



EXCEL VBA
MADE
EASY

Disclaimer

Excel VBA Made Easy is an independent publication and is not affiliated with, nor has it been authorized, sponsored, or otherwise approved by Microsoft Corporation.

Trademarks

Microsoft, Visual Basic, Excel and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Liability

The purpose of this book is to provide basic guideline for people interested in Excel VBA programming. Although every effort and care has been taken to make the information as accurate as possible, the author shall not be liable for any error, harm or damage arising from using the instructions given in this book.

ISBN: 1449959628

Copyright© Liew Voon Kiong 2009. All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, without permission in writing from the author.

Acknowledgement

I would like to express my sincere gratitude to many people who have made their contributions in one way or another to the successful publication of this book.

My special thanks go to my children Xiang, Yi and Xun. My daughter Xiang edited this book while my sons Yi and Xun contributed their ideas and even wrote some of the sample programs for this book. I would also like to appreciate the support provided by my beloved wife Kim Huang and my youngest daughter Yuan. I would also like to thank the million of visitors to my **VBA Tutorial** website at http://www.vbtutor.net/VBA/vba_tutorial.html for their support and encouragement.

About the Author

Dr. Liew Voon Kiong holds a bachelor degree in Mathematics(BSc), a master degree in management (MM) and a doctoral degree in business administration(DBA). He has been involved in programming for more than 15 years. He created the popular online Visual Basic Tutorial at www.vbtutor.net in 1996 and since then the web site has attracted millions of visitors and it is one of the top searched **Visual Basic** websites in many search engines including Google. In order to provide more support for Excel VBA hobbyists, he has written this book based on his online VBA tutorial at http://www.vbtutor.net/VBA/vba_tutorial.html. He is also the author of **Visual Basic 6 Made Easy** and **Visual Basic 2008 Made Easy**

CONTENTS

Chapter 1	Introduction to Excel VBA	1
Chapter 2	Working with Variables in Excel VBA	7
Chapter 3	Using Message box and Input box in Excel VBA	16
Chapter 4	Using If....Then....Else in Excel VBA	25
Chapter 5	For.....Next Loop	32
Chapter 6	Do.....Loop	40
Chapter 7	Select Case.....End Select	44
Chapter 8	Excel VBA Objects Part 1–An Introduction	46
Chapter 9	Excel VBA Objects Part 2 –The Workbook Object	55
Chapter 10	Excel VBA Objects Part 3 –The Worksheet Object	60
Chapter 11	Excel VBA Objects Part 4–The Range Object	66
Chapter 12	Working with Excel VBA Controls	75
Chapter 13	VBA Procedures Part 1-Functions	86
Chapter 14	VBA Procedures Part 2-Sub Procedures	96
Chapter 15	String Handling Functions	99
Chapter 16	Date and Time Functions	102
Chapter 17	Sample Excel VBA Programs	109

Chapter 1

Introduction to Excel VBA

1.1 The Concept of Excel VBA

VBA is the acronym for Visual Basic for Applications. It is an integration of the Microsoft's event-driven programming language Visual Basic with Microsoft Office applications such as Microsoft Excel, Microsoft Word, Microsoft PowerPoint and more. By running Visual Basic IDE within the Microsoft Office applications, we can build customized solutions and programs to enhance the capabilities of those applications.

Among the Visual Basic for applications, Microsoft Excel VBA is the most popular. There are many reasons why we should learn VBA for Microsoft Excel, among them is you can learn the fundamentals of Visual Basic programming within the MS Excel environment, without having to purchase a copy of Microsoft Visual Basic software. Another reason is by learning Excel VBA; you can build custom made functions to complement the built-in formulas and functions of Microsoft Excel. Although MS Excel has many built-in formulas and functions, they are not enough for certain complex calculations and applications. For example, it is very difficult to calculate monthly payment for a loan taken using Excel's built-in formulas, but it is relatively easier to write VBA code for such calculation. This book is written in such a way that you can learn VBA for MS Excel at your own pace.

You can write Excel VBA code in every version of Microsoft Office, including MS Office 97, MS Office 2000, MS Office 2002, MS Office 2003, MS Office XP, MS Office 2007 and MS Office 2010. By using VBA, you can build some very powerful tools in

MS Excel, including financial and scientific applications that can perform financial calculations and programs that can perform statistical analyses.

1.2 The Visual Basic Editor in MS Excel

There are two ways which you can start VBA programming in MS Excel. The first is to place a command button on the spreadsheet and start programming by clicking the command button to launch the Visual Basic Editor. The second way is to launch the Visual Basic Editor by clicking on the Tools menu then select Macro from the drop-down menu and choose Visual Basic Editor. Lets start with the command button first. In order to place a command button on the MS Excel spreadsheet, you click the View item on the MS Excel menu bar and then click on toolbars and finally select the Control Toolbox after which the control toolbox bar will appear, as shown in Figure 1.1. ,then click on the command button and draw it on the spreadsheet, as shown in Figure 1.2.

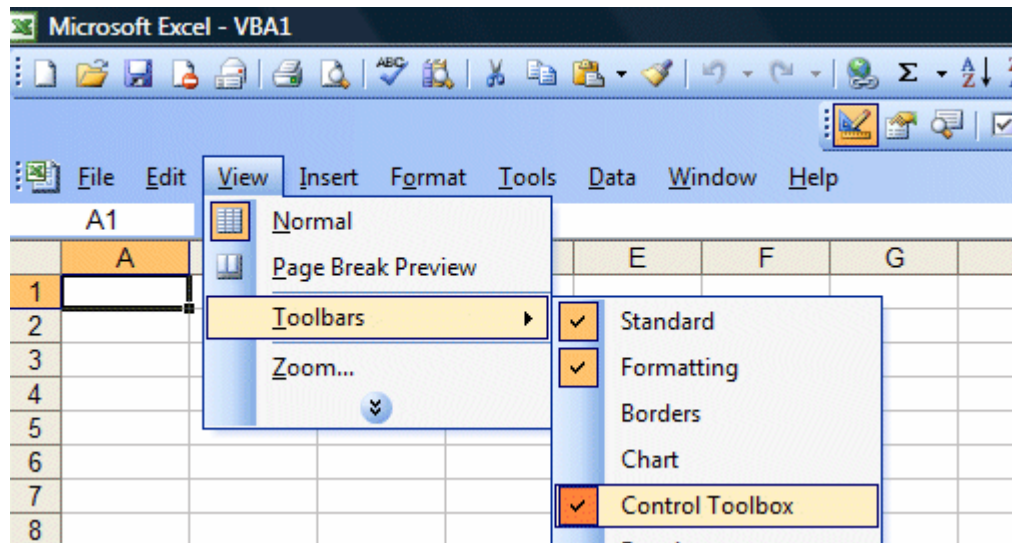


Figure 1.1: Displaying Control Toolbox in MS Excel.

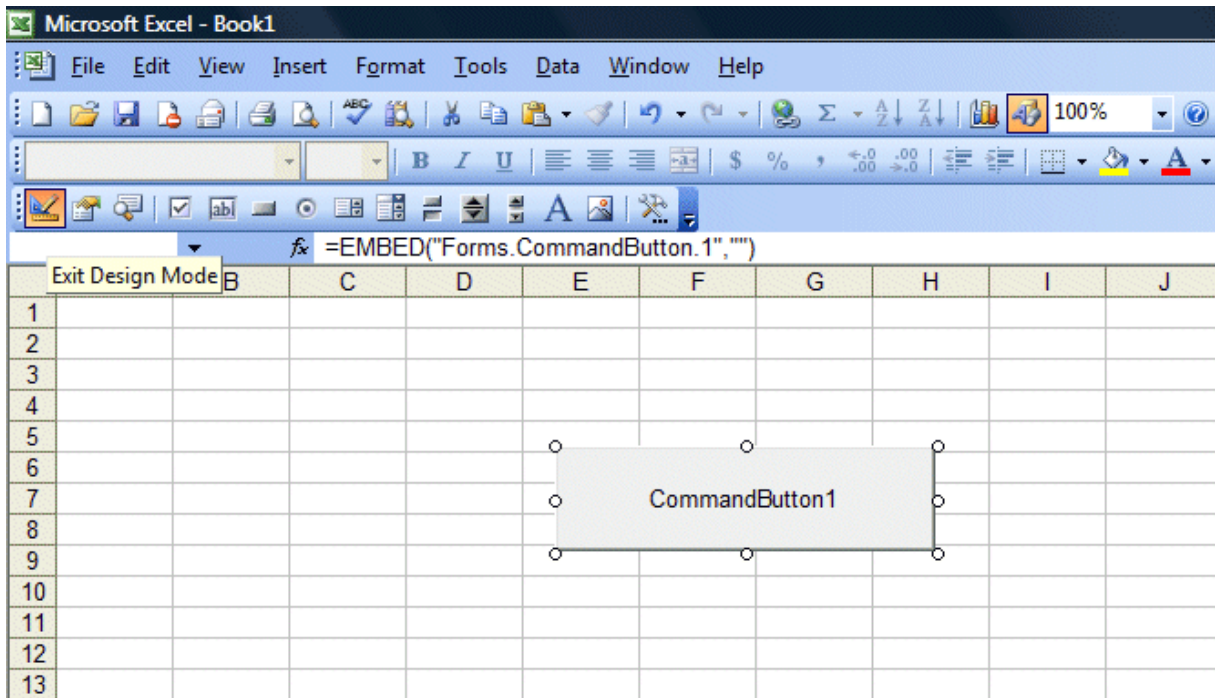


Figure 1.2: The Command Button in Design Mode

Now you select the command button and make sure the design button on the far left of the control toolbox is depressed. Next, click on the command button to launch the Visual Basic Editor. Enter the statements as shown in figure 1.3. Let's write out the code here:

Example 1.1

```
Private Sub CommandButton1_Click ()
```

```
    Range ("A1:A10").Value="Visual Basic "
```

```
    Range ("C11").Value=Range (" A11").Value +Range ("B11").Value
```

```
End Sub
```

The first statement will fill up cell A1 to cell A10 with the phrase "Visual Basic" while the second statement add the values in cell A11 and cell B11 and then display the sum in cell C11. To run the program, you need to exit the Visual Basic Editor by clicking the Excel button on the far left corner of the tool bar. When you are in the MS Excel environment, you can exit the design mode by clicking the design button, then click on the command button.

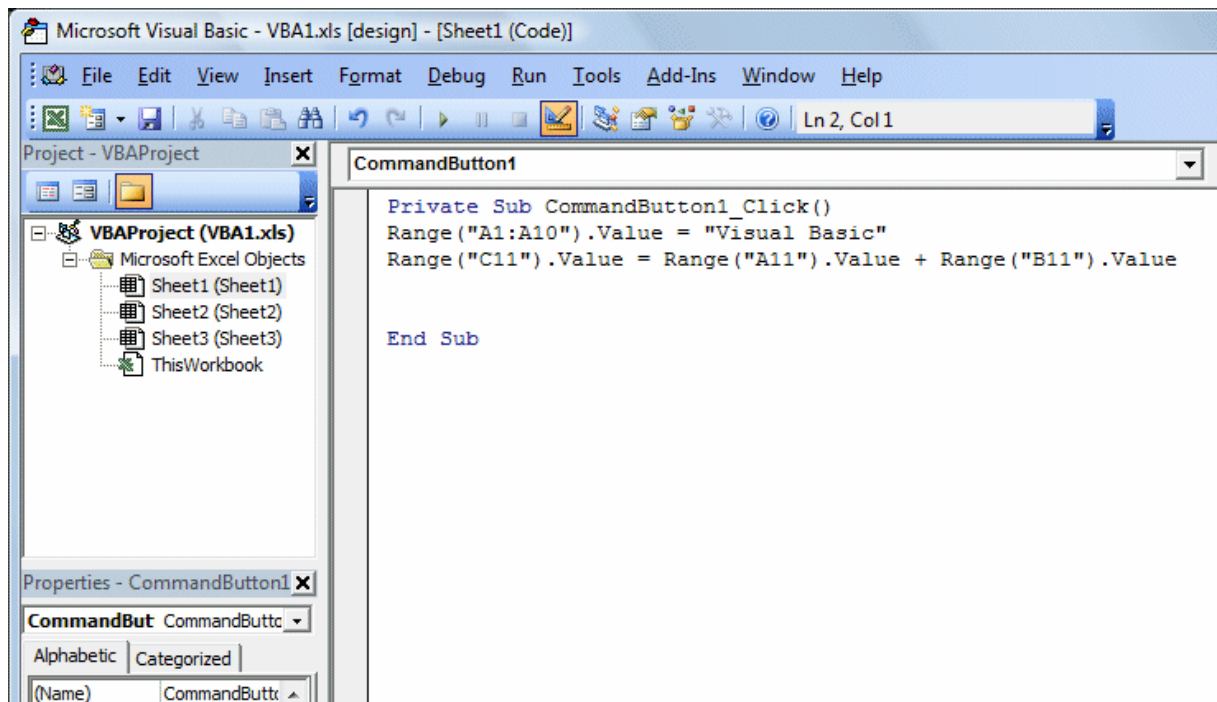


Figure 1.3: The Visual Basic Editor IDE in MS Excel

Running the above VBA will give you the output as shown in Figure 1.4

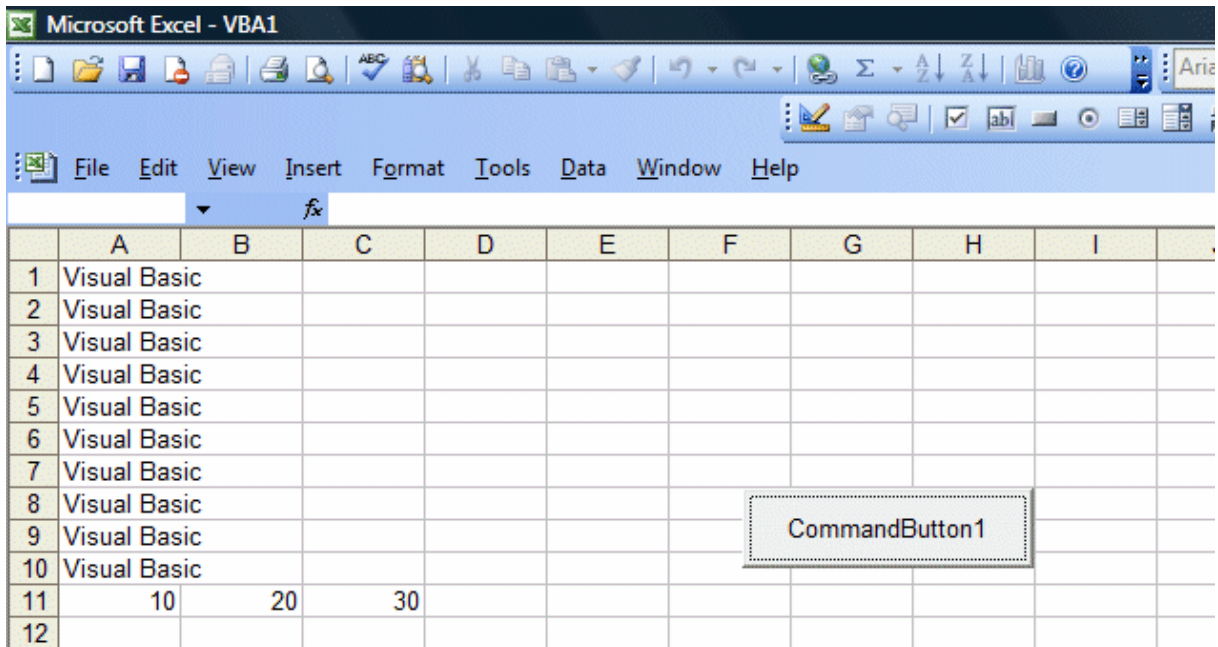


Figure 1.4:

1.3 The Excel VBA Code

Writing Excel VBA code is almost exactly the same as writing code in Visual Basic, which means you have to use syntaxes similar to Visual Basic. However, there are codes specially designed for use in MS Excel, such as the use of the object or function called **Range**. It is the function that specifies the value of a cell or a range of cells in MS Excel spreadsheet. The format of using Range is as follows:

`Range("cell Name").Value=K` or `Range("Range of Cells").Value=K`

Where Value is the property of Range and K can be a numeric value or a string

Example 1.2

```
Private Sub CommandButton1_Click ()
```

```
    Range ("A1").Value= "VBA"
```

```
End Sub
```

The above example will enter the text "VBA" into cell A1 of the MS Excel spreadsheet when the user presses the command button. You can also use Range without the Value property, as shown in Example 1.3:

Example 1.3

```
Private Sub CommandButton1_Click ()
```

```
    Range ("A1") = 100
```

```
End Sub
```

In the above example, clicking the command button will enter the value of 100 into cell A1 of the MS Excel spreadsheet. The following example demonstrates how to input values into a range of cells:

Example 1.4

```
Private Sub CommandButton1_Click ()
```

```
    Range ("A1:A10") = 100
```

```
End Sub
```

Chapter 2

Working with Variables in Excel VBA

2.1 The Concept of Variables

Variables are like mail boxes in the post office. The contents of the variables change every now and then, just like the mail boxes. In Excel VBA, variables are areas allocated by the computer memory to hold data. Like the mail boxes, each variable must be given a name. To name a variable in Excel VBA, you have to follow a set of rules, as follows:

a) Variable Names

The following are the rules when naming the variables in VBA

- ❖ It must be less than 255 characters
- ❖ No spacing is allowed
- ❖ It must not begin with a number
- ❖ Period is not permitted

Examples of valid and invalid variable names are displayed in Table 2.1

Table 2.1: Examples of valid and invalid variable names

Valid Name	Invalid Name	
My_Car	My.Car	
this year	1NewBoy	
Long_Name_Can_beUSE	He&HisFather	*& is not acceptable
Group88	Student ID	* Space not allowed

b) Declaring Variables

In VBA, we need to declare the variables before using them by assigning names and data types. There are many VBA data types, which can be grossly divided into two types, namely the numeric data types and the non-numeric data types.

i) Numeric Data Types

Numeric data types are types of data that consist of numbers, which can be computed mathematically with various standard arithmetic operators such as addition, subtraction, multiplication, division and more. In VBA, the numeric data are divided into 7 types, which are summarized in Table 2.2.

