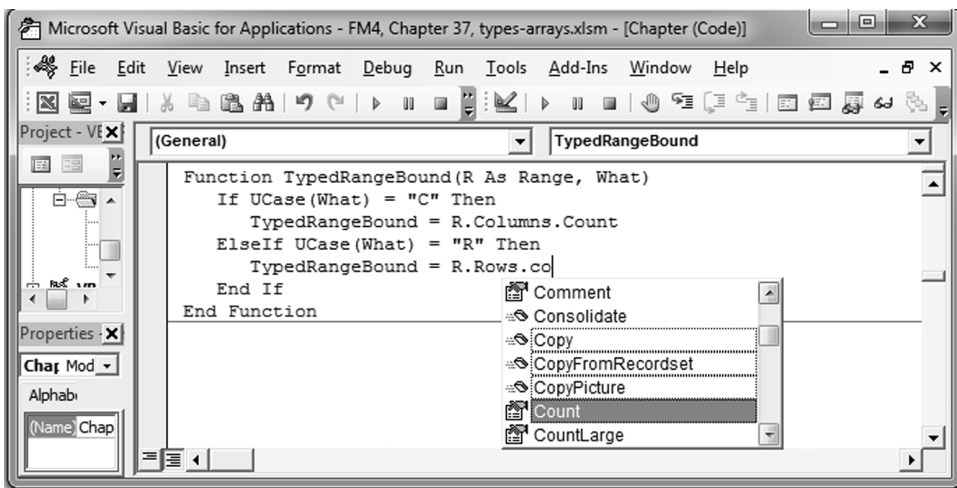
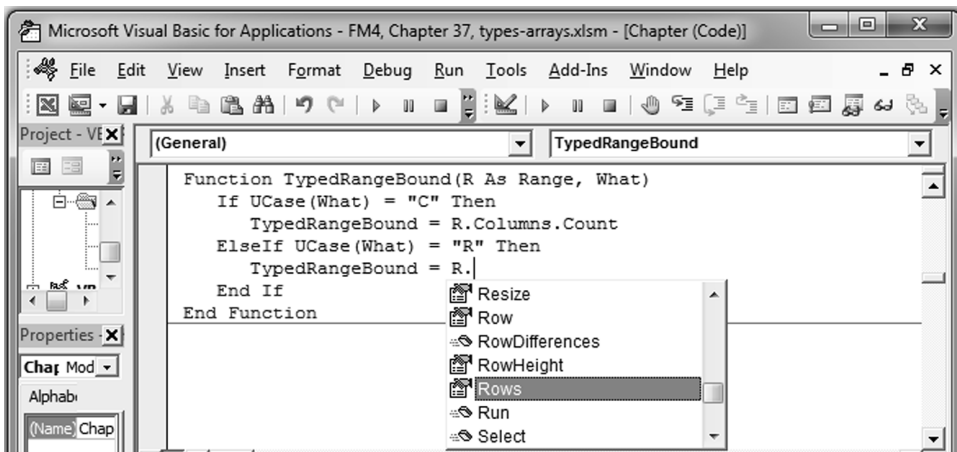
This works an order of magnitude faster but can be further improved.

---

### 37.10 Using Types

All values, variables, and functions in VBA are categorized into types, either by default or explicitly. By default all variables and functions in VBA are of the type **Variant**. **Variant** is a category of values (type) that includes all other categories. In most cases we can just ignore the type, but sometimes it can be very useful to give a variable a type other than **Variant**. Variable types allow VBA to give us information about the variable as we use it, and this will become apparent as we start using Excel objects in the following chapters. For now we will just explain the mechanics of defining a typed variable, and provide a short demonstration of the help offered by VBA when dealing with typed variables. A type is given to a variable when it is defined by following the variable's name with the word **As** followed with a type name. For example, the statement "Dim x As Integer" defines a variable named x of the type Integer.

To demonstrate the usefulness of typed variables recall the function **RangeBound** from section 37.4. Here is a new version with the first parameter explicitly given the type Range. Although not necessary (it worked without it), it does make life easier. When you type in the function and a variable has a type, VBA can give you hints as you try to use properties. In our example, as soon as you type the period after R, VBA provides a list of possible Properties or Methods. If the selected property (Rows, in our case) has properties of its own, then a period following its name will produce a list for you to choose from.



### 37.11 Summary

VBA functions use variables to store information. Variables can hold all sorts of information. Declaring and using variables that can hold only a specific type of information (**Typed Variables**) can make your programming task easier and your programs more readable, and use less computer memory.

An array is a group of variables of the same type, sharing the same name and referenced individually using one or more indices. In VBA an array index is an integer. By default the index of the first element in an array is 0; this can be changed to 1 for all arrays used in a module by using the **Option Base 1** statement. The size and number of dimensions of an array are set at the time the array is declared and have to be known when the program is written. Dynamic arrays are arrays whose size (but not number of dimensions) can be set at run time.

---

## Exercises

- Write a function **NewPV(CF, r)** which calculates the present value of a given cash flow **CF** at interest rate  $r$  for 5 periods:

$$NewPV(CF, r) = \frac{CF}{(1+r)^1} + \frac{CF}{(1+r)^2} + \frac{CF}{(1+r)^3} + \frac{CF}{(1+r)^4} + \frac{CF}{(1+r)^5}$$

	A	B	C	D
1	<b>NEWPV IN ACTION</b>			
2	<b>CF</b>	<b>r</b>	<b>NewPV</b>	
3	100.0000	10%	379.0787	<-- =NewPV(A3,B3)
4	50.0000	10%	189.5393	<-- =NewPV(A4,B4)
5	100.0000	1%	485.3431	<-- =NewPV(A5,B5)
6	50.0000	1%	242.6716	<-- =NewPV(A6,B6)

- Rewrite the function in exercise 2 as **BetterNewPV(CF, r, n)**, so it could deal with  $n$  periods.

	A	B	C	D	E
1	<b>BETTERNEWPV IN ACTION</b>				
2	<b>CF</b>	<b>r</b>	<b>n</b>	<b>BetterNewPV</b>	
3	100.0000	5%	5	432.9477	<-- =BetterNewPV(A3,B3,C3)
4	50.0000	10%	5	189.5393	<-- =BetterNewPV(A4,B4,C4)
5	100.0000	1%	10	947.1305	<-- =BetterNewPV(A5,B5,C5)
6	50.0000	1%	10	473.5652	<-- =BetterNewPV(A6,B6,C6)

3. A bank offers different interest rates on loans. The rate is based on the size of the periodical repayment (**CF**) and the following table. Rewrite the function in exercise 2 as **BankPV(CF, r, n)** so that it reflects the present value of a loan in the bank.

For Periodical Repayments <=	The Interest Rate Is
100.00	$r$
500.00	$r - 0.5\%$
1,000.00	$r - 1.1\%$
5,000.00	$r - 1.7\%$
1,000,000.00	$r - 2.1\%$

	A	B	C	D	E
1	<b>BANKPV IN ACTION</b>				
2	<b>CF</b>	<b>r</b>	<b>n</b>	<b>BankPV</b>	
3	-1	5%	5	E	<-- =BankPV(A3,B3,C3)
4	100.00	5%	5	432.95	<-- =BankPV(A4,B4,C4)
5	100.01	5%	5	439.04	<-- =BankPV(A5,B5,C5)
6	1000.00	5%	5	4464.36	<-- =BankPV(A6,B6,C6)
7	1000.01	5%	5	4540.79	<-- =BankPV(A7,B7,C7)
8	5000.00	5%	5	22703.71	<-- =BankPV(A8,B8,C8)
9	5000.01	5%	5	22964.11	<-- =BankPV(A9,B9,C9)

4. A bank offers different interest rates on deposit accounts. The rate is based on the size of the periodical deposit (**CF**) and the following table. Write a future value function **BankFV(CF, r, n)**.

For Periodical Deposits	The Interest Rate Is
<=100.00	$r$
<=500.00	$r + 0.5\%$
<=1,000.00	$r + 1.1\%$
<=5,000.00	$r + 1.7\%$
>5,000.00	$r + 2.1\%$



	A	B	C	D	E
1	<b>BANKFV IN ACTION</b>				
2	<b>CF</b>	<b>r</b>	<b>n</b>	<b>BankFV</b>	
3	-1	5%	5	E	<-- =Bankfv(A3,B3,C3)
4	100.00	5%	5	580.19	<-- =Bankfv(A4,B4,C4)
5	100.01	5%	5	588.86	<-- =Bankfv(A5,B5,C5)
6	1,000.00	5%	5	5992.91	<-- =Bankfv(A6,B6,C6)
7	1,000.01	5%	5	6099.47	<-- =Bankfv(A7,B7,C7)
8	5,000.00	5%	5	30497.07	<-- =Bankfv(A8,B8,C8)
9	5,000.01	5%	5	30856.78	<-- =Bankfv(A9,B9,C9)

5. Another bank offers 1% increase in interest rate to savings accounts with a balance of more than 10,000.00. Write a future value function **Bank1FV(CF, r, n)** that reflects this policy.

	A	B	C	D	E
1	<b>BANK1FV IN ACTION</b>				
2	<b>CF</b>	<b>r</b>	<b>n</b>	<b>Bank1FV</b>	
3	-1	5%	5	E	<-- =Bank1FV(A3,B3,C3)
4	9999.00	5%	5	59620.97	<-- =Bank1FV(A4,B4,C4)
5	10000.00	5%	5	59626.94	<-- =Bank1FV(A5,B5,C5)
6	10001.00	5%	5	59759.16	<-- =Bank1FV(A6,B6,C6)
7				5.96	<-- =D5-D4
8				132.22	<-- =D6-D5

6. The bank in exercise 5 changed its bonus policy and now offers the interest rate increase based on the following table. Rewrite **Bank1FV(CF, r, n)** to reflect this change.

Balance	Interest Rate
<=1,000.00	$r + 0.2\%$
<=5,000.00	$r + 0.5\%$
<=10,000.00	$r + 1.0\%$
>10,000.00	$r + 1.3\%$

	A	B	C	D	E
1	<b>BANK2FV IN ACTION</b>				
2	<b>CF</b>	<b>r</b>	<b>n</b>	<b>Bank2FV</b>	
3	-1	5%	5	E	<-- =Bank2FV(A3,B3,C3)
4	9999.00	5%	5	60237.93	<-- =Bank2FV(A4,B4,C4)
5	10000.00	5%	5	60243.96	<-- =Bank2FV(A5,B5,C5)
6	10001.00	5%	5	60288.29	<-- =Bank2FV(A6,B6,C6)

7. Write a version of the present value function with two interest rates, one for positive cash flows and another for negative cash flows. The function should be written for use in a worksheet, and accept both column and row ranges as parameters. The function declaration line should be:

Function MyPV(CF As Variant, PositiveR As Double, \_  
NegativeR As Double) As Double

	A	B	C	D	E	F	G
1	<b>MYPV IN ACTION</b>						
2	PositiveR	5%	100	100	100	272.3248	<-- =MyPV(C2:E2,\$B\$2,\$B\$3)
3	NegativeR	10%	-100	-100	-100	-248.6852	<-- =MyPV(C3:E3,\$B\$2,\$B\$3)
4			-100	100	100	86.17762	<-- =MyPV(C4:E4,\$B\$2,\$B\$3)
5			-63	<-- =MyPV(C2:C4,\$B\$2,\$B\$3)			

8. Write a future value version of the function in exercise 7.
9. A bank offers different interest rates on loans. The rate is based on the size of the periodical repayment (**CF<sub>i</sub>**) and the following table. Write a present value function **BankPV(CF, r)** so that it reflects the present value of a loan in the bank. The function should be useable as a worksheet function. **CF** could be either a row range or a column range.