Table 2.2: Numeric Data Types

Type	Storage	Range of Values
Byte	1 byte	0 to 255
Integer	2 bytes	-32,768 to 32,767
Long	4 bytes	-2,147,483,648 to 2,147,483,648
Single	4 bytes	-3.402823E+38 to -1.401298E-45 for negative values 1.401298E-45 to 3.402823E+38 for positive values.
Double	8 bytes	-1.79769313486232e+308 to -4.94065645841247E-324 for negative values 4.94065645841247E-324 to 1.79769313486232e+308 for positive values.
Currency	8 bytes	-922,337,203,685,477.5808 to 922,337,203,685,477.5807
Decimal	12 bytes	+/- 79,228,162,514,264,337,593,543,950,335 if no decimal is use +/- 7.9228162514264337593543950335 (28 decimal places).

ii) Non-numeric Data Types

Nonnumeric data types are summarized in Table 2.3

Table 2.3: Nonnumeric Data Types

Data Type	Storage	Range
String(fixed length)	Length of string	1 to 65,400 characters
String(variable length)	Length + 10 bytes	0 to 2 billion characters
Date	8 bytes	January 1, 100 to December 31, 9999
Boolean	2 bytes	True or False
Object	4 bytes	Any embedded object
Variant(numeric)	16 bytes	Any value as large as Double
Variant(text)	Length+22 bytes	Same as variable-length string

You can declare the variables implicitly or explicitly. For example, `sum=text1.text` means that the variable `sum` is declared implicitly and ready to receive the input in `textbox1`. Other examples of implicit declaration are `volume=8` and `label="Welcome"`. On the other hand, for explicit declaration, variables are normally declared in the general section of the code window using the `Dim` statements. Here is the syntax:

```
Dim variableName as DataType
```

Example 2.1

```
Dim password As String
Dim yourName As String
Dim firstnum As Integer
Dim secondnum As Integer
Dim total As Integer
Dim BirthDay As Date
```

You may also combine them into one line, separating each variable with a comma, as follows:

```
Dim password As String, yourName As String, firstnum As Integer.
```

If the data type is not specified, VBE will automatically declare the variable as a Variant. For string declaration, there are two possible formats, one for the variable-length string and another for the fixed-length string. For the variable-length string, just use the same format as Example 2.1 above. However, for the fixed-length string, you have to use the format as shown below:

```
Dim VariableName as String * n
```

Where n defines the number of characters the string can hold. For example, Dim yourName as String * 10 mean yourName can hold no more than 10 Characters.

Example 2.2

In this example, we declared three types of variables, namely the string, date and currency.

```
Private Sub CommandButton1_Click()  
    Dim YourName As String, BirthDay As Date, Income As Currency  
    YourName = "Alex"  
    BirthDay = "1 April 1980"  
    Income = 1000  
    Range("A1") = YourName  
    Range("A2") = BirthDay  
    Range("A3") = Income  
End Sub
```

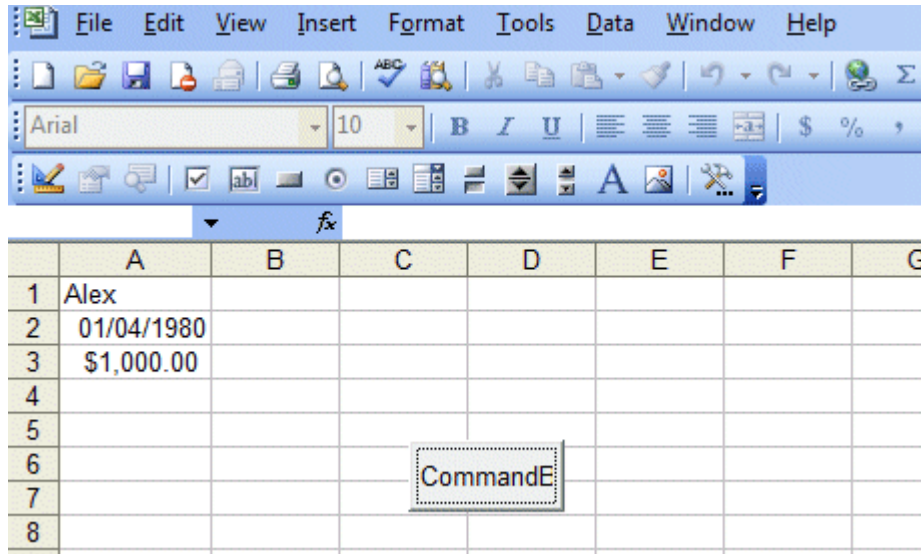


Figure 2.1: Output screen for Example 2.2

2.2 The use of Option Explicit

The use of Option Explicit is to help us to track errors in the usage of variable names within a program code. For example, if we commit a typo, VBE will pop up an error message “Variable not defined”. Indeed, Option Explicit forces the programmer to declare all the variables using the Dim keyword. It is a good practice to use Option Explicit because it will prevent us from using incorrect variable names due to typing errors, especially when the program gets larger. With the usage of Option Explicit, it will save us time in debugging our programs.

When Option Explicit is included in the program code, we have to declare all variables with the Dim keyword. Any variable not declared or wrongly typed will cause the program to popup the “Variable not defined” error message. We have to correct the error before the program can continue to run.

Example 2.3

This example uses the Option Explicit keyword and it demonstrates how a typo is being tracked.

Option Explicit

```
Private Sub CommandButton1_Click()  
    Dim YourName As String, password As String  
    YourName = "John"  
    password = 12345  
    Cells(1, 2) = YourNam  
    Cells(1, 3) = password  
End Sub
```

The typo is *YourNam* and the error message ‘variable not defined’ is displayed .

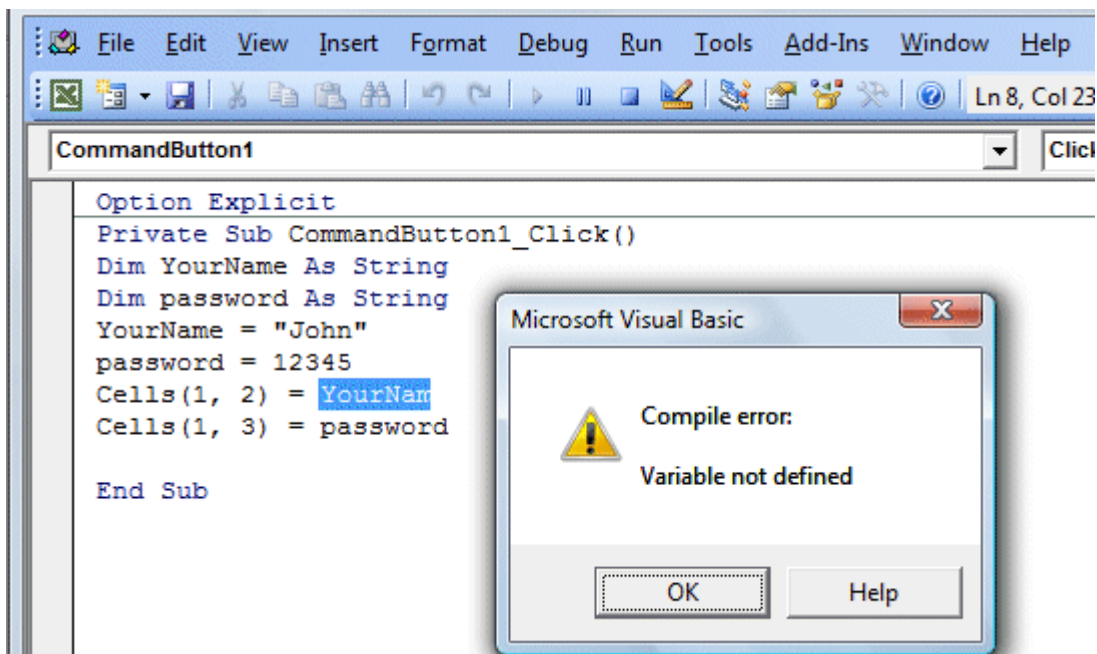


Figure 2.2: Error message due to typo error

2.3 Assigning Values to the Variables

After declaring various variables using the Dim statements, we can assign values to those variables. The general format of an assignment is

Variable=Expression

The variable can be a declared variable or a control property value. The expression could be a mathematical expression, a number, a string, a Boolean value (true or false) and more. Here are some examples:

```
firstNumber=100
```

```
secondNumber=firstNumber-99
```

```
userName="John Lyan"
```

```
userpass.Text = password
```

```
Label1.Visible = True
```

```
Command1.Visible = false
```

```
ThirdNumber = Val(usernum1.Text)
```

```
total = firstNumber + secondNumber+ThirdNumber
```

2.4 Performing Arithmetic Operations in Excel VBA

In order to compute input from the user and to generate results in Excel VBA, we can use various mathematical operators. In Excel VBA, except for + and -, the symbols for the operators are different from normal mathematical operators, as shown in Table 2.3.

Table 2.3: Arithmetic Operators

Operator	Mathematical function	Example
^	Exponential	$2^4=16$
*	Multiplication	$4*3=12$
/	Division	$12/4=3$
Mod	Modulus (return the remainder from an integer division)	$15 \text{ Mod } 4=3$
\	Integer Division (discards the decimal places)	$19 \setminus 4=4$
+ or &	String concatenation	"Visual"&"Basic"="Visual Basic"

Example 2.4

Option Explicit

```

Private Sub CommandButton1_Click ()
    Dim number1, number2, number3 as Single
    Dim total, average as Double
    number1=Cells (1, 1).Value
    number2=Cells (2, 1).Value
    number3= Cells (3, 1).Value
    Total=number1+number2+number3
    Average=Total/3
    Cells (5, 1) =Total
    Cells (6, 1) =Average

```

```

End Sub

```

In example 2.4, three variables are declared as single and another two variables are declared as variant. Variant means the variable can hold any numeric data type. The program computes the total and average of the three numbers that are entered into three cells in the Excel spreadsheet.

Example 2.5

Option Explicit

```
Private Sub CommandButton1_Click()
    Dim secondName As String, yourName As String
    firstName = Cells(1,1).Value
    secondName = Cells(2,1).Value
    yourName = firstName + " " + secondName
    Cells(3,1) = yourName
```

End Sub

In the above example, three variables are declared as string. The variable `firstName` and the variable `secondName` will receive their data entered by the user into `Cells(1,1)` and `cells(2,1)` respectively. The variable `yourName` will be assigned the data by combining the first two variables. Finally, `yourName` is displayed on `Cells (3, 1)`. Performing addition on strings will result in concatenation of the strings, as shown in figure 2.3 below. Names in A1 and A2 are joined up and displayed in A3.

	A	B	C	D
1	James			
2	Barrack			
3	James Barrack			
4			CommandButton1	
5				

Figure 2.3: Concatenation of Strings

Chapter 3

Using Message box and Input box

There are many built-in functions available in Excel VBA which we can use to streamline our VBA programs. Among them, *message box* and *input box* are most commonly used. These two functions are useful because they make the Excel VBA macro programs more interactive. The input box allows the user to enter the data while the message box displays output to the user.

3.1 The MsgBox () Function

The objective of the MsgBox function is to produce a pop-up message box and prompt the user to click on a command button before he or she can continue. The code for the message box is as follows:

```
yourMsg=MsgBox(Prompt, Style Value, Title)
```

The first argument, Prompt, displays the message in the message box. The Style Value determines what type of command button that will appear in the message box. Table 3.1 lists the command button that can be displayed. The Title argument displays the title of the message box.

Table 3.1: Style Values and Command Buttons

Style Value	Named Constant	Button Displayed
0	vbOkOnly	Ok button
1	vbOkCancel	Ok and Cancel buttons
2	vbAbortRetryIgnore	Abort, Retry and Ignore buttons.
3	vbYesNoCancel	Yes, No and Cancel buttons
4	vbYesNo	Yes and No buttons
5	vbRetryCancel	Retry and Cancel buttons

We can use the named constant in place of integers for the second argument to make the programs more readable. In fact, VBA will automatically show a list of named constants where you can select one of them. For example, **yourMsg=MsgBox("Click OK to Proceed", 1, "Startup Menu")** and **yourMsg=Msg("Click OK to Proceed". vbOkCancel,"Startup Menu")** are the same. `yourMsg` is a variable that holds values that are returned by the `MsgBox ()` function. The values are determined by the type of buttons being clicked by the users. It has to be declared as Integer data type in the procedure or in the general declaration section. Table 3.2 shows the values, the corresponding named constants and the buttons.

Table 3.2: Returned Values and Command Buttons

Value	Named Constant	Button Clicked
1	vbOk	Ok button
2	vbCancel	Cancel button
3	vbAbort	Abort button
4	vbRetry	Retry button
5	vbIgnore	Ignore button
6	vbYes	Yes button
7	vbNo	No button

Example 3.1

In this example, the message in cell (1,2) “Your first VBA program” will be displayed in the message box. As no named constant is added, the message will simply display the message and the “OK” button, as shown in Figure 3.1

```
Private Sub CommandButton1_Click()
    Dim YourMsg As String
    Cells(1, 2) = "Your first VBA program"
    YourMsg = Cells(1, 2)
    MsgBox YourMsg
End Sub
```

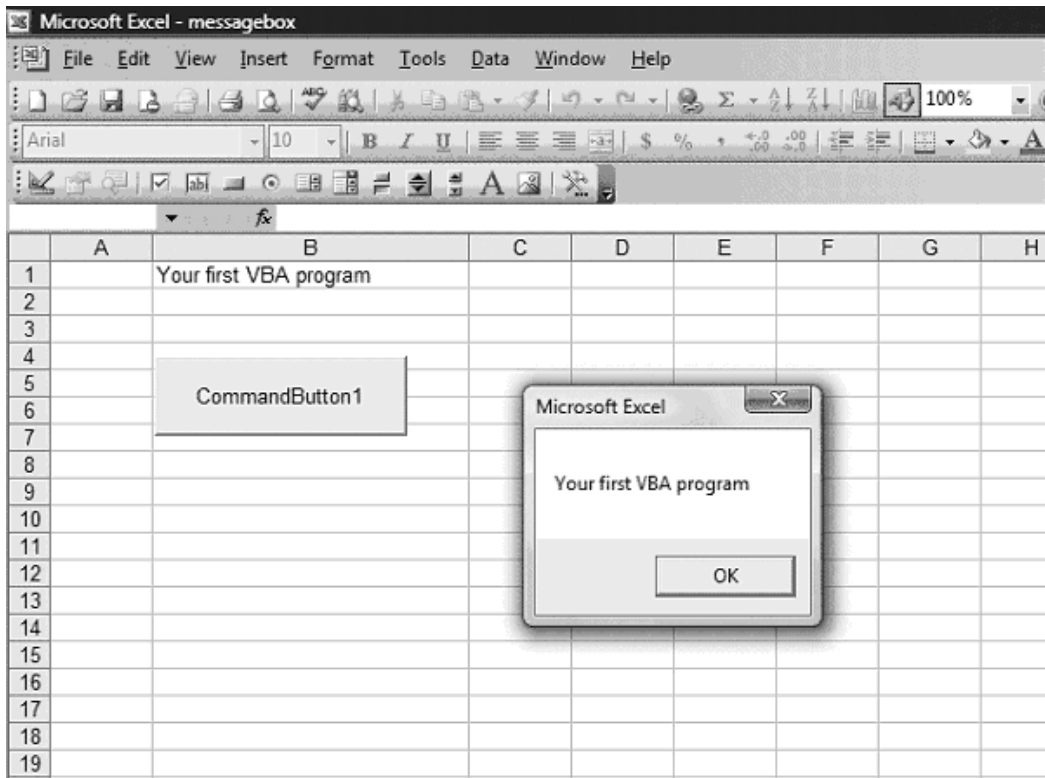


Figure 3.1: Message box with the OK button

Example 3.2

In this Example, the named constant *vbYesNoCancel* is added as the second argument, so the message box will display the Yes, No and the Cancel buttons, as shown in Figure 3.2.

```
Private Sub CommandButton1_Click()
    Dim YourMsg As String
    Cells(1, 2) = "Your first VBA program"
    YourMsg = Cells(1, 2)
    MsgBox YourMsg, vbYesNoCancel
End Sub
```

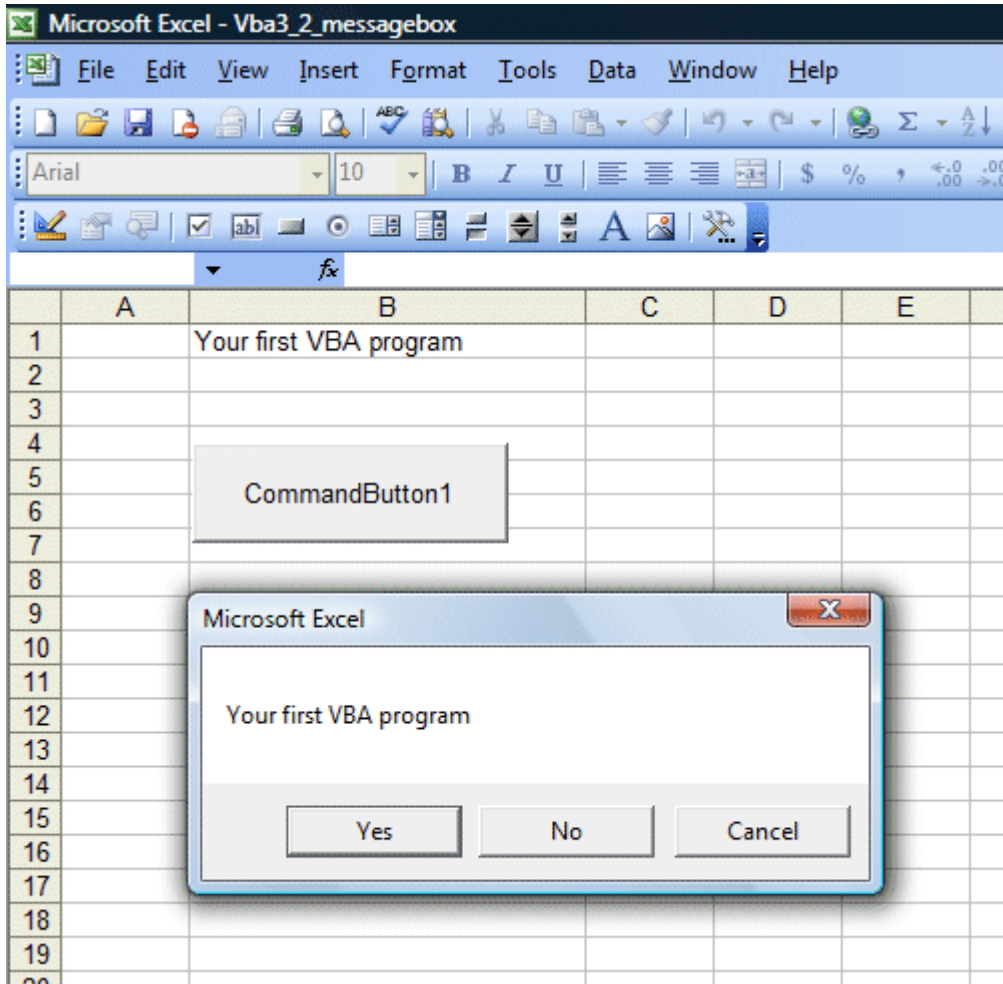






Figure 3.2: Message box with the Yes, No and Cancel buttons

To make the message box look more sophisticated, you can add an icon beside the message. There are four types of icons available in VBE, as shown in Table 11.3.

Table 3.3

Value	Named Constant	Icon
16	vbCritical	
32	vbQuestion	
48	vbExclamation	
64	vbInformation	

Example 3.3

The code in this example is basically the same as Example 3.2, but the named vbExclamation is added as the third argument. The two name constants can be joined together using the “+” sign. The message box will now display the exclamation icon, as shown in Figure 3.3.

```
Private Sub CommandButton1_Click()  
    Dim YourMsg As String  
    Cells(1, 2) = "Your first VBA program"  
    YourMsg = Cells(1, 2)  
    MsgBox YourMsg, vbYesNoCancel + vbExclamation  
End Sub
```

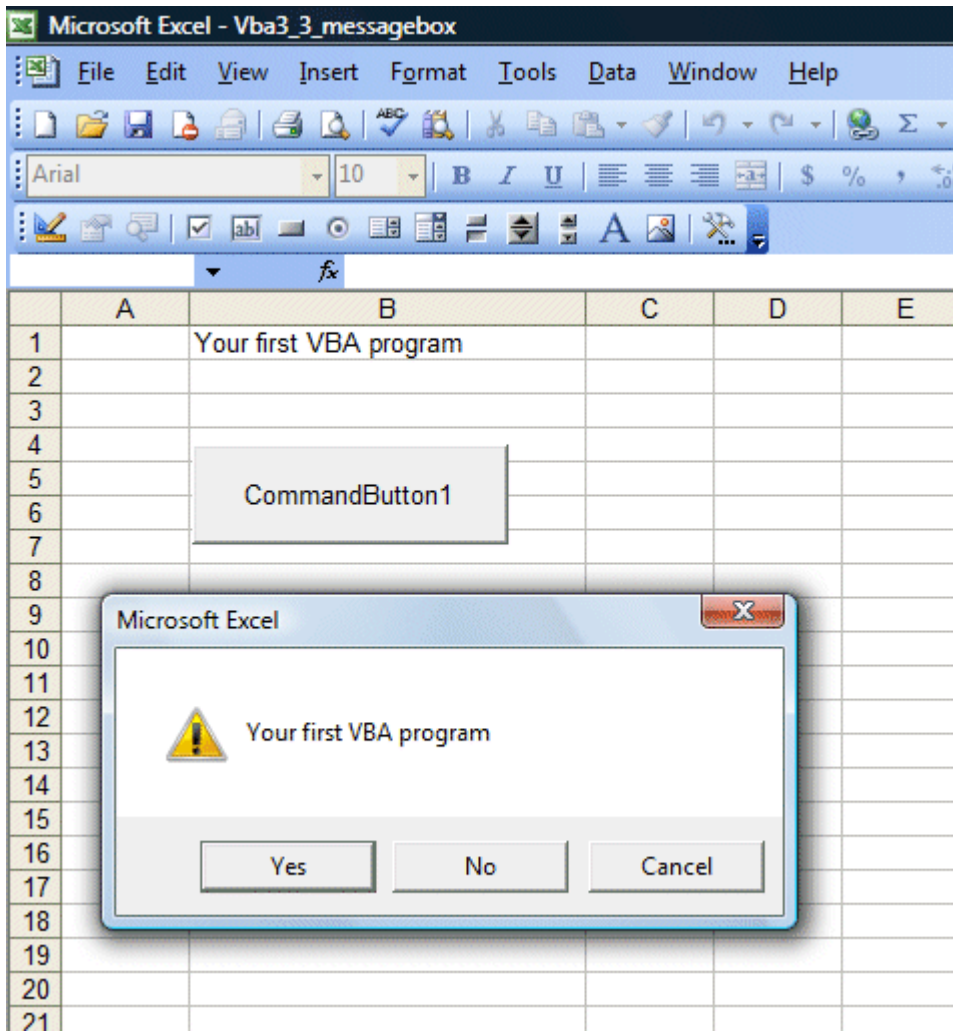


Figure 3.3: Message box with the exclamation icon.

You can even track which button is clicked by the user based on the returned values shown in Table 3.2. In Example 3.4, the conditional operators If....Then...Else are used. You do not have to really understand the program logics yet, they will be explained in later chapter.

Example 3.4

```

Private Sub CommandButton1_Click()
Dim testMsg As Integer
testMsg = MsgBox("Click to Test", vbYesNoCancel + vbExclamation, "Test
Message")
If testMsg = 6 Then
Cells(1,1).Value = "Yes button was clicked"
ElseIf testMsg = 7 Then
Cells(1,1).Value = "No button was clicked"
Else
Cells(1,1).Value = "Cancel button was clicked"
End If
End Sub

```

3.2 The InputBox() Function

An InputBox() is a function that displays an input box where the user can enter a value or a message in the form of text. The format is

```
myMessage=InputBox(Prompt, Title, default_text, x-position, y-position)
```

myMessage is a variant data type but typically it is declared as a string, which accepts the message input by the users. The arguments are explained as follows:

- Prompt - The message displayed in the inputbox.
- Title - The title of the Input Box.
- default-text - The default text that appears in the input field where users can use it as his intended input or he may change it to another message.
- x-position and y-position - the position or the coordinates of the input box.

Example 3.5

The Interface of example 3.5 is shown in Figure 3.4

```
Private Sub CommandButton1_Click()  
Dim userMsg As String  
userMsg = InputBox("What is your message?", "Message Entry Form", "Enter  
your message here", 500, 700)  
Cells(1,1).Value=userMsg  
End Sub
```

When the user clicks the OK button, the input box as shown in Figure 3.4 will appear. Notice that the caption of the input box is "Message Entry Form" and the prompt message is "What is your message". After the user enters the message and clicks the OK button, the message will be displayed in cell A1

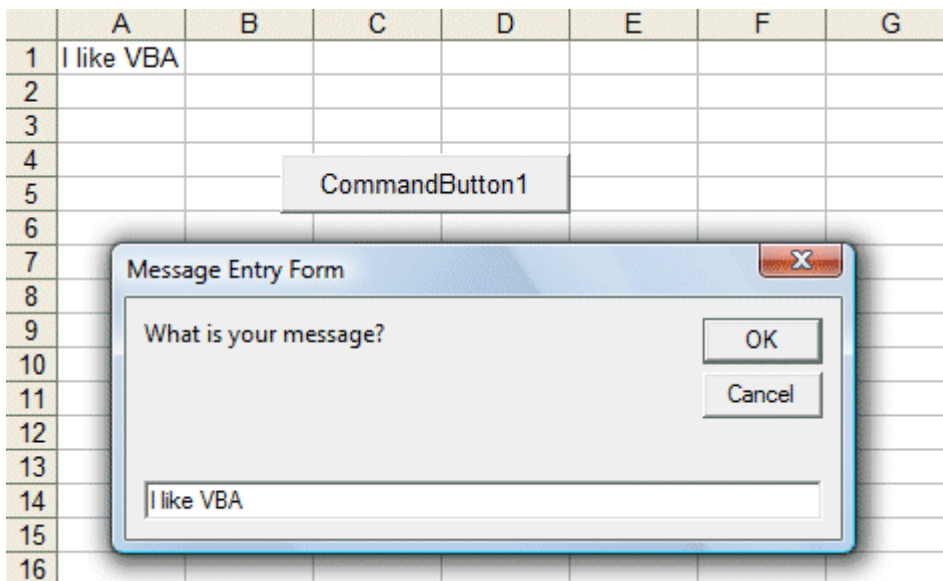


Figure 3.4: The input box

Chapter 4

Using If....Then....Else in Excel VBA

Visual Basic Editor (VBE) in MS Excel is just as powerful as the stand alone Visual Basic compiler in the sense that you can use the same commands in programming. For example, you can use If...Then...Else structure to control program flow in VBE that execute an action based on certain conditions. To control the program flow, we need to use the conditional operators as well as the logical operators, which are discussed in the following sections.

4.1 Conditional Operators

To control the VBA program flow, we can use various conditional operators. Basically, they resemble mathematical operators. Conditional operators are very powerful tools which let the VBA program compare data values and then decide what action to take. For example, it can decide whether to execute or terminate a program. These operators are shown in Table 4.1.

Table 4.1: Conditional Operators

Operator	Meaning
=	Equal to
>	More than
<	Less Than
>=	More than and equal
<=	Less than and equal
<>	Not Equal to

* You can also compare strings with the above operators. However, there are certain rules to follow: Upper case letters are lesser than lowercase letters, "A"<"B"<"C"<"D".....<"Z" and numbers are lesser than letters.

4.2 Logical Operators

In addition to conditional operators, there are a few logical operators that offer added power to the VBA programs. They are shown in Table 4.2.

Table 4.2: Logical Operators

Operator	Meaning
And	Both sides must be true
or	One side or other must be true
Xor	One side or other must be true but not both
Not	Negates truth

4.3 Using If.....Then.....Elseif....Else Statements with Operators

To effectively control the VBA program flow, we shall use the If...Then...Else statement together with the conditional operators and logical operators. The general format for If...Then...Elseif....Else statement is as follows:

```
If conditions Then
    VB expressions
```

```
Elseif
    VB expressions
Else
    VB expressions
End If
```

* Any If...Then...Else statement must end with End If. Sometime it is not necessary to use Else.

Example 4.1

```
Private Sub CommandButton1_Click()
    Dim firstnum, secondnum As Single
    firstnum = Cells(1,1).Value
    secondnum = Cells(1,2).Value
    If firstnum>secondnum Then
        MsgBox " The first number is greater than the second number"
    If firstnum<secondnum Then
        MsgBox " The first number is less than the second number"
    Else
        MsgBox " The two numbers are equal "
    End If
End Sub
```

In this example, the program compares the values in cells (1, 1) and cells (1, 2) and displays the appropriate comment in a message box. For example, If the first number

is less than the second number, it will show the message “The first number is less than the second number”, as shown in Figure 4.1.

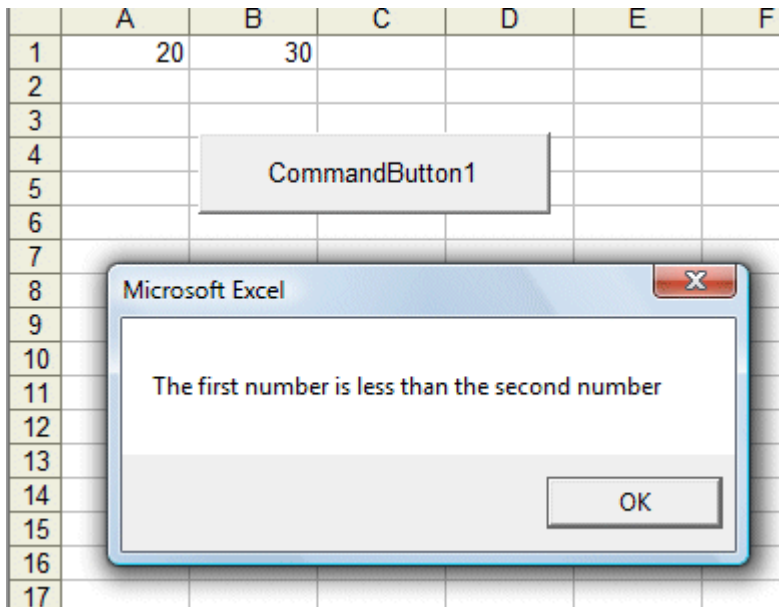


Figure 4.1

Example 4.2

In this program, you place the command button on the MS Excel spreadsheet and go into the VBE by clicking the button. At the VBE, key in the program code as shown below:

```
Private Sub CommandButton1_Click()
```



```

Dim mark As Integer
Dim grade As String
Randomize Timer
mark = Int(Rnd * 100)
Cells(1, 1).Value = mark
If mark < 20 And mark >= 0 Then
grade = "F"
Cells(2, 1).Value = grade
ElseIf mark < 30 And mark >= 20 Then
grade = "E"
Cells(2, 1).Value = grade
ElseIf mark < 40 And mark >= 30 Then
grade = "D"
Cells(2, 1).Value = grade
ElseIf mark < 50 And mark >= 40 Then
grade = "C-"
Cells(2, 1).Value = grade
ElseIf mark < 60 And mark >= 50 Then
grade = "C"
Cells(2, 1).Value = grade
ElseIf mark < 70 And mark >= 60 Then
grade = "C+"
Cells(2, 1).Value = grade
ElseIf mark < 80 And mark >= 70 Then
grade = "B"
Cells(2, 1).Value = grade
ElseIf mark <= 100 And mark > -80 Then
grade = "A"
Cells(2, 1).Value = grade
End If
End Sub

```

We use randomize timer and the Rnd function to generate random numbers. In order to generate random integers between 0 and 100, we combined the Int and Rnd

functions, $\text{Int}(\text{Rnd} * 100)$. For example, when $\text{Rnd} = 0.6543$, then $\text{Rnd} * 100 = 65.43$, and $\text{Int}(65.43) = 65$. Using the statement `cells(1,1).Value=mark` will place the value of 65 into cell(1,1).