For Periodical Repayments <=	The Interest Rate Is
100.00	$r$
500.00	$r - 0.5\%$
1,000.00	$r - 1.1\%$
5,000.00	$r - 1.7\%$
1,000,000.00	$r - 2.1\%$

10. A bank offers different interest rates on deposit accounts. The rate is based on the size of the periodical deposit (**CF<sub>i</sub>**) and the following table. Write a future value function **BankFV(CF, r)**. The function should be useable as a worksheet function. **CF** could be either a row range or a column range.

For Periodical Deposits	The Interest Rate Is
$\leq 100.00$	$r$
$\leq 500.00$	$r + 0.5\%$
$\leq 1,000.00$	$r + 1.1\%$
$\leq 5,000.00$	$r + 1.7\%$
$> 5,000.00$	$r + 2.1\%$

11. Another bank offers 1% increase in interest rate to savings accounts with a balance of more than 10,000.00. Write a future value function **Bank1FV(CF, r)** that reflects this policy. The function should be useable as a worksheet function. **CF** could be either a row range or a column range.



---

# 38 Subroutines and User Interaction

---

## 38.1 Overview

A subroutine is a VBA user routine used to automate routine or repetitive operations in Excel. Subroutines are sometimes called macros. Modules and module variables are introduced as the last subject of this chapter.

---

## 38.2 Subroutines

A subroutine looks like a function, but the word “Sub” replaces the word **Function** in the definition. The parentheses following the subroutine name are blank (recall that the parentheses following the function name give the function’s parameters). Separating the first and last line are the statements that the subroutine executes. The following is a very simple subroutine that puts a message on the screen:

```
Sub SayHi()  
    MsgBox "Hi", , "I say Hi"  
End Sub
```

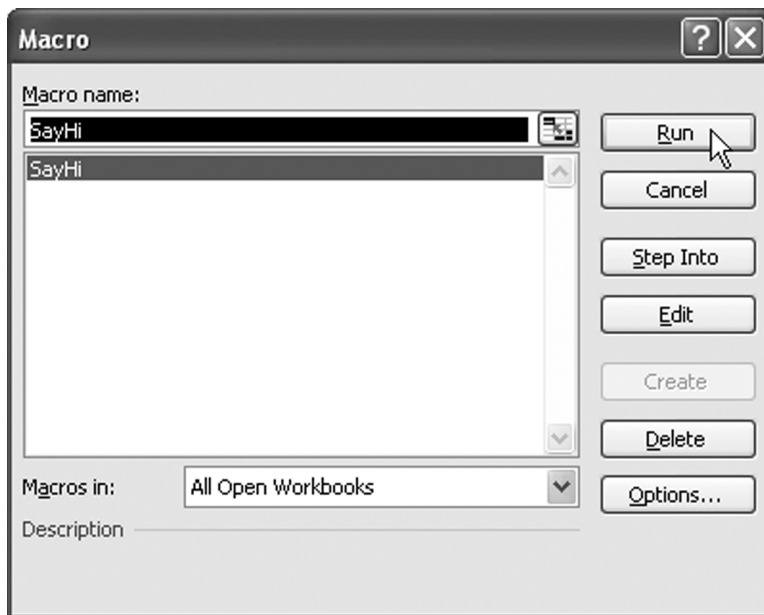
The above subroutine introduces a built-in VBA subroutine called **MsgBox**. It also introduces the way one subroutine is activated (called) from another. **MsgBox** is named as a command on a line followed by its list of arguments separated by commas. Notice the syntax:

```
MsgBox "Hi", , "I say Hi"
```

The commas separate the three arguments of every **MsgBox**:

- “Hi” is the message which will be displayed.
- The second argument is empty: Notice the space between the commas. This argument can be used to define buttons for the message box. This topic is discussed in section 38.3.
- The third argument is “I say Hi”—this is the message box title.

A subroutine can be activated (run) from an Excel worksheet in various ways. The simplest way of running a subroutine is from the **Macros** button on the **Developer** tab on the **Ribbon** or by using the keyboard shortcut [Alt] + F8. Either way, the macro selection box appears.<sup>1</sup> The box lists all available subroutines alphabetically. Find our subroutine, click on its name, and click the **Run** button.



And this is what you will see:



1. If you don't have the **Developer** tab, go to **File|Options|Customize the Ribbon**. In the right side of the resulting screen, mark the **Developer** box.

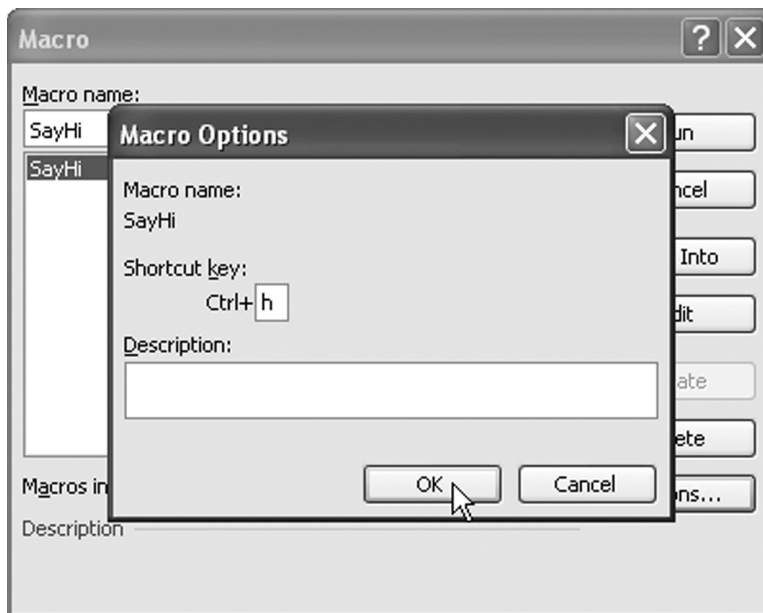
At this point Excel is locked up—you have to click the **OK** button before you can proceed.

### Keyboard Shortcut for Subroutines

Using a keyboard shortcut is a faster way to make a subroutine run. To attach a shortcut to our subroutine:

- Select the **Options** button from the macro selection box.
- Type a character in the provided space and click **OK**.
- Close the macro selection box using the corner **X**.

You can now activate the subroutine using the shortcut ([Ctrl] + h, in our case).



## Recording a Subroutine

One easy way to start writing subroutines is to record the sequence of actions you want in the subroutine, and then edit the resulting subroutine to produce the final results you need. Something we do a lot in this book is to insert the function **Getformula** in a cell to the right of the cell with the interesting formula. Let's record a subroutine that performs this action (see Figures 38.1–38.3).

1. Select the cell to the right of the cell with the formula we are interested in (B4, in our case)
2. Select the **Developer** tab on the ribbon.
3. In the code group click **Use Relative References**. This causes Excel to record relative cell addresses in the subroutine rather than its default, which is to record the actual cell addresses.
4. Click **Record Macro**.
5. At this point you have the option to name the subroutine as well as all sorts of other options. Since we are going to change most of these options in the near future, we ignore all options and finally click **OK** to start recording.
6. Type in the formula **=Getformula(A4)**.
7. Click **Stop Recording** (this is the same button that used to be the **Record Macro** button).



RECORDING A SUBROUTINE—SCREENS

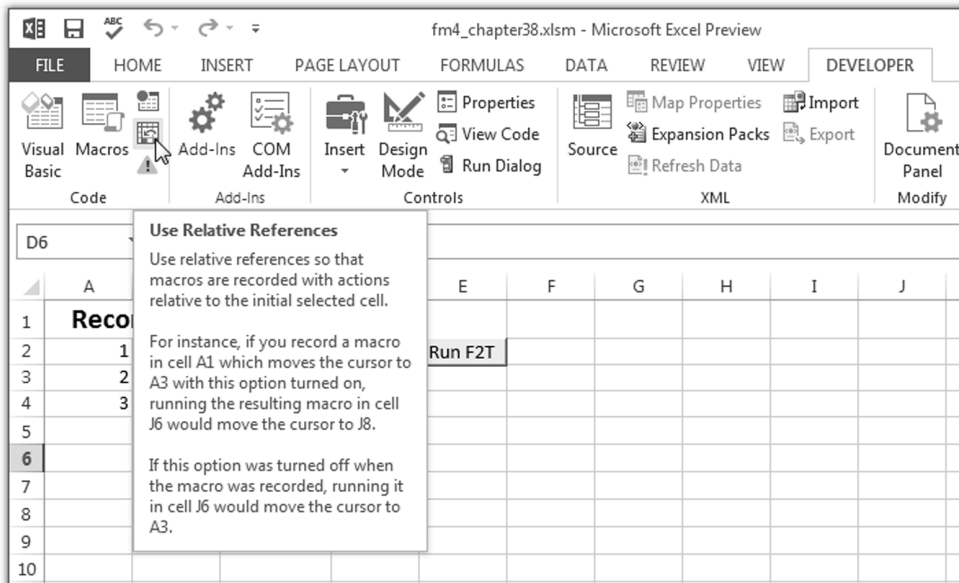


Figure 38.1  
Indicating recording with relative references.

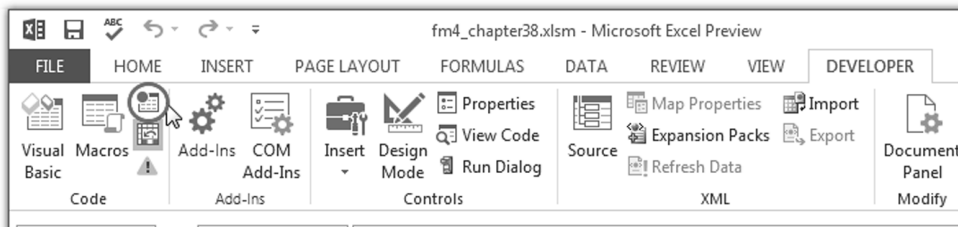


Figure 38.2  
Start recording the subroutine.