Now, based on the mark in cells(1,1), I use the `If.....Then....Elseif` statement to put the corresponding grade in cells(2,1). So, when you click on command button 1, it will generate a random number between 1 and 100 and places it in cells (1, 1) and the corresponding grade in cells (2,1). The output is shown in Figure 4.2.

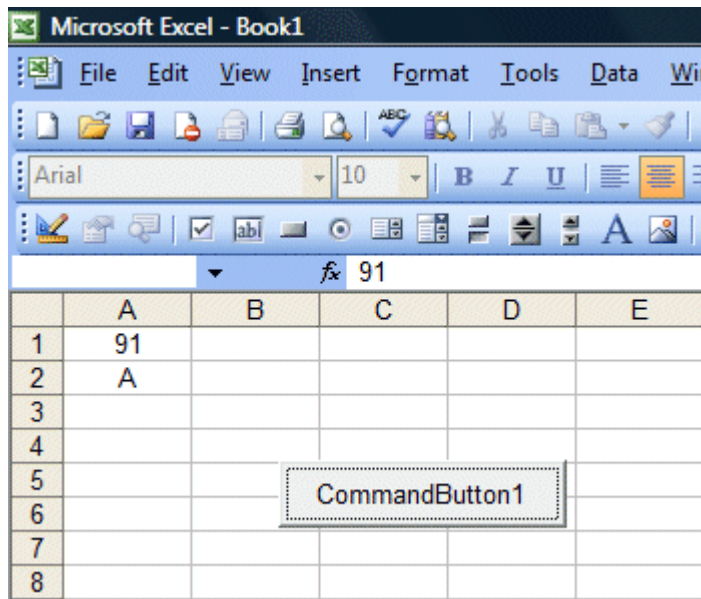


Figure 4.2

Example 4.3

This example demonstrates the use of the Not operator.

```
Private Sub CommandButton1_Click()  
    Dim x, y As Integer  
    x = Int(Rnd * 10) + 1  
    y = x Mod 2  
    If Not y = 0 Then  
        MsgBox " x is an odd number"  
    Else  
        MsgBox " x is an even number"  
    End If  
End Sub
```

In the above example, Rnd is a randomize function that produces random number between 0 and 1. So Rnd*10 produces a random number between 0 and 9. Int is a function in VBA that returns an integer. Therefore, Int(Rnd*10)+1 generates random numbers between 1 and 10. Mod is the operator that returns the remainder when a number is divided by another number. If x is an even number, x Mod 2 will produce a zero. Based on this logic, if x Mod 2 is not zero, it is an odd number; otherwise it is an even number.

Chapter 5

For.....Next Loop

Looping is a very useful feature of Excel VBA because it makes repetitive works easier. There are two kinds of loops in VB, the For.....Next loop and the Do...Loop. In this chapter, we will discuss the For....Next loop. The format of a For.....Next loop is

```
For counter=startNumber to endNumber (Step increment)
    One or more VB statements
Next
```

We will demonstrate the usage of the For....Next loop with a few examples.

Example 5.1

```
Private Sub CommandButton1_Click()
    Dim i As Integer
    For i = 1 To 10
        Cells(i, 1).Value = i
    Next
End Sub
```

In this example, you place the command button on the spreadsheet then click on it to go into the Visual Basic editor. When you click on the button , the VBA program will fill cells(1,1) with the value of 1, cells(2,1) with the value of 2, cells(3,1) with the value of 3.....until cells (10,1) with the value of 10. The position of each cell in the Excel

spreadsheet is referenced with cells (i,j), where i represents row and j represent column.

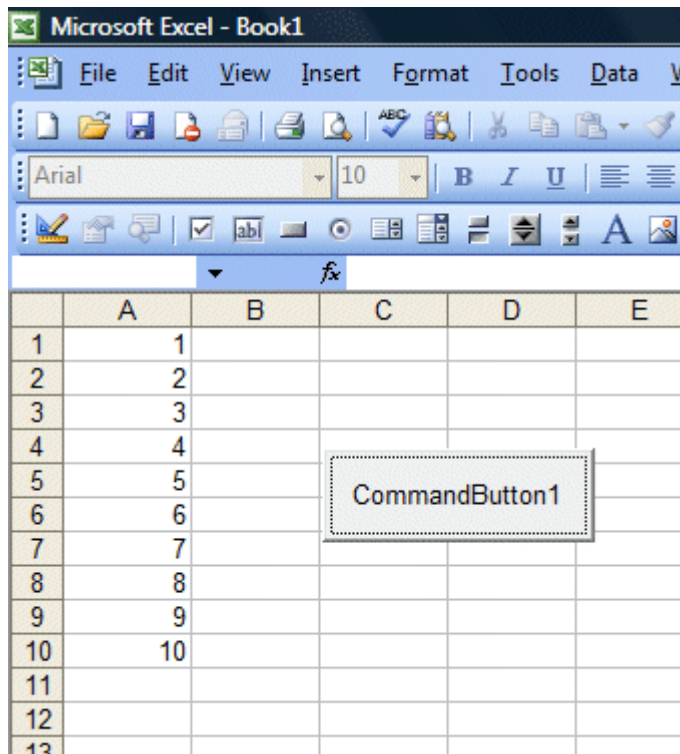


Figure 5.1: For....Next loop with single step increment

Example 5.2

In this example, the step increment is used. Here, the value of i increases by 2 after each loop. Therefore, the VBA programs will fill up alternate cells after each loop. When you click on the command button, cells (1, 1) will be filled with the value of 1, cells (2, 1) remains empty, cells (3, 1) filled with value of 3 and etc.

```
Private Sub CommandButton1_Click()  
    Dim i As Integer  
    For i = 1 To 15 step 2  
        Cells(i, 1).Value = i  
    Next  
End Sub
```

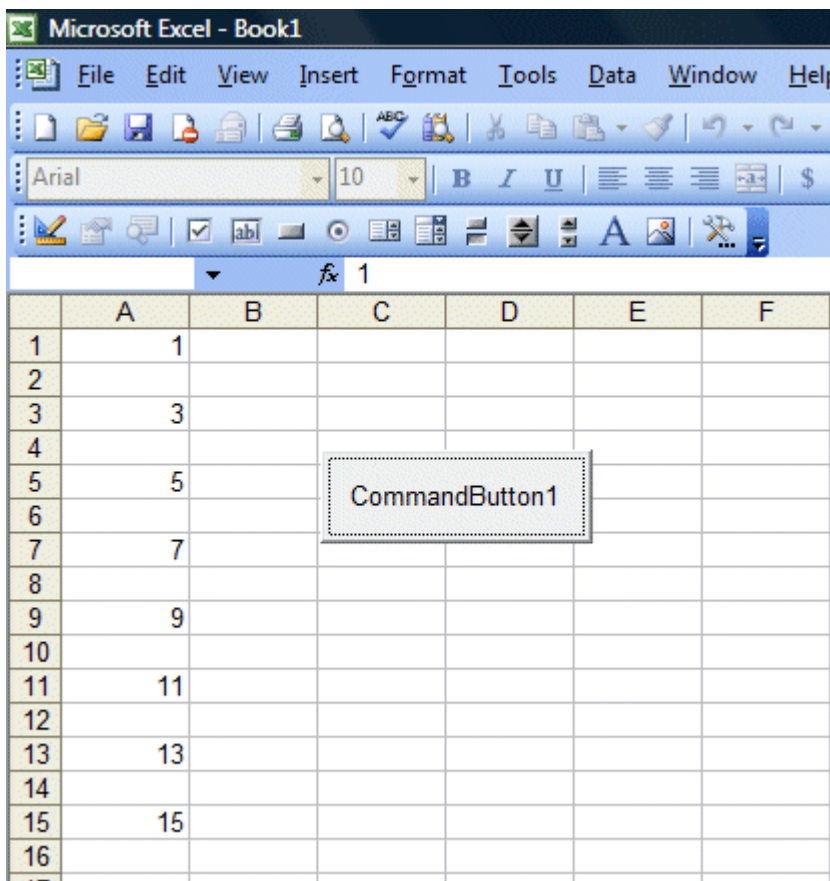


Figure 5.2: For....Next loop with step increment

If you wish to exit the ForNext loop after a condition is fulfilled, you can use the *Exit For* statement, as shown in Example 5.3.

Example 5.3

In this example, the program will stop once the value of I reaches the value of 10.

```
Private Sub CommandButton1_Click()  
    Dim i As Integer  
    For i = 1 To 15  
        Cells(i, 1).Value = i  
        If i >= 10 Then  
            Exit For  
        End If  
    Next i  
End Sub
```

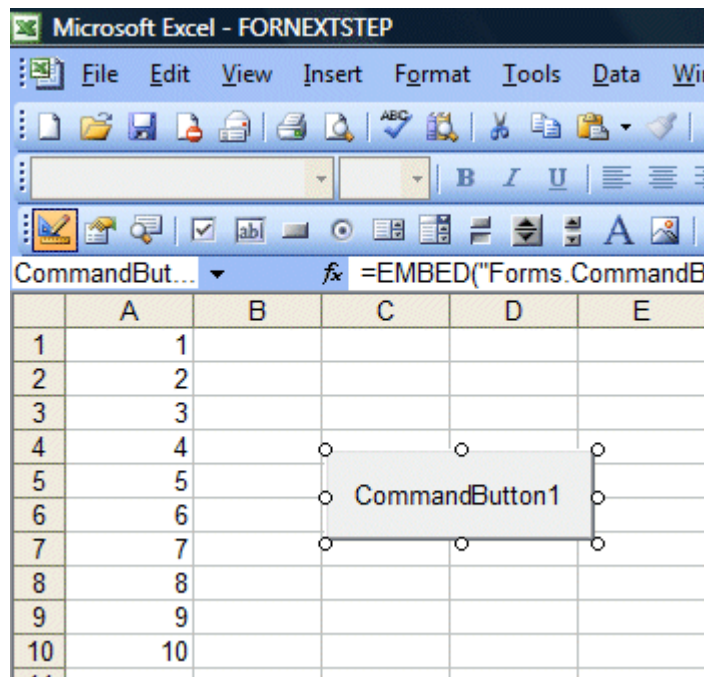


Figure 5.3: The output of Example 5.3

In previous examples, the For...Next loop will only fill up values through one column or one row only. To be able to fill up a range of values across rows and columns, we can use the nested loops, or loops inside loops. This is illustrated in **Example 5.4**.

Example 5.4

```
Private Sub CommandButton1_Click ()
    Dim i, j As Integer
    For i = 1 To 10
        For j = 1 To 5
            Cells (i, j).Value = i + j
        Next j
    Next i
End Sub
```

In this example, when $i=1$, the value of j will iterate from 1 to 5 before it goes to the next value of i , where j will iterate from 1 to 5 again. The loops will end when $i=10$ and $j=5$. In the process, it sums up the corresponding values of i and j . The concept can be illustrated in the table below:

i \ j	1	2	3	4	5
1	(1, 1)	(1, 2)	(1, 3)	(1, 4)	(1, 5)
2	(2, 1)	(2, 2)	(2, 3)	(2, 4)	(2, 5)
3	(3, 1)	(3, 2)	(3, 3)	(3, 4)	(3, 5)
4	(4, 1)	(4, 2)	(4, 3)	(4, 4)	(4, 5)
5	(5, 1)	(5, 2)	(5, 3)	(5, 4)	(5, 5)
6	(6, 1)	(6, 2)	(6, 3)	(6, 4)	(6, 5)
7	(7, 1)	(7, 2)	(7, 3)	(7, 4)	(7, 5)
8	(8, 1)	(8, 2)	(8, 3)	(8, 4)	(8, 5)
9	(9, 1)	(9, 2)	(9, 3)	(9, 4)	(9, 5)
10	(10, 1)	(10, 2)	(10, 3)	(10, 4)	(10, 5)

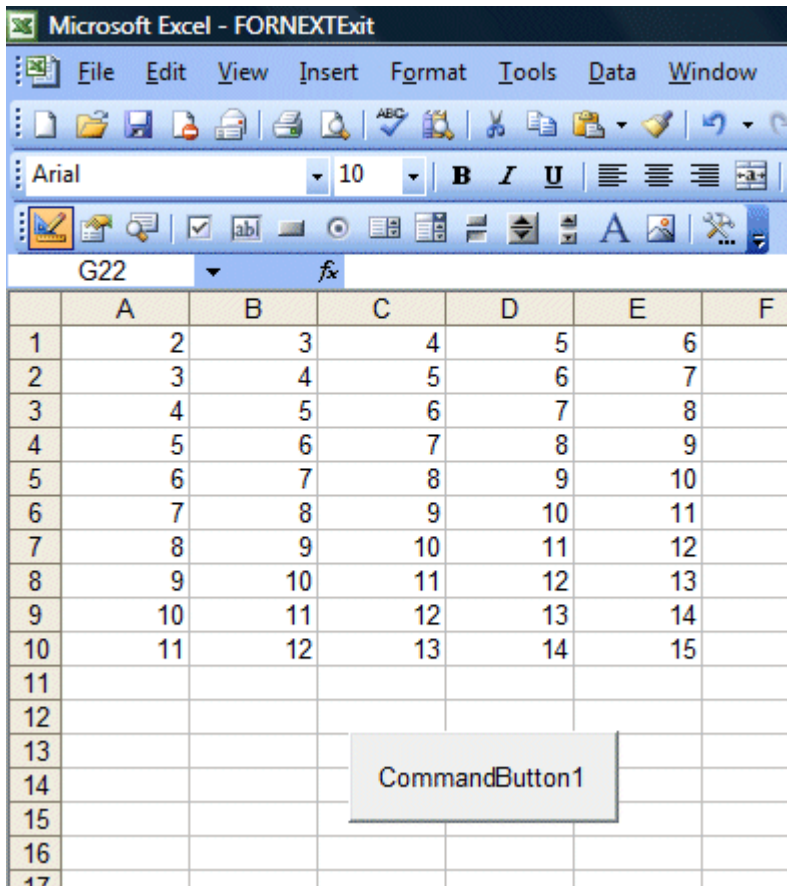


Figure 5.4: The output of Example 5.4

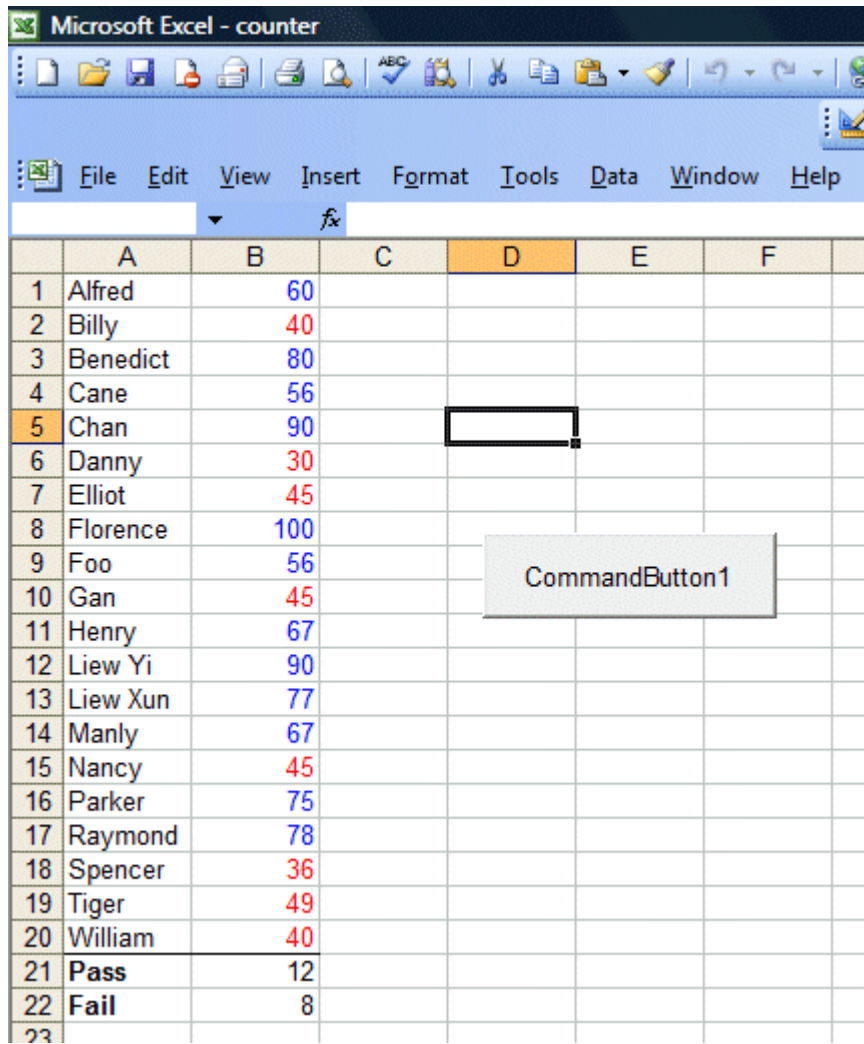
Example 5.5

This is a simple VBA counter that can count the number of passes and the number of failures for a list of marks obtained by the students in an examination. The program also differentiates the passes and failures with blue and red colors respectively. Let's examine the code below:

```
Private Sub CommandButton1_Click()
```

```
Dim i, counter As Integer
For i = 1 To 20
If Cells(i, 2).Value > 50 Then
counter = counter + 1
Cells(i, 2).Font.ColorIndex = 5
Else
'do nothing
Cells(i, 2).Font.ColorIndex = 3
End If
Next i
Cells(21, 2).Value = counter
Cells(22, 2).Value = 20 - counter
End Sub
```

This program combines the For..Next and the If ...Then...Else statements to control the program flow. If the value in that cell is more than 50, the value of counter is increased by 1 and the font color is changed to blue (colorIndex = 5) , otherwise there is no increment in the counter and the font color is changed to red (ColorIndex=3). We will discuss more about the Color property in a later chapter. The output is shown in **Figure 5.5**.



Microsoft Excel - counter

File Edit View Insert Format Tools Data Window Help

	A	B	C	D	E	F
1	Alfred	60				
2	Billy	40				
3	Benedict	80				
4	Cane	56				
5	Chan	90				
6	Danny	30				
7	Elliot	45				
8	Florence	100				
9	Foo	56				
10	Gan	45				
11	Henry	67				
12	Liew Yi	90				
13	Liew Xun	77				
14	Manly	67				
15	Nancy	45				
16	Parker	75				
17	Raymond	78				
18	Spencer	36				
19	Tiger	49				
20	William	40				
21	Pass	12				
22	Fail	8				
23						

Figure 5.5: The VBA counter

Chapter 6

Do.....Loop

In the previous chapter, you have learned how to use the *For.....Next* loop to execute a repetitive process. In this chapter, you will learn about another looping method known as the *Do Loop*. There are four ways you can use the *Do Loop* as shown below :

The formats are

a) Do While condition

Block of one or more VB statements

Loop

b) Do

Block of one or more VB statements

Loop While condition

c) Do Until condition

Block of one or more VB statements

Loop

d) Do

Block of one or more VB statements

Loop Until condition

Example 6.1

```
Private Sub CommandButton1_Click()  
    Dim counter As Integer  
    Do  
        counter = counter + 1  
        Cells(counter, 1) = counter  
    Loop While counter < 10  
End Sub
```

In this example, the program will keep on adding 1 to the preceding counter value as long as the counter value is less than 10. It displays 1 in cells (1,1), 2 in cells(2,1)..... until 10 in cells (10,1).

Example 6.2

```
Private Sub CommandButton1_Click()  
    Dim counter As Integer  
    Do Until counter = 10  
        counter = counter + 1  
        Cells(counter, 1) = 11 - counter  
    Loop  
End Sub
```

Example 6.3

```

Private Sub CommandButton1_Click ()

    Dim counter As Integer
    Do Until counter = 10
        counter = counter + 1
        Cells(counter, 1) = 11 - counter
    Loop
End Sub

```

In this example, the program will keep on adding 1 to the preceding counter value until the counter value reaches 10. It displays 10 in cells (1, 1), 9 in cells (2, 1)..... until 1 in cells (10,1).

Example 6.3

```

Private Sub CommandButton1_Click()
    Dim counter , sum As Integer
    'To set the alignment to center
    Range("A1:C11").Select
    With Selection
        .HorizontalAlignment = xlCenter
    End With

    Cells(1, 1) = "X"
    Cells(1, 2) = "Y"

```

```
Cells(1, 3) = "X+Y"
```

```
Do While counter < 10
```

```
counter = counter + 1
```

```
Cells(counter + 1, 1) = counter
```

```
Cells(counter + 1, 2) = counter * 2
```

```
sum = Cells(counter + 1, 1) + Cells(counter + 1, 2)
```

```
Cells(counter + 1, 3) = sum
```

```
Loop
```

```
End Sub
```

The above program will display the values of X in cells(1,1) to cells(11,1). The values of Y are X^2 and the values are displayed in column 2, i.e. from cells(2,1) to cells(2,11). Finally, it shows the values of X+Y in column 3, i.e. from cells(3,1) to cells(3,11)

Chapter 7

Select Case.....End Select

Normally it is sufficient to use the conditional statement If....Then....Else for multiple options or selections programs. However, if there are too many different cases, the If...Then...Else structure could become too bulky and difficult to debug if problems arise. Fortunately, Visual Basic provides another way to handle complex multiple choice cases, that is, the *Select Case.....End Select* decision structure. The general format of a Select Case...End Select structure is as follow:

```
Select Case variable
    Case value 1
        Statement
    Case value 2
        Statement
    Case value 3
        Statement
    .
    Case Else
    End Select
```

In the following example, the program will process the grades of students according to the marks given.

Example 7.1

```
Private Sub CommandButton1_Click()  
    Dim mark As Single  
    Dim grade As String  
    mark = Cells(1, 1).Value  
    'To set the alignment to center  
    Range("A1:B1").Select  
    With Selection  
        .HorizontalAlignment = xlCenter  
    End With  
    Select Case mark  
        Case 0 To 20  
            grade = "F"  
            Cells(1, 2) = grade  
        Case 20 To 29  
            grade = "E"  
            Cells(1, 2) = grade  
        Case 30 To 39  
            grade = "D"  
            Cells(1, 2) = grade  
        Case 40 To 59  
            grade = "C"  
            Cells(1, 2) = grade  
        Case 60 To 79  
            grade = "B"  
            Cells(1, 2) = grade  
        Case 80 To 100  
            grade = "A"  
            Cells(1, 2) = grade  
        Case Else  
            grade = "Error!"  
            Cells(1, 2) = grade  
    End Select  
End Sub
```

Chapter 8

Excel VBA Objects Part 1–An Introduction

8.1: Objects

Most programming languages today deal with objects, a concept called object oriented programming. Although Excel VBA is not a truly object oriented programming language, it does deal with objects. VBA object is something like a tool or a thing that has certain functions and properties, and can contain data. For example, an *Excel Worksheet* is an object, a *cell* in a worksheet is an object, a *range of cells* is an object, the *font* of a cell is an object, a *command button* is an object, and a *text box* is an object and more.

In order to view the VBA objects, you can insert a number of objects or controls into the worksheet, and click the command button to go into the code window. The upper left pane of the code window contains the list of objects you have inserted into the worksheet; you can view them in the dropdown dialog when you click the down arrow. The right pane represents the events associated with the objects.

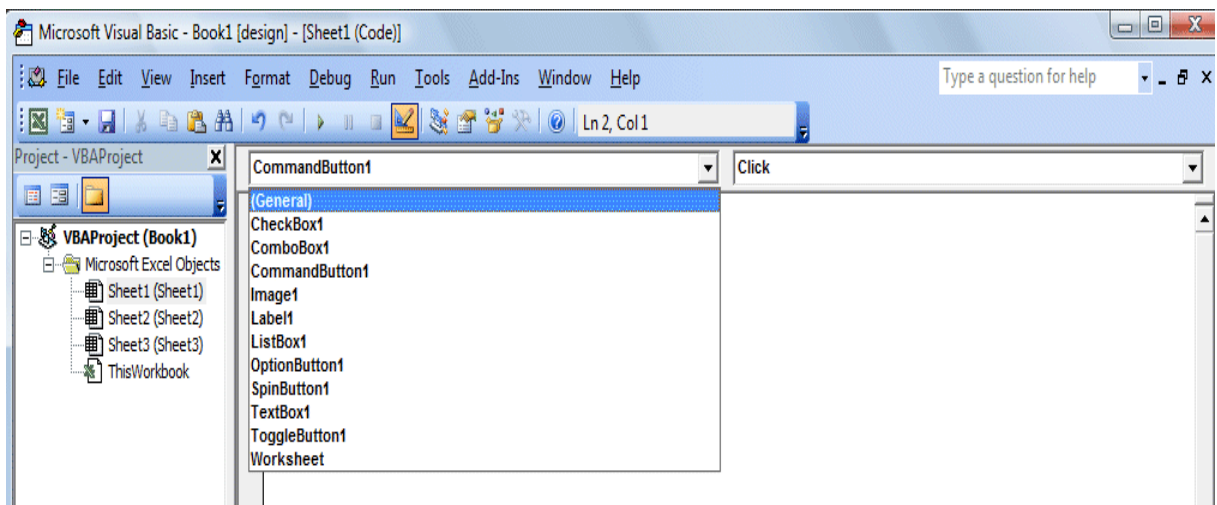


Figure 8.1: Some common Excel VBA objects.

To view all the available objects, you can click on the objects browser in the code window.

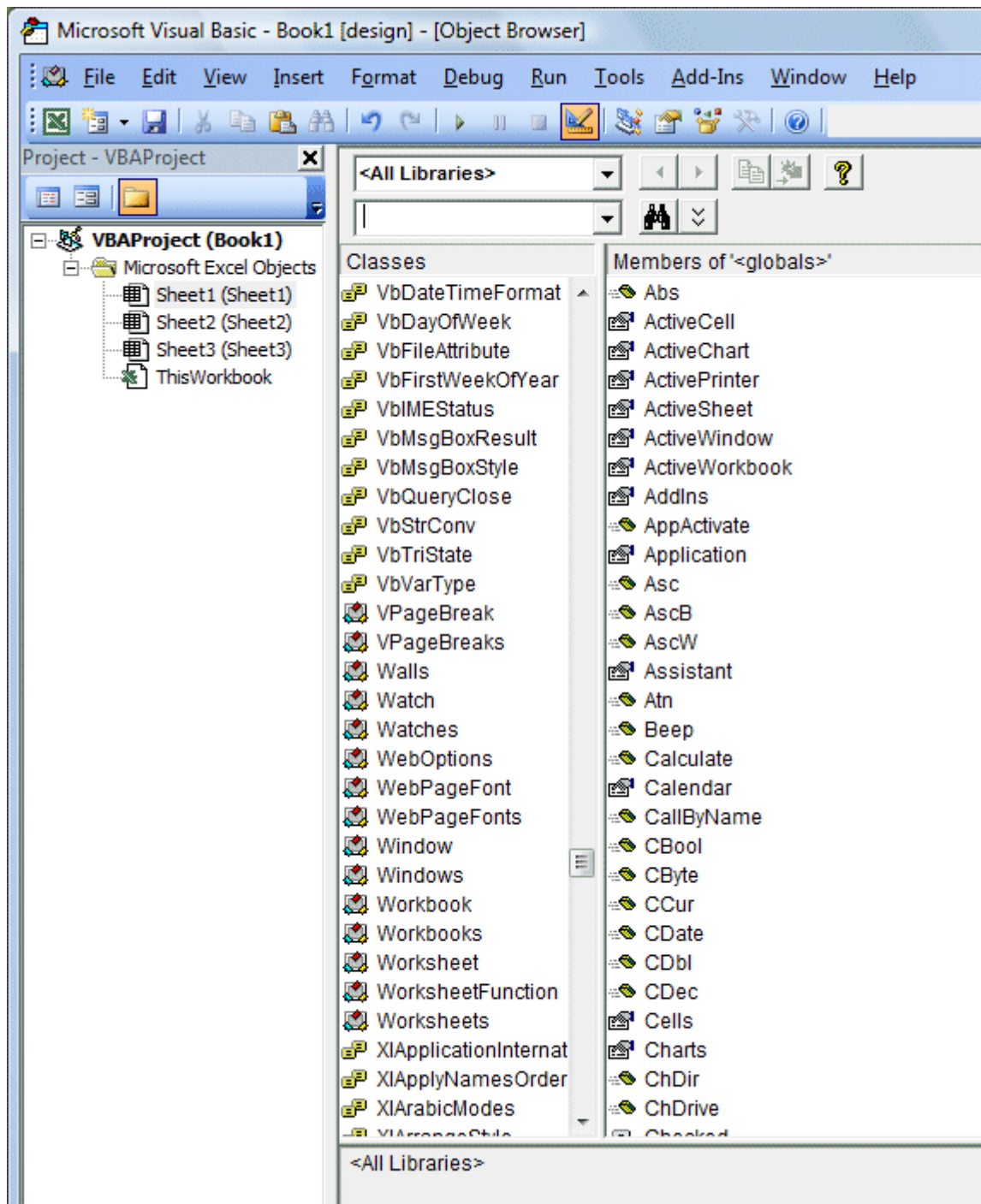


Figure 8.2: Objects browser that lists the entire Excel VBA objects.

8.2: Properties and Methods

8.2.1 Properties

An Excel VBA object has properties and methods. Properties are like the characteristics or attributes of an object. For example, Range is an Excel VBA object and one of its properties is value. We connect an object to its property by a period (a dot or full stop). The following example shows how we connect the property value to the Range object.

Example 8.1

```
Private Sub CommandButton1_Click()  
    Range("A1:A6").Value = 10  
End Sub
```

In this example, by using the value property, we can fill cells A1 to A6 with the value of 10. However, because value is the default property, it can be omitted. So the above procedure can be rewritten as

Example 8.2

```
Private Sub CommandButton1_Click()  
    Range("A1:A6")= 10  
End Sub
```

Cell is also an Excel VBA object, but it is also the property of the range object. So an object can also be a property, it depends on the hierarchy of the objects. Range has higher hierarchy than cells, and interior has lower hierarchy than Cells, and color has lower hierarchy than Interior, so you can write

```
Range("A1:A3").Cells(1, 1).Interior.Color = vbYellow
```

This statement will fill cells (1, 1) with yellow color. Notice that although the Range object specifies a range from A1 to A3, but the cells property specifies only cells(1,1) to be filled with yellow color, it sort of overwrites the range specified by the Range object.

Font is an object which belongs to the Range object. Font has its own properties. For example, Range("A1:A4").Font.Color= vbYellow , the color property of the object Font will fill all the contents from cell A1 to cell A4 with yellow color.

Sometime it is not necessary to type the properties, Excel VBA IntelliSense will display a drop-down list of proposed properties after you type a period at the end of the object name. You can then select the property you want by double clicking it or by highlighting it then press the Enter key. The IntelliSense drop-down is shown in Figure 8.3.

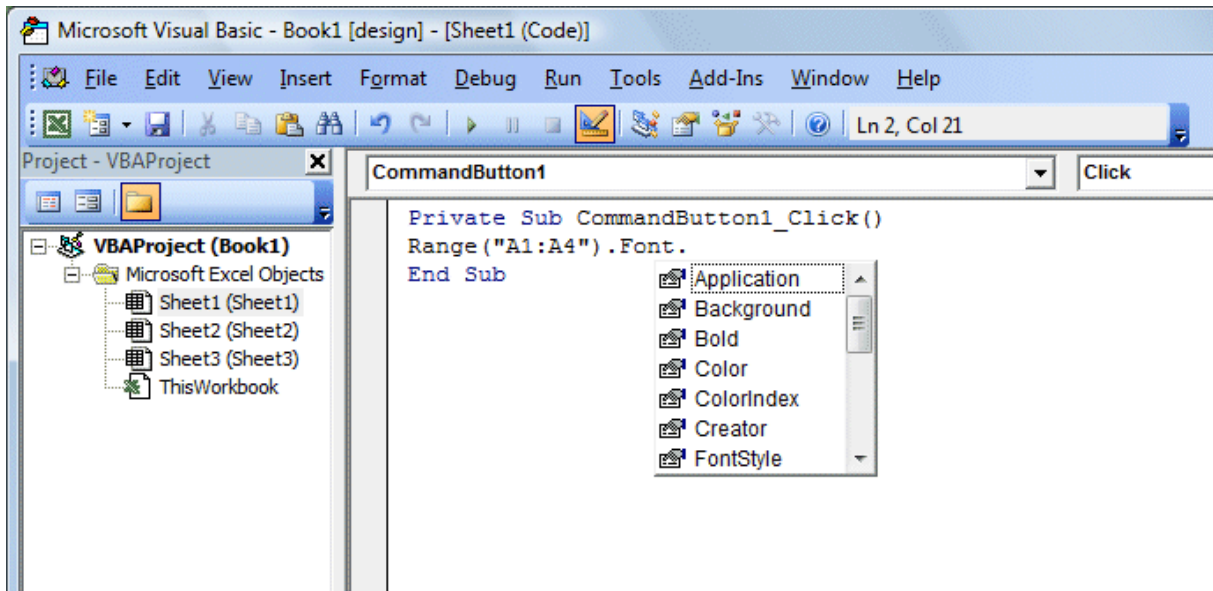


Figure 8.3: The IntelliSense Drop-Down List

Count is also a property of the Range object. It shows the number of cells in the specified range. For example, Range ("A1:A10").Count will return a value of 10. In order to display the number of cells returned, you can write the following code.