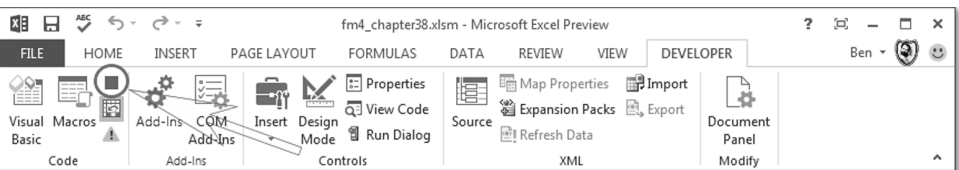


Figure 38.3  
End subroutine recording. Button is in the same place as the **Record Macro** button.

If we now go to the VBA editor we can see that a new module has been added to the workbook. The module contains **Macro1**:

```
Sub Macro1()  
    ActiveCell.FormulaR1C1 = _  
        "=getformula(RC[-1])"  
End Sub
```

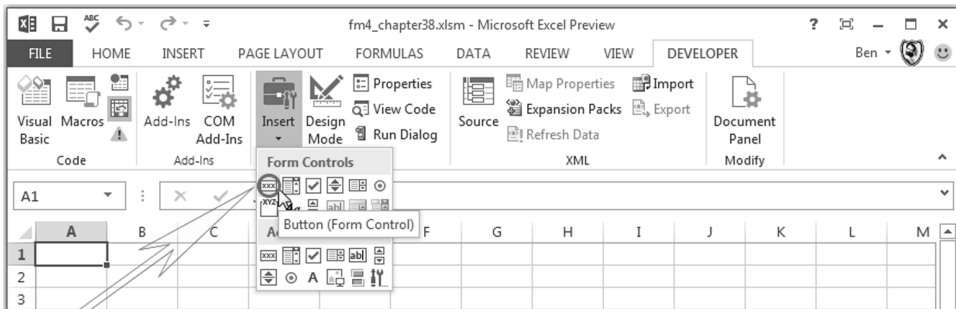
We can add some bells and whistles to this recorded subroutine. In the recorded subroutine below, we changed the name and put in a line to prevent the accidental overwriting of a non-blank cell.

```
Sub RecordGetformula()  
    ' Puts in Getformula, points to cell  
    ' to the left  
    If IsEmpty(ActiveCell) Then  
        ActiveCell.FormulaR1C1 = _  
            "=getformula(RC[-1])"  
    End If  
End Sub
```

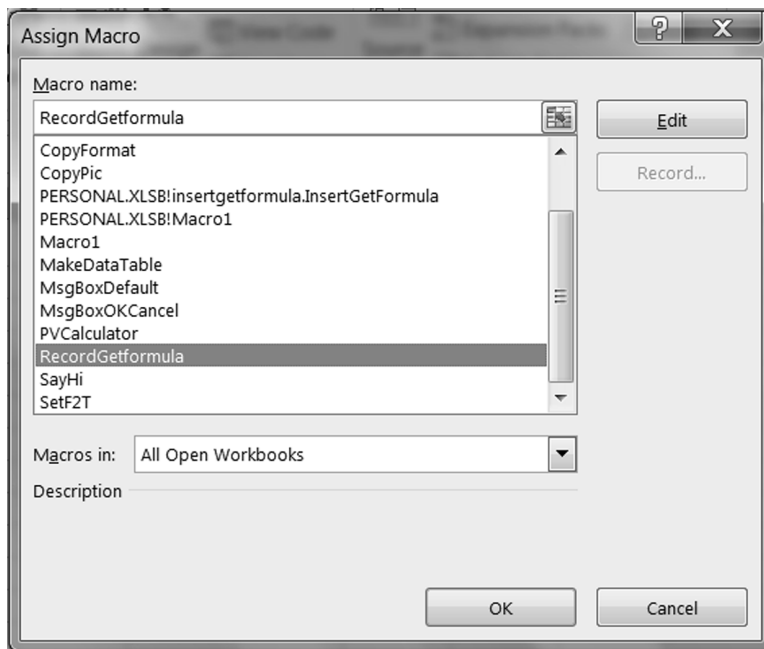
### Running a Subroutine from a Button on the Worksheet

Instead of running the subroutine from the **Developer** tab or running it from a key combination, we can also insert a button on the worksheet to run a subroutine on that worksheet. To illustrate, we insert a button that runs the subroutine **RecordGetformula**:

1. Select the **Developer** tab on the ribbon.
2. In the **Controls** group, click **Insert**.
3. From the **Form Controls**, select **Button**.



4. Drag the crosshair on the sheet to draw the button.
5. Once you release the mouse button, the **Assign Macro** dialogue will appear.



6. Select our subroutine and click **OK**.

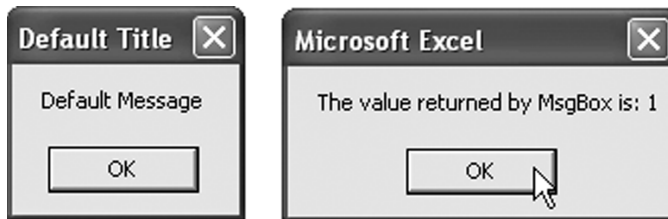
7. The button should be selected (control handles all round) and you can now edit the text that appears on the button. If the button is not selected, right-clicking on it will open a local menu enabling you to change the text or the subroutine assigned to the button.

	A	B	C	D	E
1	RECORDING IN ACTION				
2	1				
3	2			Apply Getformula	
4	3	<-- =A2+A3			
5					

38.3 User Interaction

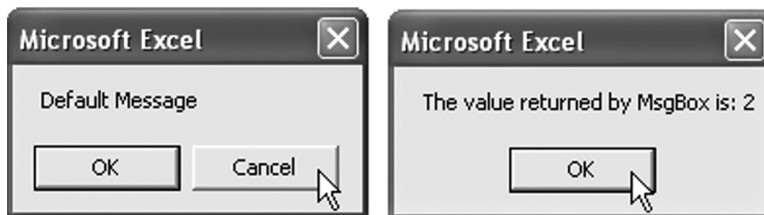
In this section we show how to use subroutines to elicit data from the user of the spreadsheet. We illustrate with the **MsgBox** command, which (as discussed above) displays a message on the screen and returns a value based on the button clicked. Some of the different options available with this function are demonstrated in the following subroutines:

```
Sub MsgBoxDefault()  
    Dim Temp As Integer  
    Temp = MsgBox("Default Message", , _  
        "Default Title")  
    MsgBox _  
        "The value returned by MsgBox is: " & Temp  
End Sub
```



*Note:* The default configuration of **MsgBox** produces one **OK** button. The default title is “Microsoft Excel.” Clicking the **OK** button makes **MsgBox** return the value 1.

```
Sub MsgBoxOKCancel()  
    Dim Temp As Integer  
    Temp = MsgBox("Default Message", _  
        vbOKCancel)  
    MsgBox _  
        "The value returned by MsgBox is: " _  
        & Temp  
End Sub
```



As previously noted, the second argument to **MsgBox** determines which buttons are displayed. This incarnation of the demo subroutine uses the constant **vbOKCancel** to produce the two buttons **OK** and **Cancel**. Note that if the **Cancel** button is clicked, **MsgBox** returns the value 2.

**InputBox: Getting Data from the User**

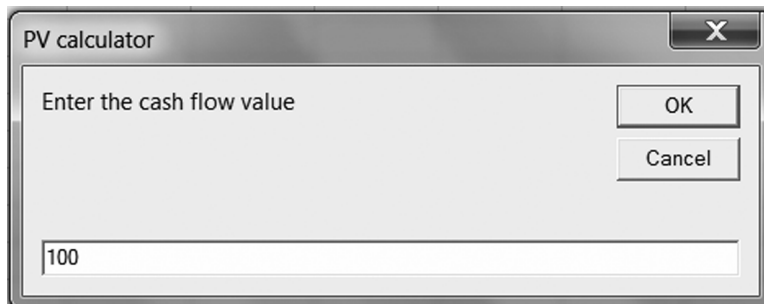
**InputBox** is an internal VBA function used to get textual information from the user into a variable in a subroutine. The workings of the function are demonstrated in the following present value calculator subroutine. The subroutine **PVCalculator** calculates  $\sum_{t=1}^{10} \frac{CF}{(1.05)^t}$ , where *CF* is a number inputted by the user:

```
Sub PVCalculator()  
    Dim CF  
    CF = InputBox("Enter the cash flow value", _  
        "PV calculator", "100")  
    MsgBox "The present value of 4" & CF & _  
        "At 5% for 10 periods is: " & _  
        Round(Application.PV(0.05, 10, -CF), _  
            2), vbInformation, "PV calculator"  
End Sub
```

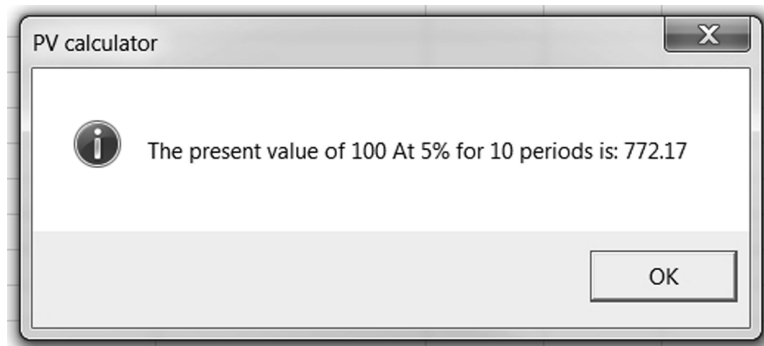
Note the syntax:

```
CF = InputBox("Enter the cash flow value", _  
    "PV calculator", "100")
```

- “Enter ... please,” the first argument in **InputBox**, is the message to display.
- “PV calculator,” the second argument, is the title for the box.
- “100,” the third argument, is the default string to place in the box. If you do not replace this by some other value, this will also be the returned value from the function.
- Running the subroutine should result in the following:



At this point you can replace “100” by some other number (in this example, we’ve chosen to leave it). Clicking on the **OK** button results in the following box:






---

### 38.4 Using Subroutines to Change the Excel Workbook




Subroutines can be used to make changes to a spreadsheet. Here’s a small example, very similar to the example presented at the end of Chapter 35. In this version of the subroutine we change the current region’s format to numbers with comma separators and without decimals.

```
Sub Format()  
    ActiveCell.CurrentRegion.NumberFormat _  
        = "#,##0"  
End Sub
```

**ActiveCell.CurrentRegion** is the range around the active cell (B5 in the screen snaps), the same range that would be selected by pressing [Ctrl] + A in the worksheet (A3:C7 in the screen shots).

B5		:	  	13560.8494419882	
	A	B	C	D	E
1	FORMAT IN ACTION			Run Format	
2					
3	81232.02236	71433.41596	43136.94352		
4	40486.60055	7707.299728	49201.05641		
5	44493.08534	13560.84944	68840.9526		123.123
6	19958.58414	28129.13586	28730.06792		
7	58843.8395	69566.7708	53717.7971		
8					
9		456.456			
10					

After:

B5		:	  	13560.8494419882	
	A	B	C	D	E
1	FORMAT IN ACTION			Run Format	
2					
3	81,232	71,433	43,137		
4	40,487	7,707	49,201		
5	44,493	13,561	68,841		123.123
6	19,959	28,129	28,730		
7	58,844	69,567	53,718		
8					
9		456.456			
10					



The next subroutine changes the actual data in **ActiveCell.CurrentRegion** to thousands by dividing each number in the range by 1,000 and rounding it to the nearest integer.

```
Sub ConvertToThousands()  
    s = ActiveCell.CurrentRegion.Cells.Count  
    For i = 1 To s  
        ActiveCell.CurrentRegion(i).Value = _  
            Round(ActiveCell.CurrentRegion(i). _  
                Value / 1000, 0)  
    Next i  
End Sub
```

B5	:				123456.789
	A	B	C	D	E
1	CONVERT TO THOUSANDS IN ACTION			Run Sub	
2					
3	812232.0224	71433.41596	43136.94352		
4	404586.6005	7707.299728	498201.0564		
5	444993.0853	123456.789	68840.9526		123.123
6	19958.58414	28129.13586	288730.0679		
7	58843.8395	69566.7708	538717.7971		
8					
9		456.456			
10					

After:

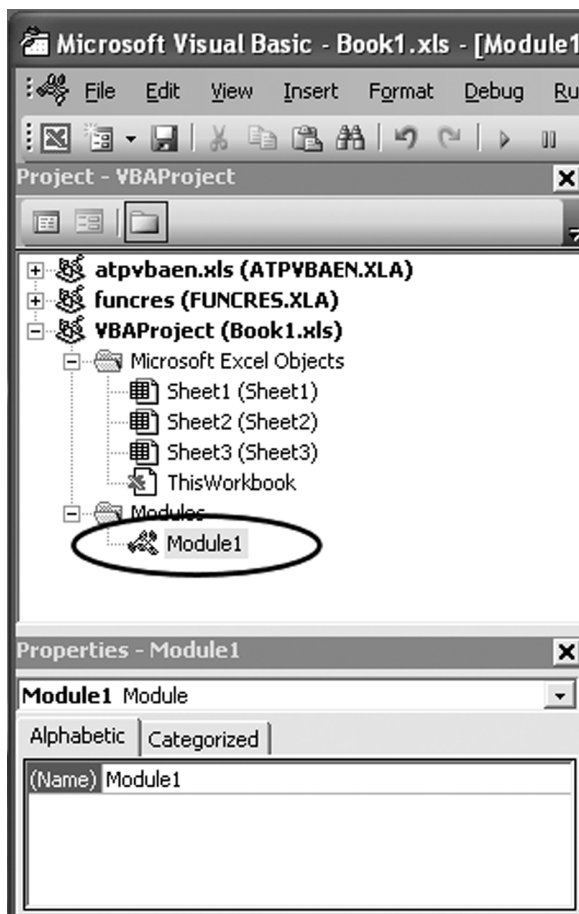
B5	:				123
	A	B	C	D	E
1	CONVERT TO THOUSANDS IN ACTION			Run Sub	
2					
3	812	71	43		
4	405	8	498		
5	445	123	69		123.123
6	20	28	289		
7	59	70	539		
8					
9		456.456			
10					

---

## 38.5 Modules

VBA organizes user-defined functions and subroutines in units called modules. We can (and sometimes should) have more than one module in a VBA project (i.e., the part of the workbook that has our functions and subroutines). Modules have names: By default VBA uses the name “Module” followed by a number to indicate the module’s name, but you might find it useful to give them a somewhat more descriptive name.

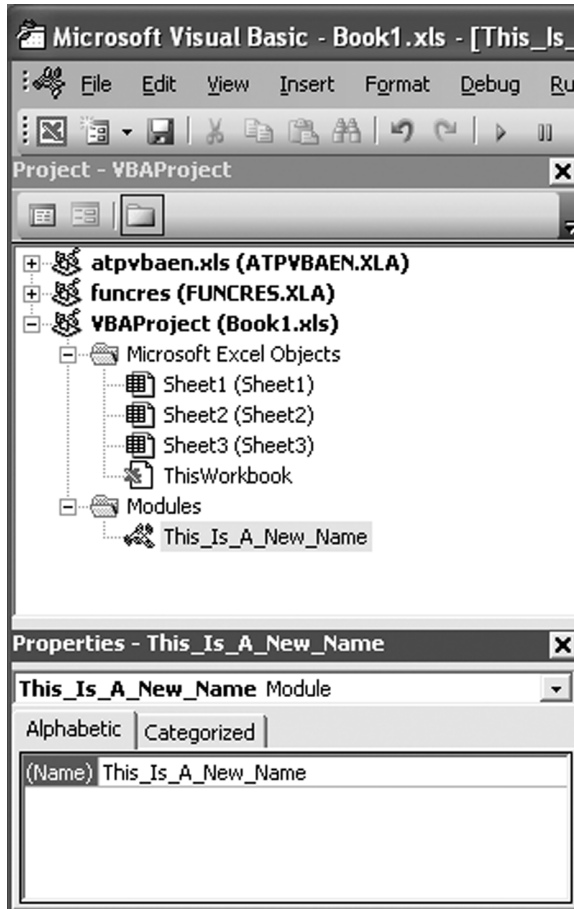
To rename a module (in the VBA Editor), select the module on the **Project Explorer** pane:



If the **Project Explorer** pane is not visible, select **Project Explorer** from the **View** menu.

Once a module is selected, the module's list of properties should appear in the **Properties Pane**. If the **Properties Pane** is not visible, select **Properties Window** from the **View** menu. Click on the module's name (it should be the only property available) and change it. A module name should start with an alphabetic character and consist of only alphabetic characters, digits, and the underscore character (\_); no other characters should be used.

Once you tap the Enter key, the name is changed. Notice the change in the Project Explorer.



Modules must have unique names, and they cannot be named after subroutines and functions. If a module called Tom has a function called Tom in it, the function Tom will not be available to the workbook. One common practice is to start module names (and only module names) with M.

### Module Variables

The **Dim** statement can be used before any routine in the module to define a module variable. Module variables are recognized anywhere in the module and keep their value until the workbook is closed. Module variables can be used to store information relevant to more than one routine without the need to pass the information via the parameters. Module variables are more commonly used in large modules with many interacting routines, so the following demonstration is, of necessity, somewhat trivial:

```
Dim MyStatus
Sub SetMyStatus()
    MyStatus = InputBox _
        ("Enter value for my status", , "OK")
    Calculate
End Sub
Function MyStatusIs()
    MyStatusIs = MyStatus
End Function
Sub ShowMyStatus()
    MsgBox "MyStatus is: " & MyStatus
End Sub
Function MyStatusIsVolatile()
    Application.Volatile
    MyStatusIsVolatile = MyStatus
End Function
```

When you first open the workbook, here is what you see:

	A	B	C	D
1	MODULE VARIABLES IN ACTION		Set MyStatus	
2				
3		0 <-- =MyStatusIs()	Show MyStatus	
4		0 <-- =MyStatusIsVolatile()		
5				

If you click on **Set MyStatus** you get the **Input Box**:

	A	B	C	D
1	MODULE VARIABLES IN ACTION		Set MyStatus	
2				
3		0 <-- =MyStatusIs()	Show MyStatus	
4		0 <-- =MyStatusIsVolatile()		
5				
6				
7				
8				
9				
10				
11				
12				
13				

Microsoft Excel

Enter value for my status

OK

Cancel

And a click on the OK button will produce the following:

	A	B	C	D
1	MODULE VARIABLES IN ACTION		Set MyStatus	
2				
3		0 <-- =MyStatusIs()	Show MyStatus	
4	OK	<-- =MyStatusIsVolatile()		

We now know that the variable **MyStatus** has the value “OK.” So why is the function **MyStatusIs** returning a zero, or for that matter, why is **MyStatusIsVolatile** returning the (correct) value of “OK”?

Application.Volatile

The answer to the question above lies with the **Application.Volatile** statement in **MyStatusIsVolatile**. When **Application.Volatile** is used as the first statement in a function used in a worksheet, the function gets recalculated whenever something gets recalculated on the worksheet. **MyStatusIs** will only recalculate if its (nonexisting) parameter changes, in this case only if we edit the cell and press Enter. So if we if we select cell A3, press F2 (for Edit Cell), and press Enter we get

MODULE VARIABLES IN ACTION		Set MyStatus
OK	<-- =MyStatusIs()	Show MyStatus
OK	<-- =MyStatusIsVolatile()	

38.6 Summary

A subroutine is a VBA user routine used to automate routine or repetitive operations in Excel. VBA provides two important and very flexible functions for user interaction: **MsgBox** and **InputBox**. VBA groups subroutines and functions into units called modules; keeping related functions and subroutines grouped is useful when dealing with large projects. All of these topics, explored in this chapter, will help you with financial programming in Excel.

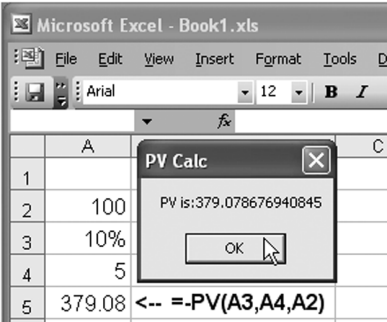
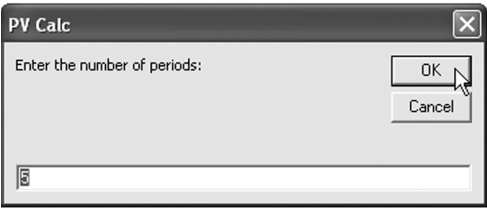
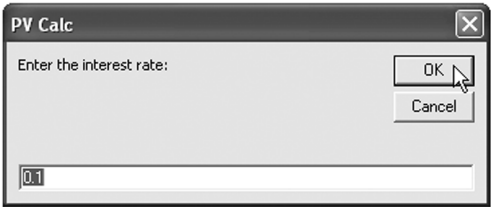
Exercises

1. Write a subroutine that displays the following message box. The message box should be on top of all other windows, and prevent the user from doing anything in any application, until one of the buttons is clicked.
- Hint: You need to use some options of **MsgBox** that were not covered in the text, use the VBA help system.