Example 8.3

```
Private Sub CommandButton1_Click()  
  
    Dim tcount As Integer  
    tcount = Range("A1:A6").count  
    Range("A10") = tcount  
  
End Sub
```

8.2.2 Methods

Besides having properties, Excel VBA objects also have methods. Methods normally do something or perform certain operations. For example, *ClearContents* is a method of the range object. It clears the contents of a cell or a range of cells. You can write the following code to clear the contents:

Example 8.4

```
Private Sub CommandButton1_Click()  
  
    Range("A1:A6").ClearContents  
  
End Sub
```

You can also let the user select his own range of cells and clear the contents by using the InputBox function, as shown in Example 8.5

Example 8.5

```
Private Sub CommandButton1_Click()  
    Dim, selectedRng As String  
    selectedRng = InputBox("Enter your range")  
    Range(selectedRng).ClearContents  
End Sub
```

In order to clear the contents of the entire worksheet, you can use the following code:

```
Sheet1.Cells.ClearContents
```

However, if you only want to clear the formats of an entire worksheet, you can use the following syntax:

```
Sheet1.Cells.ClearFormats
```

To select a range of cells, you can use the *Select* method. This method selects a range of cells specified by the Range object. The syntax is

```
Range("A1:A5").Select
```

Example 8.6: The code to select a range of cells

```
Private Sub CommandButton1_Click()  
    Range("A1:A5").Select  
End Sub
```

Example 8.7

This example allows the user to specify the range of cells to be selected.

```
Private Sub CommandButton1_Click()  
    Dim selectedRng As String  
    selectedRng = InputBox("Enter your range")  
    Range(selectedRng).Select  
End Sub
```

To deselect the selected range, we can use the *Clear* method.

```
Range("C1Rj:CmRn").Clear
```

Example 8.8

In this example, we insert two command buttons, the first one is to select the range and the second one is to deselect the selected range.

```
Private Sub CommandButton1_Click()  
    Range("A1:A5").Select  
End Sub  
  
Private Sub CommandButton2_Click()  
    Range("A1:A5").Clear  
End Sub
```


Instead of using the *Clear* method, you can also use the *ClearContents* method.

Another very useful method is the *Autofill* method. This method performs an autofill on the cells in the specified range with a series of items including numbers, days of week, months of year and more. The format is

`Expression.AutoFill (Destination, Type)`

Where *Expression* can be an object or a variable that returns an object. *Destination* means the required *Range* object of the cells to be filled. The destination must include the *source range*. *Type* means type of series, such as days of week, months of year and more. The *AutoFill* type constant is something like *xlFillWeekdays*, *XlFillDays*, *XlFillMonths* and more.

Example 8.9:

```
Private Sub CommandButton1_Click()  
    Range("A1")=1  
    Range("A2")=2  
    Range("A1:A2").AutoFill Destination:=Range("A1:A10")  
End Sub
```

In this example, the source range is A1 to A2. When the user clicks on the command button, the program will first fill cell A1 with 1 and cell A2 with 2, and then automatically fill the Range A1 to A10 with a series of numbers from 1 to 10.

Example 8.10

```

Private Sub CommandButton1_Click()
    Cells(1, 1).Value = "Monday"
    Cells(2, 1).Value = "Tuesday"
    Range("A1:A2").AutoFill Destination:=Range("A1:A10"), Type:=xlFillDays
End Sub

```

In this example, when the user clicks on the command button, the program will first fill cell A1 with “Monday” and cell A2 with “Tuesday”, and then automatically fills the Range A1 to A10 with the days of a week.

Example 8.11

This example allows the user to select the range of cells to be automatically filled using the Autofill method. This can be achieved with the use of the InputBox. Since each time we want to autofill a new range, we need to clear the contents of the entire worksheet using the *Sheet1.Cells.ClearContents* statement.

```

Private Sub CommandButton1_Click()
    Dim selectedRng As String
    Sheet1.Cells.ClearContents
    selectedRng = InputBox("Enter your range")
    Range("A1") = 1
    Range("A2") = 2
    Range("A1:A2").AutoFill
    Destination:=Range(selectedRng)
End Sub

```

Chapter 9

Excel VBA Objects Part 2 –The Workbook Object

In the previous chapter, we have learned about Excel VBA objects and their properties and methods. In this chapter, we shall learn specifically about the *Workbook* object as it is one of the most important Excel VBA objects. It is also at the top of the hierarchy of the Excel VBA objects. We will deal with properties and methods associated the *Workbook* object.

9.1 The Workbook Properties.

When we write VBA code involving the *Workbook* object, we use *Workbooks*. The reason is that we are dealing with a collection of workbooks most of the time, so using *Workbooks* enables us to manipulate multiple workbooks at the same time.

When will deal with multiple workbooks, we can use indices to denote different workbooks that are open, using the syntax *Workbooks (i)*, where *i* is an index. For example, *Workbooks (1)* denotes *Workbook1*, *Workbooks (2)* denotes *Workbook2* and more.

A workbook has a number of properties. Some of the common properties are *Name*, *Path* and *FullName* Let's look at the following example:

Example 9.1

```
Private Sub CommandButton1_Click()  
    MsgBox Workbooks(1).Name  
End Sub
```

The program will cause a message dialog box to pop up and displays the first workbook name, i.e. `workbook_object1.xls` as shown in Figure 9.1 below:

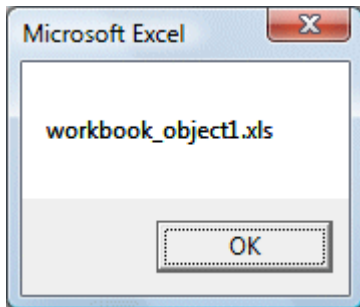


Figure 9.1: The name of the Excel workbook

If we have only one open workbook, we can also use the syntax `ThisWorkbook` in place of `Workbook (1)`, as follows:

```
Private Sub CommandButton1_Click ()
    MsgBox ThisWorkbook.Name
End Sub
```

Example 9.2

```
Private Sub CommandButton1_Click ()
    MsgBox ThisWorkbook.Path
End Sub
```

Or you can use the following code

```
Private Sub CommandButton1Click ()
    MsgBox Workbooks ("workbook_object1.xls").Path
End Sub
```

The output is shown below:

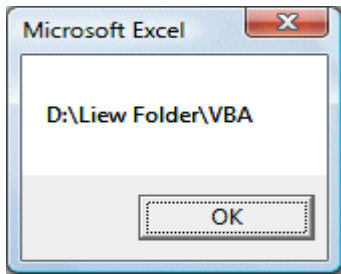


Figure 9.2: The path of the opened workbook

Example 9.3

This example will display the path and name of the opened workbook. The code is:

```
Private Sub CommandButton1_Click ()  
    MsgBox ThisWorkbook.FullName  
End Sub
```

Or

```
Private Sub CommandButton1Click()  
    MsgBox Workbooks("workbook_object1.xls").Fullname  
End Sub
```

The output is shown in Figure 9.3.

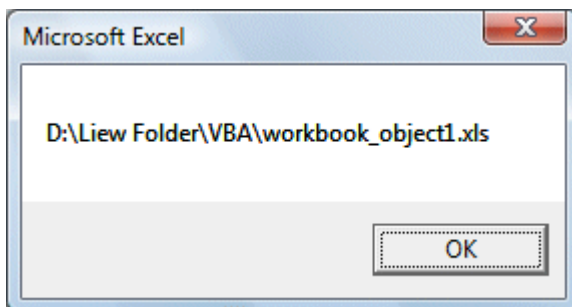


Figure 9.3

9.2 The Workbook Methods

There are a number of methods associated with the workbook object. These methods are *Save*, *SaveAs*, *Open*, *Close* and more.

Example 9.4

In this example, when the user clicks on the command button, it will open up a dialog box and ask the user to specify a path and type in the file name, and then click the save button, not unlike the standard windows SaveAs dialog, as shown in Figure 9.5.

```
Private Sub CommandButton1_Click()
    fName = Application.GetSaveAsFilename
    ThisWorkbook.SaveAs Filename:=fName
End Sub
```

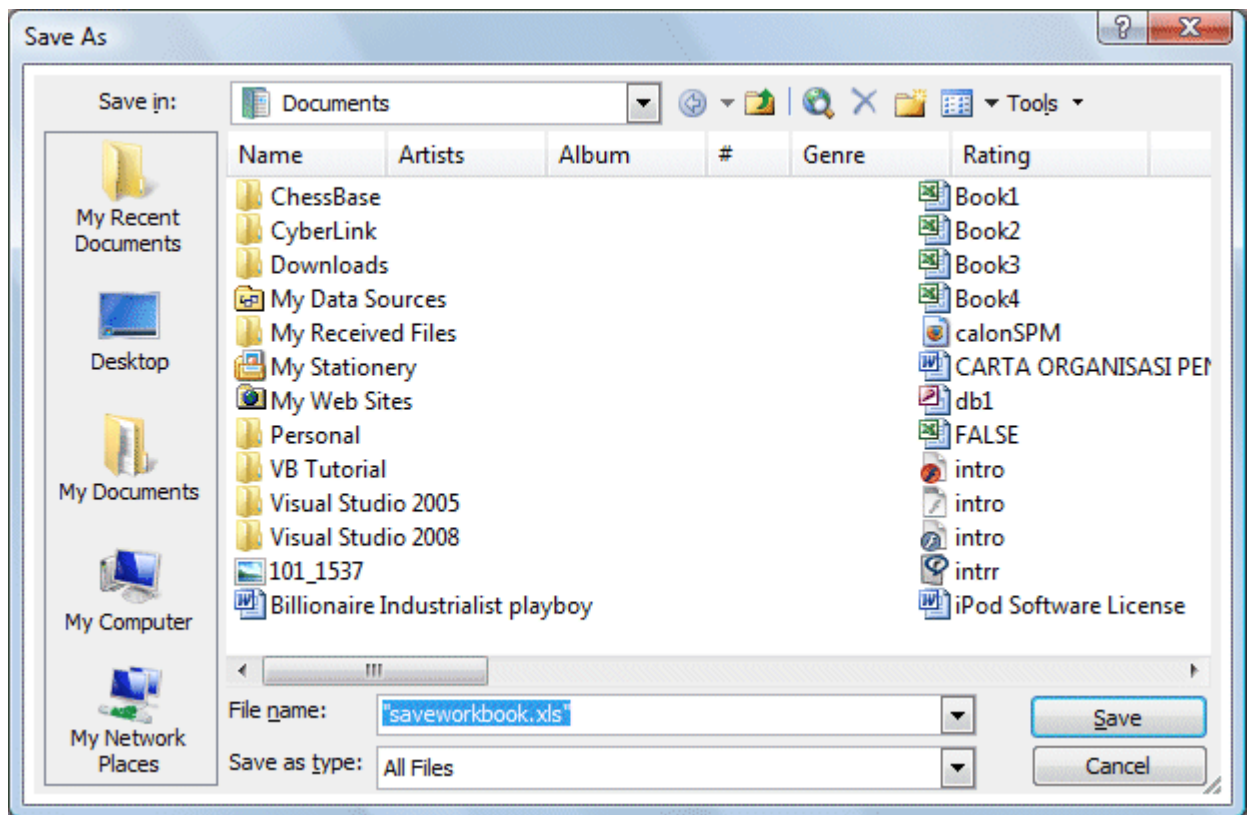


Figure 9.4: The SaveAs dialog

Another method associated with the workbook object is *open*. The syntax is

```
Workbooks.Open ("File Name")
```

Example 9.5

In this example, when the user click on the command button, it wil open the file `workbook_object1.xls` under the path `C:\Users\liewvk\Documents\`

```
Private Sub CommandButton1_Click()  
    Workbooks.Open ("C:\Users\liewvk\Documents\workbook_object1.xls")  
End Sub
```

The *close* method is the command that closes a workbook. The syntax is

```
Workbooks (i).Close
```

Example 9.6

In this example, when the user clicks the command button, it will close `Workbooks (1)`.

```
Private Sub CommandButton1_Click()  
    Workbooks (1).Close  
End Sub
```

Chapter 10

Excel VBA Objects Part 3 –The Worksheet Object

10.1 The Worksheet Properties

Similar to the Workbook Object, the Worksheet has its own set of properties and methods. When we write VBA code involving the Worksheet object, we use *Worksheets*. The reason is that we are dealing with a collection of worksheets most of the time, so using *Worksheets* enables us to manipulate multiple worksheets at the same time.

Some of the common properties of the worksheet are name, *count*, *cells*, *columns*, *rows* and *columnWidth*.

Example 10.1

```
Private Sub CommandButton1_Click()  
    MsgBox Worksheets(1).Name  
End Sub
```

The above example will cause a pop-up dialog that displays the worksheet name as sheet 1, as shown below:

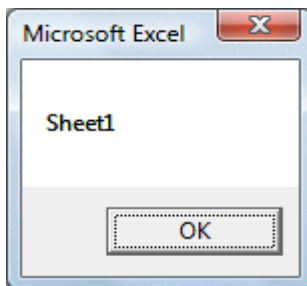


Figure 10.1

The count property returns the number of worksheets in an opened workbook.

Example 10.2

```
Private Sub CommandButton1_Click()  
    MsgBox Worksheets.Count  
End Sub
```

The output is shown in Figure 10.2.

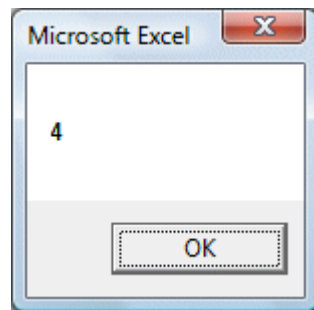


Figure 10.2

Example 10.3

The count property in this example will return the number of columns in the worksheet.

```
Private Sub CommandButton1_Click()  
    MsgBox Worksheets(1).Columns.Count  
End Sub
```

* It is suffice to write *MsgBox Columns.Count* as the worksheet is considered the active worksheet. The output is shown below:

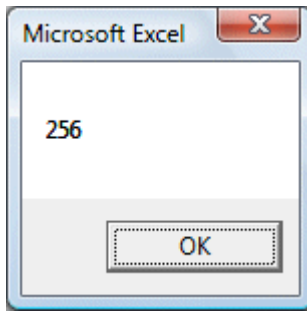


Figure 10.3

Example 10.4

The count property in this example will return the number of rows in the worksheet.

```
Private Sub CommandButton1_Click()  
    MsgBox Worksheets(1).Rows.Count  
End Sub
```

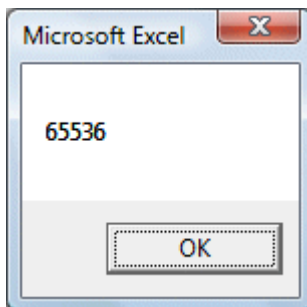


Figure 10.4

10.2 The Worksheet Methods

Some of the worksheet methods are *add*, *delete*, *select*, *SaveAs*, *copy*, *paste* and more.

Example 10.5

In this example, when the user clicks the first command button, it will add a new sheet to the workbook. When the user clicks the second command button, it will delete the new worksheet that has been added earlier.

```
Private Sub CommandButton1_Click()  
    Worksheets.Add  
End Sub  
Private Sub CommandButton2_Click()  
    Worksheets(1).Delete  
End Sub
```

Example 10.6

The select method associated with worksheet let the user select a particular worksheet. In this example, worksheet 2 will be selected.

```
Private Sub CommandButton1_Click()  
    'Worksheet 2 will be selected  
    Worksheets(2).Select  
End Sub
```

The select method can also be used together with the Worksheet's properties Cells, Columns and Rows as shown in the following examples.