2. Write a present value calculator subroutine similar to the one which appears in section 38.4. However—as illustrated below—your subroutine should ask the user for the cash flow value, the interest rate, and the number of periods. It should then display the result in a message box. Sensible default values should be supplied for all arguments. Do not use the Excel function **PV**; write your own present value function and use it. A reminder:

$$PV(CF, r, n) = \sum_{i=1}^n \frac{CF}{(1+r)^i}$$

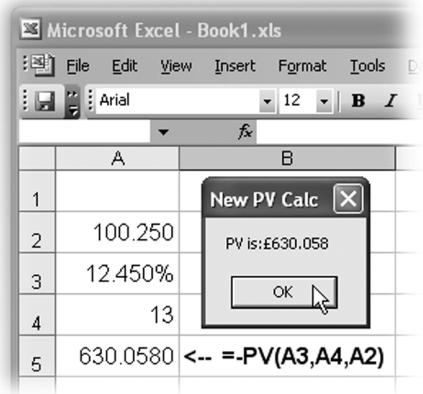
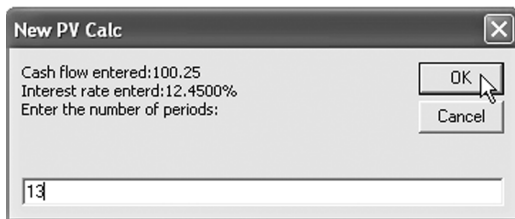
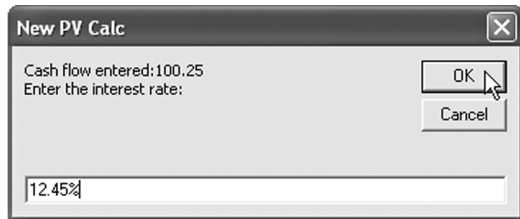


You can use the **PV** function provided by Excel, as we did, to verify the correctness of your subroutine.

3. Rewrite the subroutine in the previous exercise so that the user interface is as demonstrated in the following screen shots. Some of the functions needed to write the subroutine were not covered in the text. We used the following functions:

- **Val**—A function used to convert a string of digits to a number.
- **Left**—A function used to return the left part of a string.
- **Right**—A function used to return the right part of a string.
- **FormatPercent**—A function used to format a number.
- **FormatCurrency**—A function used to format a number.

More information about these functions is available from the VBA Help file. We recommend you use it.



*Note:* Your computer might display a different currency symbol.

4. Rewrite the subroutine in the previous exercise so it deals properly with the **Cancel** button.
- A simple version of the new subroutine will abort the subroutine if **Cancel** is clicked in any stage.
  - A more sophisticated version of the new subroutine will allow the user to reenter the data from scratch.
  - The most sophisticated version of the new subroutine will allow reentering the data using the old data as a default.

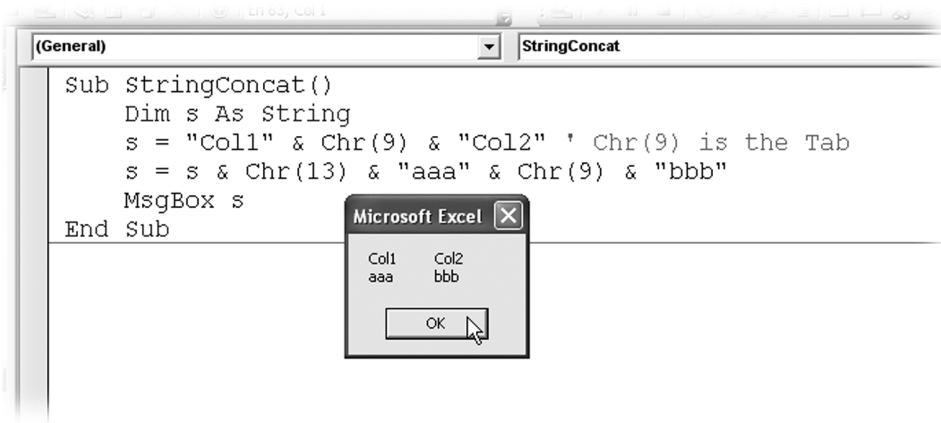
*Note:* The last version is a slightly more complicated exercise using loops within loops.



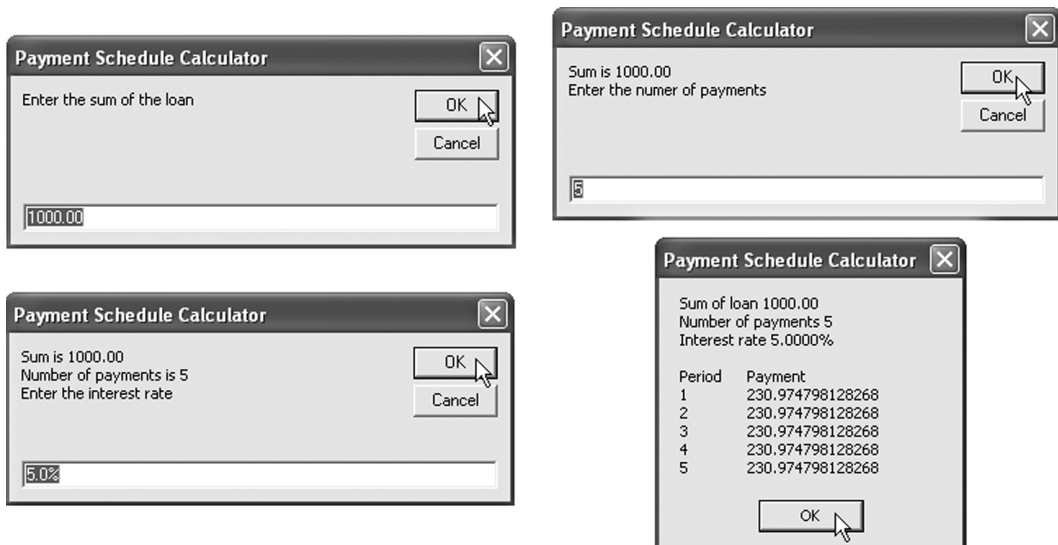
5. Write a payment schedule calculator subroutine. The subroutine is to ask the user for the sum of the loan, the number of payments, and the interest rate. Assume payment at the end of the period. The output should look like the example below.

*Hints:*

- You may want to use the worksheet function **PMT**.
- The following subroutine and its output might be of interest:



Here is an example of the requested subroutine in action:



6. Rewrite the payment schedule calculator subroutine so it displays the payments broken down into interest and capital payments. The input boxes in the example were removed for compactness.

**Payment Schedule Calculator** [X]

Sum of loan 999.99  
 Number of payments 5  
 Interest rate 5.0000%

Period	Balance	Payment	Interest	Capital
1	£999.99	£230.97	£50.00	£180.97
2	£819.02	£230.97	£40.95	£190.02
3	£629.00	£230.97	£31.45	£199.52
4	£429.47	£230.97	£21.47	£209.50
5	£219.97	£230.97	£11.00	£219.97

OK

7. Write a payment schedule calculator subroutine. The subroutine is to ask the user for the sum of the loan, the payment, and the interest rate. Assume payment at the end of the period. The subroutine should display the payments broken down into interest and capital payments. Obviously, the last payment can be smaller (but not larger) than the payment supplied by the user. The output should look like the example below (input boxes removed for compactness):

**Payment Schedule Calculator** [X]

Sum of loan £999.99    Payment £239.99    Rate 10.0000%

Period	Balance	Payment	Interest	Capital
1	£999.99	£239.99	£100.00	£139.99
2	£860.00	£239.99	£86.00	£153.99
3	£706.01	£239.99	£70.60	£169.39
4	£536.62	£239.99	£53.66	£186.33
5	£350.29	£239.99	£35.03	£204.96
6	£145.33	£159.86	£14.53	£145.33

OK

8. A somewhat more complicated version of the subroutine in exercise 7 would produce the following, better-looking results. Write this version of the subroutine. *Note:* A quick look at the Help file for the **Format** function might be advantageous at this point.

Payment Schedule Calculator

Sum of loan £9,999.99    Payment £2,399.99    Rate 9.9900%

Period	Balance	Payment	Interest	Capital
1	£9,999.99	£2,399.99	£0,999.00	£1,400.99
2	£8,599.00	£2,399.99	£0,859.04	£1,540.95
3	£7,058.05	£2,399.99	£0,705.10	£1,694.89
4	£5,363.16	£2,399.99	£0,535.78	£1,864.21
5	£3,498.95	£2,399.99	£0,349.54	£2,050.45
6	£1,448.50	£1,593.21	£0,144.71	£1,448.50

OK

9. A sliding payment schedule involves payment that changes by a fixed percentage over the life of the loan. Write a sliding payment version of the payment schedule calculator in exercise 8. In addition to all the inputs described above, the subroutine will get a payment rate of change (as percentage) from the user. This is what it should look like in action:

Payment Schedule Calculator

Sum of loan £10,000.00    Payment £2,000.00    Rate 10.00%    Payment rate 10.00%

Period	Balance	Payment	Interest	Capital
1	£10,000.00	£02,000.00	£01,000.00	£01,000.00
2	£09,000.00	£02,200.00	£00,900.00	£01,300.00
3	£07,700.00	£02,420.00	£00,770.00	£01,650.00
4	£06,050.00	£02,662.00	£00,605.00	£02,057.00
5	£03,993.00	£02,928.20	£00,399.30	£02,528.90
6	£01,464.10	£01,610.51	£00,146.41	£01,464.10

OK



---

# 39

## Objects and Add-Ins

---

### 39.1 Overview

This chapter deals with several more advanced subjects in VBA. Most of these subjects relate to the Excel Object Model. The bulk of the chapter describes some useful Excel objects and ways of dealing with them. Names, a way to make worksheets clearer and more readable, are presented in section 39.6. The chapter closes with a discussion of Excel Add-Ins, one easy way to make self-crafted functions automatically available across workbooks.

---

### 39.2 Introduction to Worksheet Objects

Objects are the basic building blocks of VBA. Although you may not be aware that you are using objects, most things you do in VBA require the manipulation of objects. We can think of an object as a sort of a container with variables, functions, and subroutines inside. All of Excel's components (workbooks, worksheets, ranges, etc.) are represented by an object in the VBA Object Hierarchy. The object's data are held in special variables called properties that can be accessed using the Dot (.) operator. One of the most important object types in VBA is the **Range Object**. A worksheet cell and a range of cells are all objects of the type **Range**. The following subsection introduces some predefined **Range Object** variables.