Example 10.5

```
Private Sub CommandButton1_Click()  
    'Cell A1 will be selected  
    Worksheets (1).Cells (1).Select  
End Sub
```

Example 10.6

```
Private Sub CommandButton1_Click()  
    'Column 1 will be selected  
    Worksheets (1).Columns (1).Select  
End Sub
```

Example 10.7

```
Private Sub CommandButton1_Click()  
    'Row 1 will be selected  
    Worksheets (1).Rows (1).Select  
End Sub
```

Excel VBA also allows us to write code for copy and paste. Let's look at the following Example:

Example 10.8

```
Private Sub CommandButton1_Click()
```

```
'To copy the content of a cell 1
```

```
    Worksheets(1).Cells(1).Select
```

```
    Selection.Copy
```

```
End Sub
```

```
Private Sub CommandButton2_Click()
```

```
'To paste the content of cell 1 to cell 2
```

```
    Worksheets(1).Cells(2).Select
```

```
    ActiveSheet.Paste
```

```
End Sub
```

Chapter 11

Excel VBA Objects Part 4–The Range Object

11.1 The Range Properties

As an Excel VBA object, the range object is ranked lower than the worksheet object in the hierarchy. We can also say that worksheet is the parent object of the range object. Therefore, the Range object also inherits the properties of the worksheet object. Some of the common properties of the range object are Columns, Rows, Value and Formula

11.1.1 Formatting Font

There are many Range properties that we can use to format the font in Excel. Some of the common ones are Bold, Italic, Underline, Size, Name, FontStyle, ColorIndex and Color. These properties are used together with the Font property.

The Bold, Italic, Underline and FontStyle properties are used to format the font style. The syntax for using Bold, Italic and Underline are similar, as shown below:

```
Range ("YourRange").Font.Bold=True
```

```
Range ("YourRange").Font.Italic=True
```

```
Range ("YourRange").Font.Underline=True
```

The FontStyle property can actually be used to replace all the properties above to achieve the same formatting effects. The syntax is as follows:

```
Range ("YourRange").Font.FontStyle="Bold Italic Underline"
```

The Name property is used to format the type of font you wish to display in the designated range. The syntax is as follows:

```
Range("A1:A3").Font.Name = "Time News Roman"
```

The formatting code is illustrated in Example 11.1.

Example 11.1

```
Private Sub CommandButton1_Click()  
    Range("A1:A3").Font.Bold = True  
    Range("A1:A3").Font.Italic = True  
    Range("A1:A3").Font.Underline = True  
    Range("A1:A3").Font.Size = 20  
    Range("A1:A3").Font.FontStyle = "Bold Italic Underline"  
    Range("A1:A3").Font.Name = "Time News Roman"  
  
End Sub
```

The Font and ColorIndex properties are used together to format the font color. You can also use the color property to display the font color,

Example 11.2

```
Private Sub CommandButton2_Click()
    Range("A4:B10").Font.ColorIndex = 4
End Sub
```

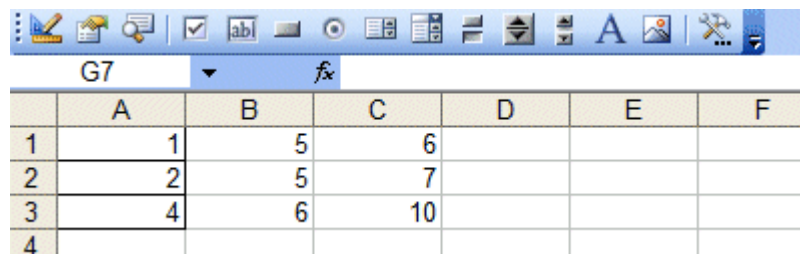
.In this example, the font color will be displayed in green (Corresponding with ColorIndex =4):

```
Range("A4:B10").Font.Color = VbRed
```

Example 11.2

```
Private Sub CommandButton1_Click()
    Range("A1:B3").Columns(3).Formula = "=A1+B1"
End Sub
```

In this example, the formula A1+B1 will be copied down column 3 (column C) from cell C1 to cell C3. The program automatically sums up the corresponding values down column A and column B and displays the results in column C, as shown in Figure 11.1.



	A	B	C	D	E	F
1	1	5	6			
2	2	5	7			
3	4	6	10			
4						

Figure 11.1

The above example can also be rewritten and produces the same result as below:

```
Range("A1:B3").Columns(3).Formula = "=Sum(A1:B1)"
```

There are many formulas in Excel VBA which we can use to simplify and speed up complex calculations. The formulas are categorized into Financial, Mathematical, Statistical, Date ,Time and others. For example, in the statistical category, we have Average (Mean), Mode and Median

Example 11.3

In this example, the program computes the average of the corresponding values in column A and column B and displays the results in column C. For example, the mean of values in cell A1 and Cell B1 is computed and displayed in Cell C1. Subsequent means are automatically copied down Column C until cell C3.

```
Private Sub CommandButton1_Click()
    Range("A1:B3").Columns(3).Formula = "=Average(A1:B1)"
End Sub
```

Example 11.4: Mode

In this example, the program computes the mode for every row in the range A1:E4 and displays them in column F. It also makes the font bold and red in color, as shown in Figure 11.2.

```
Private Sub CommandButton1_Click()
    Range("A1:E4").Columns(6).Formula = "=Mode(A1:E1)"
    Range("A1:E4").Columns(6).Font.Bold = True
    Range("A1:E4").Columns(6).Font.ColorIndex = 3
End Sub
```

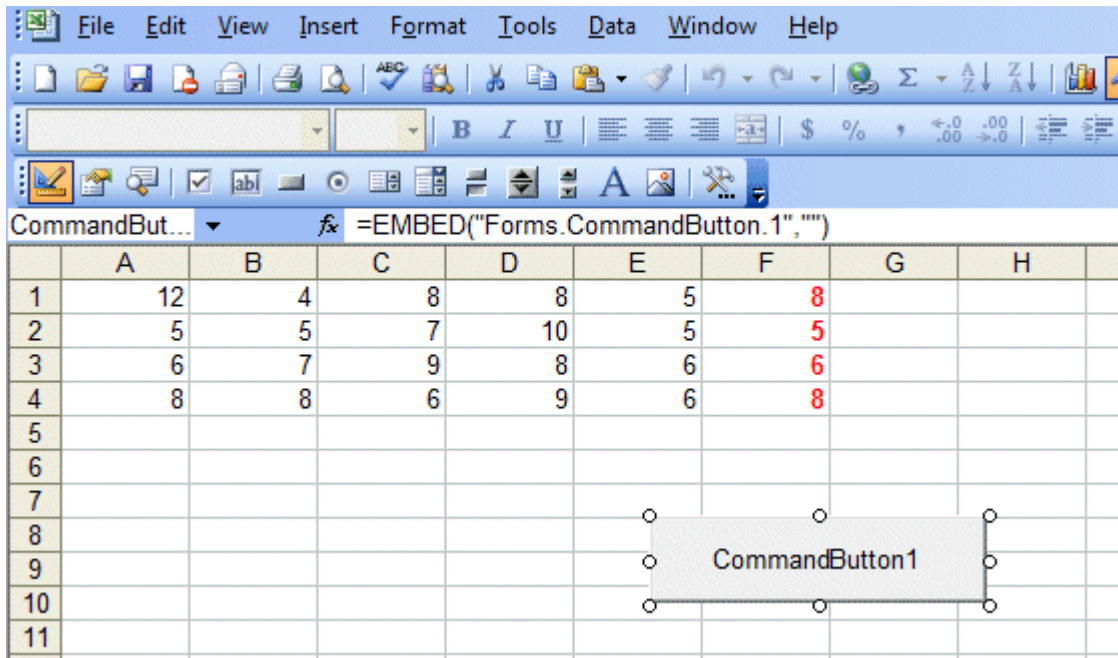


Figure 11.2: Mode for each row displayed in column F

Example 11.5: Median

In this example, the program computes the median for every row in the range A1:E4 and displays them in column F. It also makes the font bold and red in color, as shown in Figure 11.3.

```
Private Sub CommandButton1_Click()
    Range("A1:E4").Columns(6).Formula = "=Median(A1:E1)"
    Range("A1:E4").Columns(6).Font.Bold = True
    Range("A1:E4").Columns(6).Font.ColorIndex = 3
End Sub
```

	A	B	C	D	E	F	G	H
1	12	4	8	8	5	8		
2	5	5	7	10	5	5		
3	6	7	9	8	6	7		
4	8	8	6	9	6	8		
5								
6								
7								
8								
9								
10								
11								

Figure 11.3: Median for each row displayed in column F

Example 11.6

In this example, the Interior and the Color properties will fill the cells in the range A1:A3 with yellow color.

```
Private Sub CommandButton1_Click()
    Range("A1:A3").Interior.Color = vbYellow
End Sub
```

11.2 The Range Methods

The range methods allow the range object to perform many types of operations. They enable automation and perform customized calculations that greatly speed up otherwise time consuming work if carried out manually.

There are many range methods which we can use to automate our works. Some of the methods are Autofill, Clear, ClearContents, Copy, cut, PasteSpecial, and Select.

11.2.1 Autofill Method

This program allows the cells in range A1 to A20 to be filled automatically following the sequence specified in the range A1 to A2. The Destination keyword is being used here.

Example 11.7

```
Private Sub CommandButton1_Click ()  
    Set myRange = Range ("A1:A2")  
    Set targetRange = Range ("A1:A20")  
    myRange.AutoFill Destination: =targetRange  
End Sub
```

11.2.2 Select, Copy and Paste Methods

We use the Select method to select a specified range, copy the values from that range and then paste them in another range, as shown in the following example:

Example 11.9

```
Private Sub CommandButton1_Click ()  
    Range ("C1:C2").Select  
    Selection.Copy  
    Range ("D1:D2").Select  
    ActiveSheet.Paste  
End Sub
```

*We can also use the Cut method in place of Copy in the above example.

11.2.2 Copy and PasteSpecial Methods

The *Copy* and the *PasteSpecial* methods are performed together. The copy method will copy the contents in a specified range and the *PasteSpecial* method will paste the contents into another range. However, unlike the paste method, which just pastes the values into the target cells, the *PasteSpecial* method has a few options. The options are *PasteValues*, *PasteFormulas*, *PasteFormats* or *PasteAll*. The *PasteValues* method will just paste the values of the original cells into the targeted cells while the *PasteFormulas* will copy the formulas and update the values in the targeted cells accordingly.

Example 11.10

```
Private Sub CommandButton1_Click()  
    Range("C1:C2").Copy  
    Range("D1:D2").PasteSpecial Paste:=xlPasteValues  
    Range("E1:E2").PasteSpecial Paste:=xlPasteFormulas  
    Range("F1:F2").PasteSpecial Paste:=xlPasteFormats  
    Range("G1:G2").PasteSpecial Paste:=xlPasteAll  
End Sub
```

The output is displayed in Figure 11.4. The original values are pasted to the range D1:D2 while the formula is updated in the range E1:E2 but not the formats. The original formats for the font are bold and red. The formats are reflected in range F1:F2 but the formulas were not pasted there. Lastly, everything is copied over to the range G1:G2.

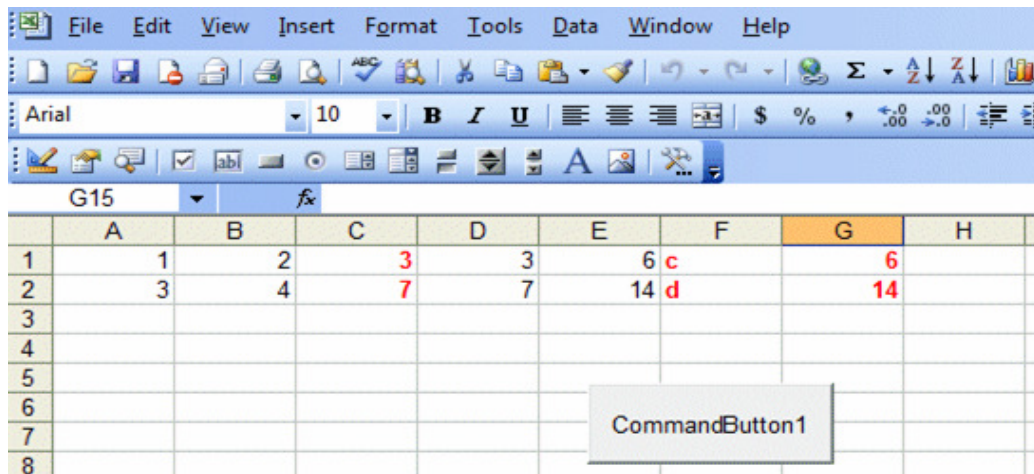


Figure 11.4

We can also modify the code above and paste them according to the Paste Values option and the Paste Formulas option, as shown in Example 11.10.

Example 11.11

```
Private Sub CommandButton1_Click()
    Range("C1:C2").Select
    Selection.Copy
    Range("D1:D2").PasteSpecial Paste:=xlPasteValues
    Range("E1:E2").PasteSpecial Paste:=xlPasteFormulas
End Sub
```

Chapter 12

Working with Excel VBA Controls

Excel VBE provides a number of controls that can be used to perform certain tasks by writing VBA code for them. These controls are also known as Active-X controls. As these controls are Excel VBA objects, they have their own properties, methods and events. They can be found on the Excel Control Toolbox, as shown in the diagram below:

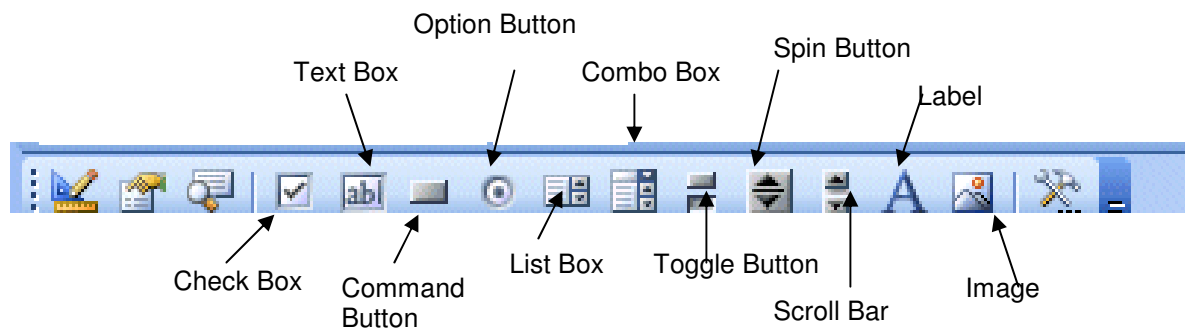


Figure 12.1: Excel VBA Controls

12.1 Check Box

The Check box is a very useful control in Excel VBA. It allows the user to select one or more items by checking the checkbox or checkboxes concerned. For example, you may create a shopping cart where the user can click on checkboxes that correspond to the items they intend to buy, and the total payment can be computed at the same time.

One of most important properties of the check box is `Value`. If the checkbox is selected or checked, the value is `true`, whilst if it is not selected or unchecked, the `Value` is `False`.