#### The Active Cell

VBA has many variables predefined for our use; one of the more useful is `ActiveCell`. `ActiveCell` is a predefined **Range Object** variable that represents the cell in the worksheet with the cursor box around it. The following function replaces the contents of the active cell with a string representation of the contents. We use the property `Formula`; this property holds the text in the cell as a string and can be changed.

```
Sub ToString()  
    ActiveCell.Formula = "'" & _  
    ActiveCell.Formula  
End Sub
```

	A	B	C	D
1	TOSTRING IN ACTION		Run Sub	
2	3.141592654	<-- =PI()		

Before

	A	B	C	D
1	TOSTRING IN ACTION		Run Sub	
2	=PI()	<-- =PI()		

After

The Selection

Another very useful predefined variable is the `Selection`. This variable represents the currently selected item in Excel. Unlike the `ActiveCell` variable, `Selection` is not limited to a range and can be any selection (range, chart, and many more). We suggest that you check the type using the `TypeName` function. Methods are functions contained within an object. Methods are used to manipulate the object. Like properties, methods can be accessed using the Dot (.) operator. The line between methods and properties is sometimes very fuzzy. The following subroutine demonstrates some methods of the **Range Object**, and use of the predefined variable `Selection`:

```
Sub SelectBlank()  
    If UCase(TypeName(Selection)) <> "RANGE" _  
    Then Exit Sub  
    Selection.SpecialCells(xlCellTypeBlanks). _  
    Select  
End Sub
```

The first line checks to see if the current selection is a range and stops the subroutine if it is not; a message would have been appropriate under non-educational circumstances.

The first part of the second line `Selection.SpecialCells (xlCellTypeBlanks)` uses the `SpecialCells` method of the **Range Object** to return a `Range` containing all the blank cells in the current selection.

The `Select` method of the returned `Range` is activated to select it.

	A	B	C	D	E	F
1	SELECTBLANK IN ACTION				Run Sub	
2	123			123		
3		123				
4			123	124		

Before

	A	B	C	D	E	F
1	SELECTBLANK IN ACTION				Run Sub	
2	123			123		
3		123				
4			123	124		

After

---

### 39.3 The Range Object

In the previous section we encountered some predefined **Range Object** variables. This section demonstrates the use of ranges in VBA and presents more of the properties and methods of the **Range Object**.

#### A Range as a Parameter to a Function

In this subsection we build a function that accepts a `Range` as a parameter. Our new function, named `MeanReturn`, accepts a column range of asset prices as a parameter and computes and returns the mean return of the assets in the column. Recall that the return of an asset for period  $t$  is

$$r_t = \frac{Price_t - Price_{t-1}}{Price_{t-1}} \text{ and the mean return of an asset is } \bar{r} = \frac{1}{N} \sum_{t=1}^N r_t. \text{ An}$$

auxiliary function `AssetReturn` is used to compute  $r_t$ .

```
Function MeanReturn(Rng)
    NumRows = Rng.Rows.Count
    Prices = Rng.Value
    T = 0
    For i = 2 To NumRows
        T = T + AssetReturn(Prices(i - 1, 1), _
            Prices(i, 1))
    Next i
    MeanReturn = T / (NumRows - 1)
End Function
```

Lines of note:

- `NumRows = Rng.Rows.Count`

In this line the Dot operator is used twice. `Rng` is our Range object. `Rows` is property of the Range object so `Rng.Rows` is an object of the Collection type that represents all the rows in our range. `Count` is a property of Collection type objects that stores the number of members in the collection, so `Rng.Rows.Count` is a variable that stores the number of rows in our range.

- `Prices = Rng.Value`

`Value` is a property of the Range object containing the values of all the cells in the range. `Value` is of the type `Variant`. If the range is more than one cell in size `Value` is a two-dimensional array. The first index of `Value` is the row index starting from 1, and the second index is the column index starting from 1.



	A	B	C
1	MEANRETURN IN ACTION		
2	100		
3	110	10.00%	<-- =(A3-A2)/A2
4	121	10.00%	<-- =(A4-A3)/A3
5	145	19.83%	<-- =(A5-A4)/A4
6	174	20.00%	<-- =(A6-A5)/A5
7		14.96%	<-- =AVERAGE(B3:B6)
8		14.96%	<-- =MeanReturn(A2:A6)

The Range Property

The Range property is one way to access a range on a worksheet. Range is a property of many Excel objects. When used on its own, as in the next subroutine, Range is a short way of writing `ActiveSheet.Range`.

```
Sub RangeDemo()  
    Range("A2").Formula = 23  
End Sub
```

As expected, the subroutine will set the formula in cell A2 of the active worksheet to 23.

	A	B	C	D	E	F
1	RANGEDEMO IN ACTION				Run Sub	
2						

Before

	A	B	C	D	E	F
1	RANGEDEMO IN ACTION				Run Sub	
2	23					

After

The next subroutine sets the formula of each cell in the range A2:C3 of the active worksheet to 23.

```
Sub RangeDemo1 ()
    Range("A2:C3").Formula = 23
End Sub
```

	A	B	C	D	E	F
1	<b>RANGEDEMO1 IN ACTION</b>				<b>Run Sub</b>	
2	23	23	23			
3	23	23	23			

Another way of addressing a range of cells using the Range property is demonstrated by the next subroutine. The subroutine sets the formula of each cell in the range A2:C3 of the active worksheet to 23. The first argument to Range is the cell in the top left corner of the range, and the second is the cell in the bottom right corner of the range.

```
Sub RangeDemo2 ()
    Range("A2", "C3").Formula = 23
End Sub
```

Range is also a property of the Range object. The range returned by Range when used this way is relative to the Range object. The next subroutine sets the formula of the cell C3 of the active worksheet to 999.

```
Sub RangeDemo3 ()
    Range("B2").Range("B2").Formula = 999
End Sub
```

*Note:* `Range ("B2")` returns the range (or cell) B2 of the active worksheet. `Range ("B2") .Range ("B2")` returns the cell B2 of the range that has B2 as the top left corner. In worksheet terms, `Range ("B2") .Range ("B2")` returns the cell C3.

The next subroutine sets the formula of each cell in the range C2:D3 of the active worksheet to 23. The subroutine uses the cell C2 as a starting point.

```
Sub RangeDemo4()  
    Range("C2").Range("A1", "B2").Formula = 23  
End Sub
```

*Note:* `Range ("C2")` is the same as `Range ("C2") .Range ("A1")` and refers to the cell C2 in the worksheet. `Range ("C2") .Range ("B2")` refers to the cell D3 in the worksheet, B2 means one column to the left and one line down. And so `Range ("C2") .Range ("A1", "B2")` is the same as `Range ("C2", "D3")`.

---

## 39.4 The With Statement

The `With` statement allows you to perform a series of statements on a specified object without restating the obvious (the object's name and its pedigree, which can be very long). If you have more than one property to change or more than one method to use for a single object, use the `With` statement. `With` statements make your procedures run faster, and help you avoid repetitive typing. The following, somewhat contrived, subroutine sets some properties of the font of the cell in the top left-hand corner of the current region of the active cell. The font is set to be Arial, bold, and 15 points in size.

```
Sub WithoutDemo()  
    ActiveCell.CurrentRegion.Range("A1"). _  
        Font.Bold = True  
    ActiveCell.CurrentRegion.Range("A1"). _  
        Font.Name = "Arial"  
    ActiveCell.CurrentRegion.Range("A1"). _  
        Font.size = 15  
End Sub
```

And here is the same subroutine using the With statement:

```
Sub WithDemo()  
    With ActiveCell.CurrentRegion. _  
        Range("A1").Font  
        .Bold = True  
        .Name = "Arial"  
        .size = 15  
    End With  
End Sub
```

Notice the **Dot (.)** operator before the properties in the With statement. Recall from Chapter 38 that ActiveCell.CurrentRegion is the contiguous range of non-empty cells around the active cell (C3 in the screen shots), the same range that would be selected by pressing [Ctrl] + A in the worksheet (A1:D4 in the screen shots).

	A	B	C	D	E	F
1	WITHDEMO IN ACTION				Run Sub	
2	999	999	999	999		
3	999	999	999	999		
4	999	999	999	999		

Before

	A	B	C	D	E	F
1	WITHDEMO IN ACTION				Run Sub	
2	999	999	999	999		
3	999	999	999	999		
4	999	999	999	999		

After

---

**39.5 Collections**

A `Collection` is a set of items that can be referred to as a unit. Members can be added using the `Add` method and removed using the `Remove` method. Specific members can be referred to using an integer index. The number of members currently in a `Collection` is available via the `Count` method. Our use of `Collections` will be restricted to using the (quite numerous) arsenal of `Collections` that are part of the Excel Object Model, to name but a few, `Range` is a collection of cells, `Worksheets` is a collection of all the worksheets in a workbook, and `Workbooks` is a collection of all the open workbooks in Excel.

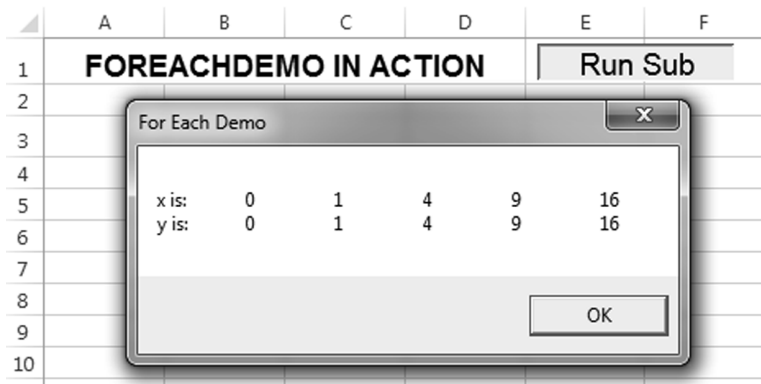
**The For Each Statement in Use with Arrays and Collections**

The `For Each` statement is a variation of the `For` loop. This statement comes in two distinct flavors. The first variation uses the statement to loop over a VBA array as demonstrated in the following subroutine:

```
Sub ForEachDemo()  
    Dim A(4)  
    For i = 0 To 4: A(i) = i * i: Next i  
    x = "x is: "  
    y = "y is: "  
    For Each Element In A  
        x = x & vbTab & Element  
        Element = Element * 2  
    Next Element  
    For Each Element In A  
        y = y & vbTab & Element  
    Next Element  
    MsgBox x & vbCrLf & y, , "For Each Demo"  
End Sub
```

Points of note:

1. The current member of the array is available to the statements within the loop body through the loop variable (Element in the above function).
2. The loop variable (Element in the above example) has to be of the type Variant irrespective of the array type.
3. Changes to Element will **not** be reflected in the actual array. Notice that the changes to Element in line 8 are not reflected in y.
4. You don't need to know the number of dimensions or the range of indices to loop over the array.



### The For Each Statement in Use with Collections

The second version of the For Each statement loops over Collections:

```
Sub ZeroRange()  
    Set Rng = ActiveCell.CurrentRegion  
    For Each Cell In Rng  
        Cell.Formula = 0  
    Next Cell  
End Sub
```

Here is what happens when you run the subroutine:

	A	B	C	D	E	F	G	H
1	ZERORANGE IN ACTION						Run Sub	
2								
3	1		3	4		16		
4	10		8	7		17		
5	11		13	14		18		

Before

	A	B	C	D	E	F	G	H
1	ZERORANGE IN ACTION						Run Sub	
2								
3	1		0	0		16		
4	10		0	0		17		
5	11		0	0		18		

After

Points of note:

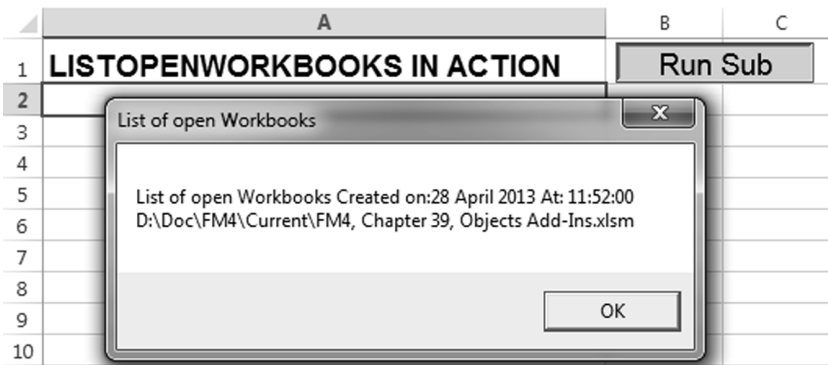
- 1. Ranges are collections and so the variable **Rng** is a collection of all the cells in the current region of the active cell (C3:D15 in our example).
- 2. **Cell** is a variable used to iterate over **all** the members of the collection.
- 3. **Cell** has to be one of the following types: Variant, Object, or the specific type of element the Collection is made of. (Recall that all variables are of type **Variant** unless specifically defined.)
- 4. **Cell** refers to the actual member of the Collection, and changes to Cell will be reflected in the Collection.
- 5. A complete explanation of the use of the Set statement is beyond the scope of this book. For our purposes, just prefix the reserved word Set to all object assignments.



### The Workbooks Collection and the Workbook Object

All the currently open workbooks are represented by a **Workbook** object in the **Workbooks Collection**. The following subroutine lists all open workbooks:

```
Sub ListOpenWorkbooks()  
    Temp = "List of open Workbooks" & _  
        " Created on:" & FormatDateTime(Date, _  
            vbLongDate) _  
        & " At: " & FormatDateTime(Time, _  
            vbLongTime)  
    For Each Element In Workbooks  
        Temp = Temp & vbCrLf & Element.FullName  
    Next Element  
    MsgBox Temp, vbOKOnly, "List of open _  
        Workbooks"  
End Sub
```



Lines of note:

```
Temp = "List of open Workbooks" & _  
      " Created on:" & _  
      FormatDateTime(Date, vbLongDate) _  
      & " At: " & _  
      FormatDateTime(Time, vbLongTime)
```

- The `Date` function returns the current system date.
- The `Time` function returns the current system time.
- The `FormatDateTime` function formats `Date` and `Time` variables for display.

```
For Each Element In Workbooks  
    Temp = Temp & vbCrLf & Element.FullName  
Next Element
```

The `For` statement loops over the entire `Workbooks` Collection. On each iteration, `Element` is one of the `Workbook` objects in the Collection. `FullName` is a property of the `Workbook` object containing the full path name of the workbook.

### The Worksheets Collection and the Worksheet Object

All the worksheets in a workbook are `Worksheet` objects in the `Worksheets` Collection that is a property of the `Workbook` object. We can use the `Worksheets` Collection without an object as a short form for `ActiveWorkbook.Worksheets`.

---

**39.6   Names**

In Excel you can use user-defined names to refer to a cell or a range of cells. Use easy-to-understand names, such as `Products`, to refer to hard-to-understand ranges, such as `Sales!C20:C30`. Using names can make formulas easy to read: Compare the formula `=sum('sheet12'!a10:a10)` to `=sum(lastYearSales)`. This section deals with the VBA side of names.

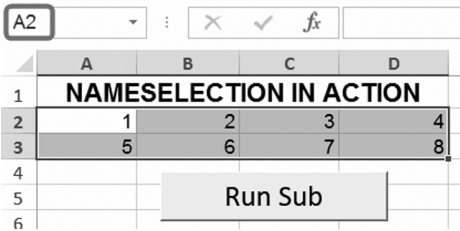
**Naming a Range Using a Subroutine**

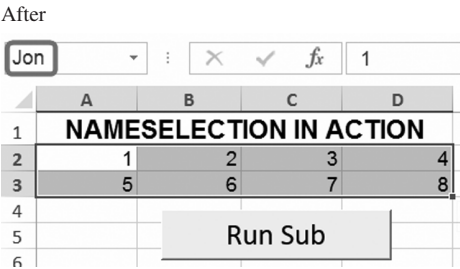
The following subroutine gives the name “Jon” to the cells selected.

```
Sub NameSelection()  
    Names.Add "Jon", "=" & _  
        Selection.Address  
End Sub
```

Select cells A2:B3 and run the subroutine. The next two screenshots show the Excel name box before and after we operate the `NameSelection` subroutine:

Before

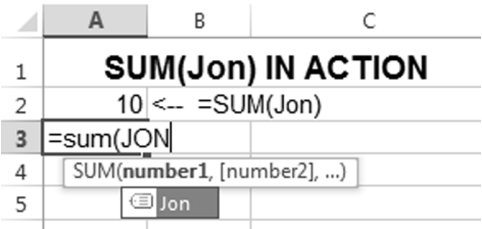




Names is a Collection of all the names in the active workbook. Add is a method of the Names Collection used to add members to the collection. We use only the first two parameters of the method. The first parameter “Jon” is the name to add to the Names Collection. The second parameter is a string containing the address, formula, or value to which the added name refers, preceded by =.

Looking for Defined Names

The name “Jon” has just now been defined, and we can use it in the workbook as demonstrated in the following screen shot. Notice that whatever capitalization you use, Excel reverts to the original “Jon.”



The name “Jon” is not directly available in VBA as demonstrated by the following function:

```
Function SumJon()  
SumJon = Application.WorksheetFunction. _  
Sum(Jon)  
End Function
```

	A	B	C
1	<b>SUM(Jon) IN ACTION</b>		
2	10	<-- =SUM(Jon)	
3	0	<-- =SumJon()	

**Referring to a Named Range**

To get to values in a named range we can use the built-in function `Application.Evaluate`, as demonstrated by the next function. Note that the function is designed to be used as an **Array Function** and the use of `Application.Volatile` to make sure the value gets updated whenever a change is made to the workbook.

```
Function JonAsArray()  
Application.Volatile  
JonAsArray = Application.Evaluate("Jon")  
End Function
```

	A	B	C
1	<b>JonASARRAY IN ACTION</b>		
2	1	3	<-- {=JonAsArray()}
3	2	4	<-- {=JonAsArray()}

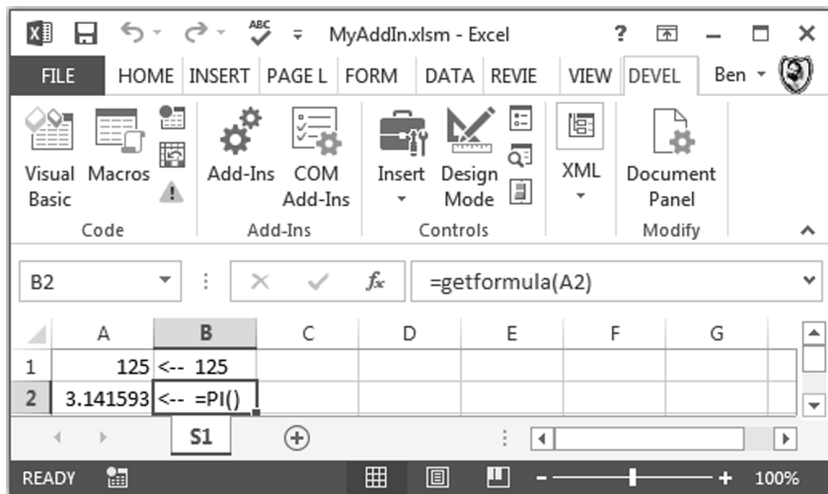
Referring to the actual range the name refers to is beyond the scope of this book.

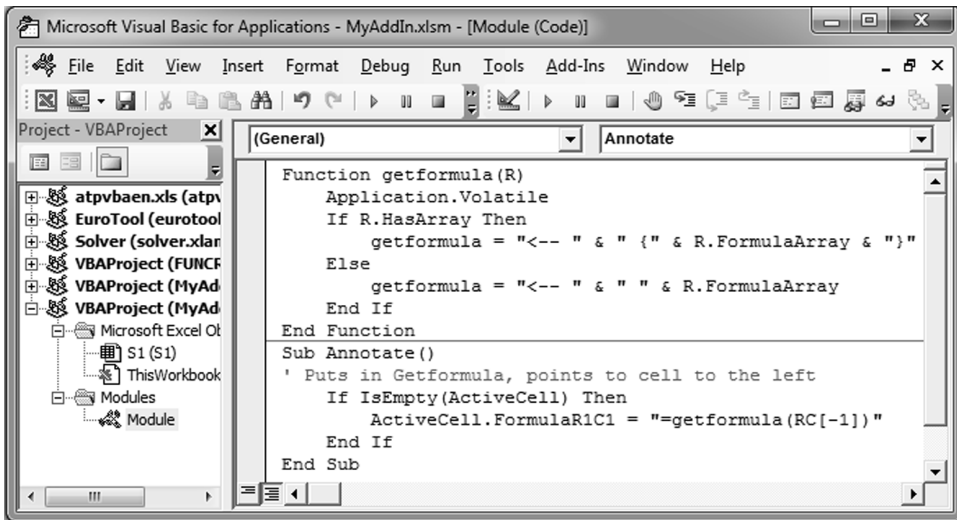
## 39.7 Add-Ins and Integration

An Excel Add-In is a file that Excel can load when it starts up. The file contains VBA code that adds additional functionality to Excel, usually in the form of new functions. Add-Ins provide an excellent way of increasing the power of Excel and they are the ideal vehicle for distributing your custom functions. This section shows you how to convert an Excel Workbook containing VBA functions to an Add-In, and how to load and use Add-Ins in Excel and VBA. The process is somewhat arcane and the steps below should be followed in the order in which they are presented.

### Create and Debug Your Base Workbook

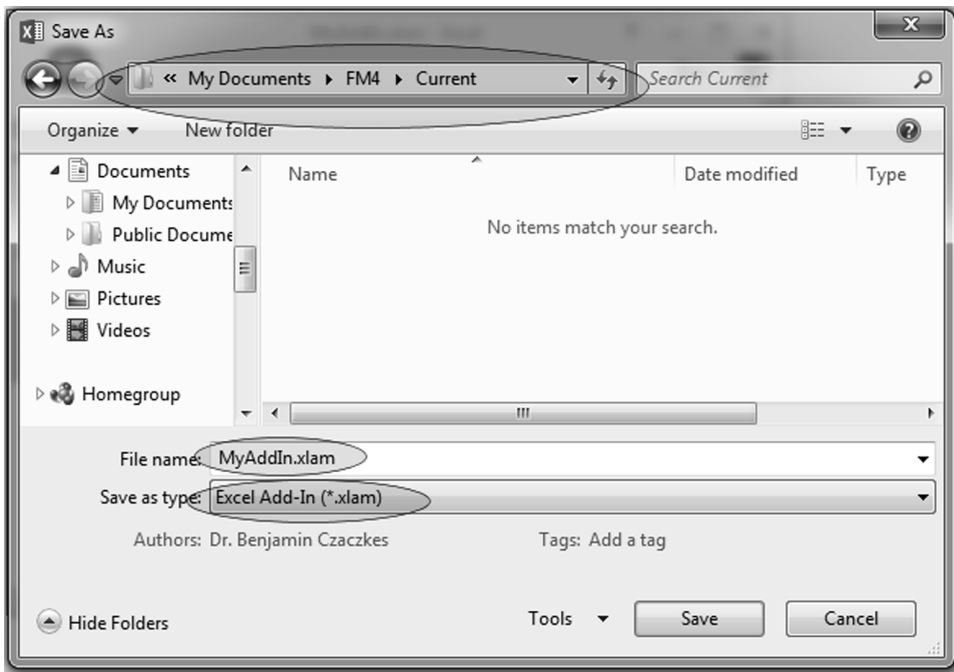
As editing an Add-In once it's created is very difficult, it is important that the original workbook on which the Add-In is based is kept intact and as a workbook. For this demonstration we have created a workbook containing one worksheet and a VBA project containing one module with one function and one subroutine.





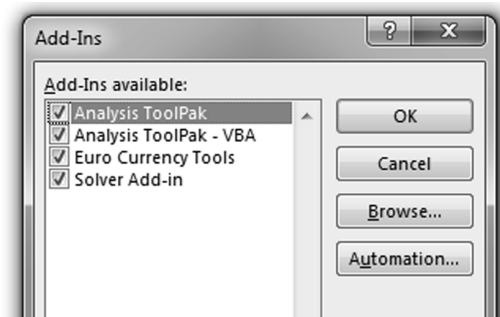
### Convert the Base Workbook to an Add-In

To make the Add-In, save the workbook as an Add-In. Select **Save As** from the Excel file menu and change **Save as type** to “Excel Add-In (\*.xlsm).” The **Save in** location will change to the Add-Ins directory on your computer. You may want to navigate to a different location (we tend to keep files together). Now click **Save**. You may want to use a new name for the Add-In (we did not).



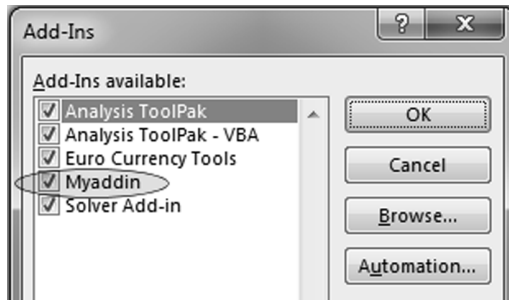
### Install and Use an Add-In from an Excel Worksheet

Installing an Add-In is done on a per computer basis (actually per computer user basis). So we do not get confused with the chapter we suggest you close Excel and reopen it with a brand new Excel workbook. Select **Developer! Add-Ins**. The following dialogue should be presented (your names may vary):



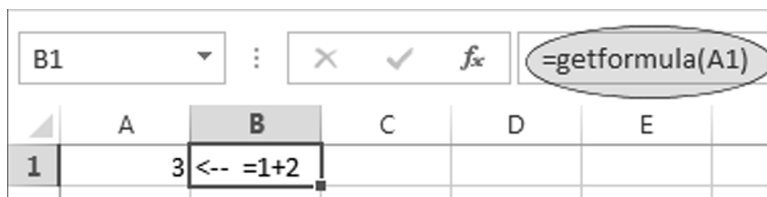
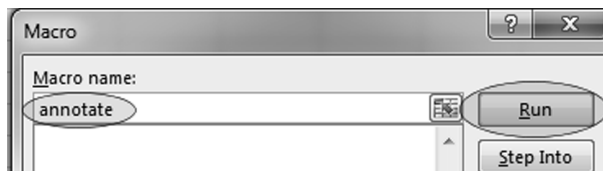


Click **Browse** and navigate to the location of your Add-In. Select it and click **OK**.



Notice that a new Add-In is available and activated. Click **OK** to close the Add-Ins dialogue. All the functions in our Add-in are now available to all workbooks in Excel. To verify insert a formula in a cell (A1), select the cell next to the cell with the formula (B1), and press [Alt] + F8.

You should get the **Macro** dialogue box. Notice that, sadly, annotate is not on the list, but if you type “annotate” in the **Macro name** box, then the **Run** button will become available and when pressed will produce the expected results.



39.8 Summary

This chapter discussed two separate topics. We started with a more extensive discussion of objects, which underlie the VBA programming concept. Objects allow you to be much more parsimonious in expressing your programming references. We finished the chapter with a discussion of how to build Add-Ins in Excel.

Exercises

1. Suppose you have a spreadsheet with a series of numbers and formulas:

	A	B	C
1	<b>EX1</b>	<b>Run Sub</b>	
2			
3	Price	Return	
4	1000.00		
5	1069.45	0.069451061	<-- =A5/A4-1
6	1158.53	0.083296895	<-- =A6/A5-1
7	1213.49	0.047440948	<-- =A7/A6-1
8	1269.56	0.046204665	<-- =A8/A7-1
9	1287.02	0.013745727	<-- =A9/A8-1
10	1316.34	0.022788538	<-- =A10/A9-1
11	1332.09	0.01195957	<-- =A11/A10-1

Suppose you want to turn this into:

	A	B	C
1	<b>EX1</b>	<b>Run Sub</b>	
2			
3	Price	Return	
4	1000.00		
5	1069.45	6.945106119	<-- =(A5/A4-1)*100
6	1158.53	8.329689467	<-- =(A6/A5-1)*100
7	1213.49	4.744094796	<-- =(A7/A6-1)*100
8	1269.56	4.620466489	<-- =(A8/A7-1)*100
9	1287.02	1.374572664	<-- =(A9/A8-1)*100
10	1316.34	2.278853849	<-- =(A10/A9-1)*100
11	1332.09	1.195956963	<-- =(A11/A10-1)*100

Write a subroutine that does this. Your subroutine should:

- Put in a set of parentheses and multiply the cell contents by 100.
- Move down one cell (see **ActiveCellDemo1**, section 39.1).
- Ask if you want to repeat the process (if “yes,” it should do it; if “no,” the subroutine should exit).

*Note:* The parentheses have to come after the “=.” The **Right** function might be used for this operation.

You may want to refer to section 39.2 for more information on the **MsgBox** function and the values it returns.

2. Rewrite the subroutine in exercise 1 so that it deals correctly with the end of the series. One possible treatment is not to ask to repeat the process when the last cell in the series is dealt with.

*Hint:* For this subroutine it might be useful to think of the last cell in the series as the cell that fulfills the criterion `Cell.Item(2, 1).Formula=""` (see section 39.2).




3. Write a subroutine that multiplies all cells in the current region by 2.
4. Rewrite the subroutine in exercise 3 so that its action is dependent on the cell’s contents.
- If the cell contents is a formula, it will be replaced by the same formula multiplied by 2.
  - If the cell contents is a number, it will be replaced by a number equal to the old number multiplied by 2.
  - On all other cells in the current region, nothing will be done.

*Note:* To make life easier, you may assume, for the purposes of this exercise, that a formula is anything beginning with “=” and a number is anything beginning with the characters “0” to “9.”

5. Rewrite the subroutine in exercise 4 so that it uses another method (the correct one) to detect the existence of a formula in a cell. Look at the different properties of the **Range** object in the Help file.
6. The annotations (using **Getformula**) for worksheet formulas in this book were done with a subroutine. For example, running the subroutine on this worksheet

A4		:	<input type="button" value="✕"/>	<input type="button" value="✓"/>	<input type="button" value="fx"/>	=B3+A3
	A		B			C
1	EX6		Run Sub			
2						
3	12		14			
4	26					

produces the following:

A4		:				=B3+A3
	A		B			C
1	EX6		Run Sub			
2						
3	12		14			
4	26	<--	=B3+A3			

Write a subroutine to perform the annotation. If the cell immediately to the right of the active cell is not empty, the subroutine should overwrite it with **Getformula** only after receiving confirmation from the user.

7. The **Selection** object represents the current selection in the worksheet. **Selection** is usually, and for our purposes always, a **Range** object. Rewrite the subroutine in exercise 6 so that it works on a selected range.

Note the following:

If the selected range is a single cell, activate the subroutine in exercise 6.

If the selected range is a column, activate a subroutine repeatedly for all cells in the column.

If the selected range is more than one column, the subroutine should abort with an appropriate message.

8. Array functions are functions that return more than one value. For example, the **Transpose** worksheet function returns its argument turned by 90 degrees, as the following worksheet demonstrates:

	A	B	C	D	E	F
1	TRANSPOSE IN ACTION					
2	1	2	3	4	1 <-- {=TRANSPOSE(A2:D2)}	
3					2 <-- {=TRANSPOSE(A2:D2)}	
4					3 <-- {=TRANSPOSE(A2:D2)}	
5					4 <-- {=TRANSPOSE(A2:D2)}	

The curly brackets were not typed in but were added by Excel to indicate an array formula. The following subroutine created the preceding worksheet:

```
Sub TransposeMe()  
    Range("E3:E6").FormulaArray = "=Transpose(A3:D3)"  
End Sub
```

The next subroutine is a more complicated version that could deal with any size or place in the row range:

```
Sub TransposeMeToo()  
    C = Selection.Columns.Count  
    R = Selection.Rows.Count  
    If C = 1 Then 'Its a Column  
        MsgBox "I don't do Columns"  
    ElseIf R = 1 Then 'Its a Row  
        Selection.Cells(1, C + 1).Range("A1:A" & C). _  
            FormulaArray = "=Transpose(" & _  
                & Selection.AddressLocal(False, False) & ")"  
    Else 'What is it?  
        MsgBox "What is it?"  
    End If  
End Sub
```

Rewrite **TransposeMeToo** so it could deal with column ranges as well as row ranges.

9. Rewrite **TransposeMeToo** of exercise 8 so it could deal with **all** ranges.