The usage of check box is illustrated in Example 12.1

Example 12.1

In this example, the user can choose to display the sale volume of one type of fruits sold or total sale volume. The code is shown in next page.

```
Private Sub CommandButton1_Click()  
    If CheckBox1.Value = True And CheckBox2.Value = False  
    Then  
        MsgBox "Quantity of apple sold is" & Cells(2, 2).Value  
    ElseIf CheckBox2.Value = True And CheckBox1.Value = False  
    Then  
        MsgBox "Quantity of orange sold is " & Cells(2, 3).Value  
    Else  
        MsgBox "Quantity of Fruits sold is" & Cells(2, 4).Value  
    End If  
End Sub
```

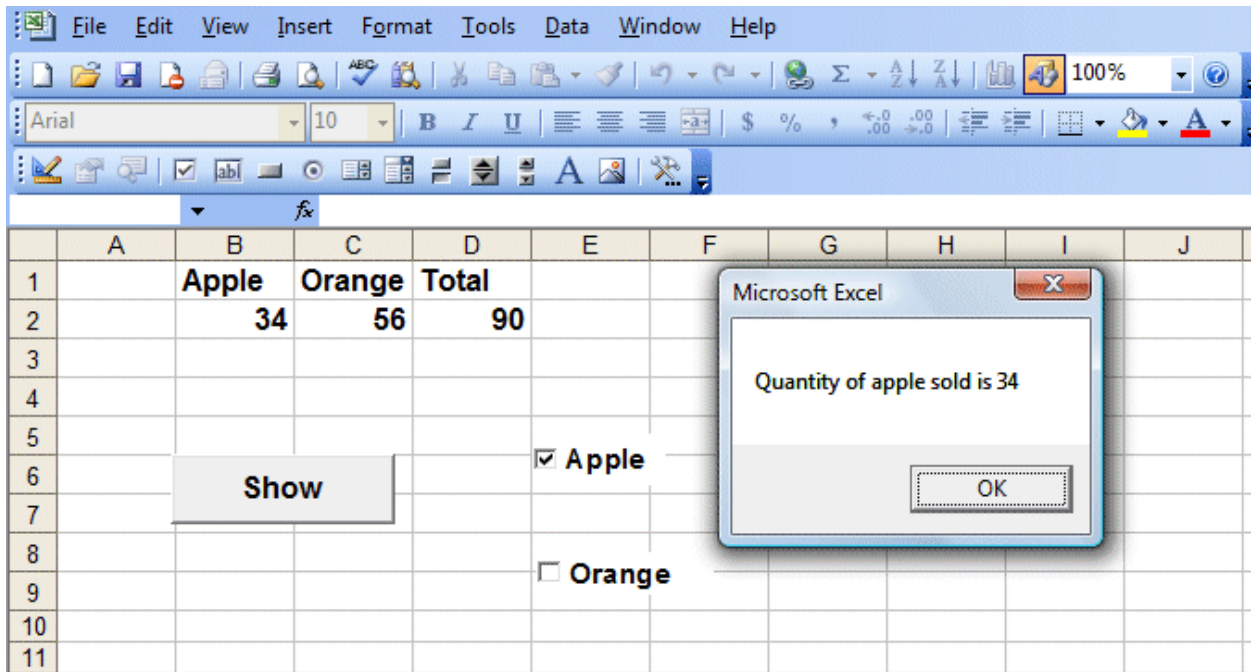



Figure 12.2: Check Boxex

12.2 Text Box

The Text Box is the standard Excel VBA control for accepting input from the user as well as to display the output. It can handle string (text) and numeric data but not images.

Example 12.2

In this example, we inserted two text boxes and display the sum of numbers entered into the two textboxes in a message box. The *Val* function is used to convert string into numeric values because the textbox treats the number entered as a string.

```
Private Sub CommandButton1_Click ()  
    Dim x As Variant, y As Variant, z As Variant  
    x = TextBox1.Text  
    y = TextBox2.Text  
    z = Val(x) + Val(y)  
    MsgBox "The Sum of " & x & " and " & y & " is " & z  
End Sub
```

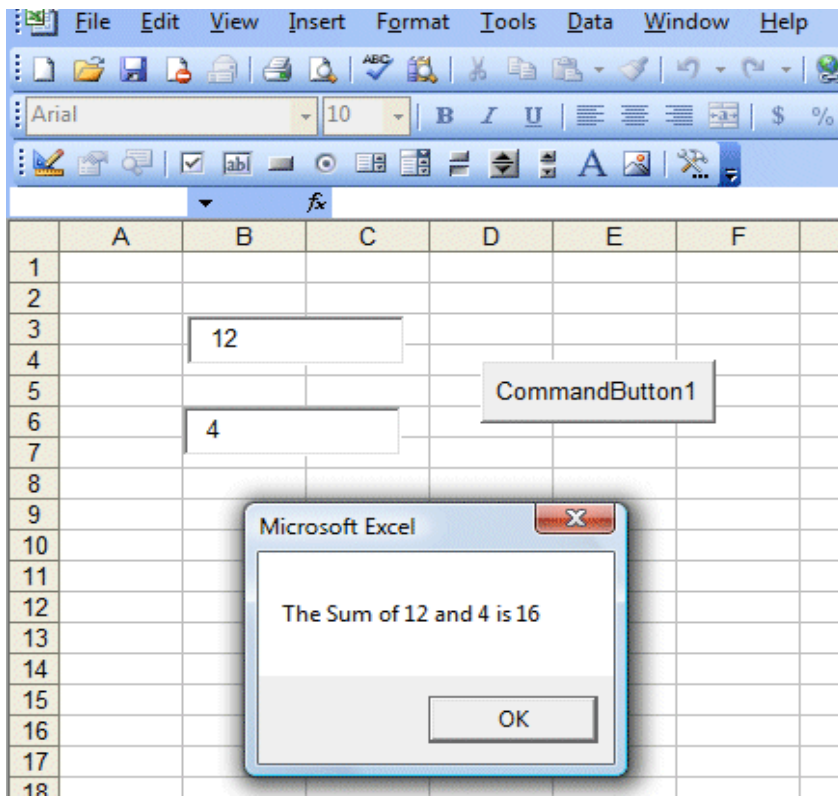


Figure 12.3: Text Boxes

12.3 Option Button

The option button control also lets the user select one of the choices. However, two or more option buttons must work together because as one of the option buttons is selected, the other option button will be deselected. In fact, only one option button can be selected at one time. When an option button is selected, its value is set to "True" and when it is deselected; its value is set to "False".

Example 12.3

This example demonstrates the usage of the option buttons. In this example, the Message box will display the option button selected by the user. The output interface is shown in Figure 12.4.

```
Private Sub OptionButton1_Click ()  
    MsgBox "Option 1 is selected"  
End Sub  
  
Private Sub OptionButton2_Click()  
    MsgBox "Option 2 is selected"  
End Sub  
  
Private Sub OptionButton3_Click()  
    MsgBox "Option 3 is selected"  
End Sub
```

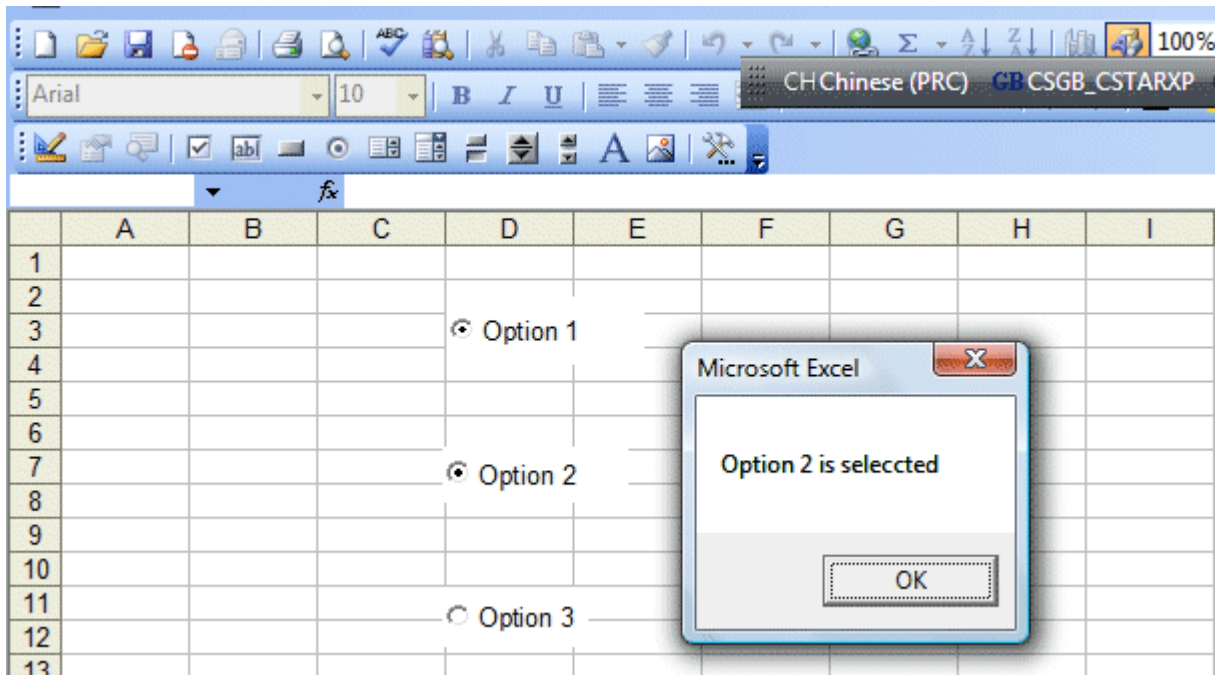


Figure 12.4: The Option Buttons

Example 12.4

In this example, **If ...Then....Elseif** statements are used to control the action when an option button is being selected, i.e., changing the background color of the selected range.

```

Private Sub CommandButton1_Click()
    If OptionButton1.Value = True Then
        Range("A1:B10").Interior.Color = vbRed
    ElseIf OptionButton2.Value = True Then
        Range("A1:B10").Interior.Color = vbGreen
    ElseIf OptionButton3.Value = True Then
        Range("A1:B10").Interior.Color = vbBlue
    End If
End Sub
  
```

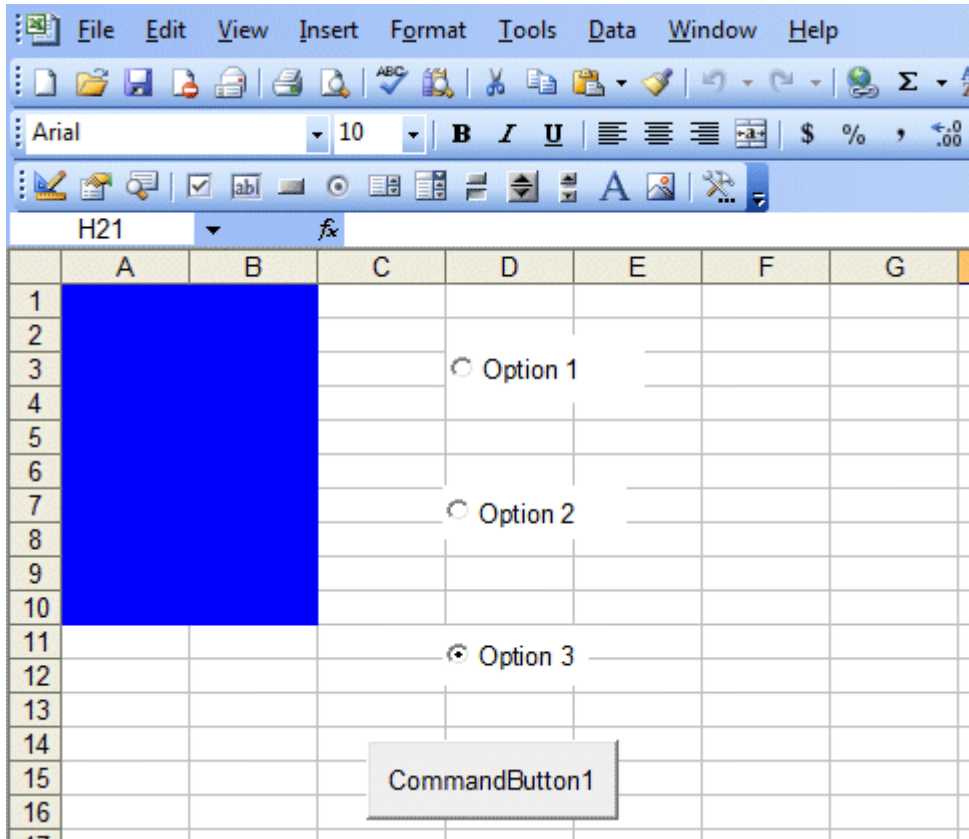


Figure 12.5: Using Option Buttons

Example 12.5

In this example, the program will change the font color of the item selected.

```
Private Sub OptionButton1_Click()
    Dim i As Integer
    For i = 1 To 12
        If Cells(i, 2) = "apple" Then
            Cells(i, 2).Font.Color = vbGreen
        End If
    Next
End Sub
```

```

End Sub

Private Sub OptionButton2_Click()
    For i = 1 To 12
        If Cells(i, 2) = "orange" Then
            Cells(i, 2).Font.Color = vbRed
        End If
    Next
End Sub

```

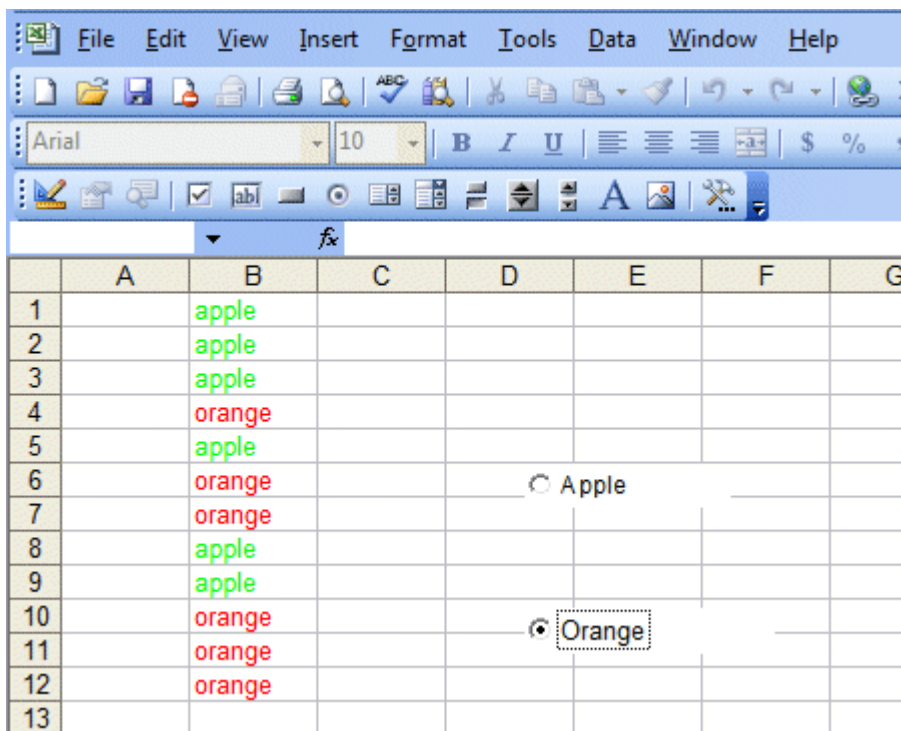


Figure 12.6: Changing font color

12.4 List Box

The function of the List Box is to present a list of items where the user can click and select the items from the list. To add items to the list, we can use the *AddItem* method.

To clear all the items in the List Box, you can use the *Clear* method. The usage of Additem method and the Clear method is shown Example 12.6.

Example 12.6

```
Private Sub CommandButton1_Click()  
    For x = 1 To 10  
        ListBox1.AddItem "Apple"  
    Next  
End Sub  
Private Sub CommandButton2_Click()  
    For x = 1 To 10  
        ListBox1.Clear  
    Next  
End Sub
```

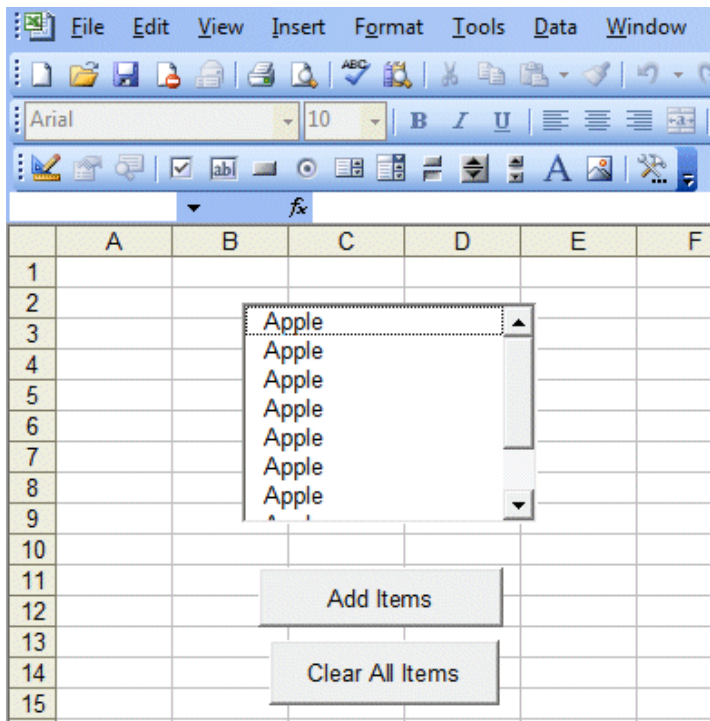


Figure 12.7

12.5 Combo Box

The function of the Combo Box is also to present a list of items where the user can click and select the items from the list. However, the user needs to click on the small arrowhead on the right of the combo box to see the items which are presented in a drop-down list. In order to add items to the list, you can also use the *AddItem* method.

Example 12.7

```
Private Sub CommandButton1_Click()  
    ComboBox1.Text = "Apple"  
    For x = 1 To 10  
        ComboBox1.AddItem "Apple"  
    Next  
End Sub  
Private Sub CommandButton2_Click()  
    ComboBox1.Clear  
End Sub
```

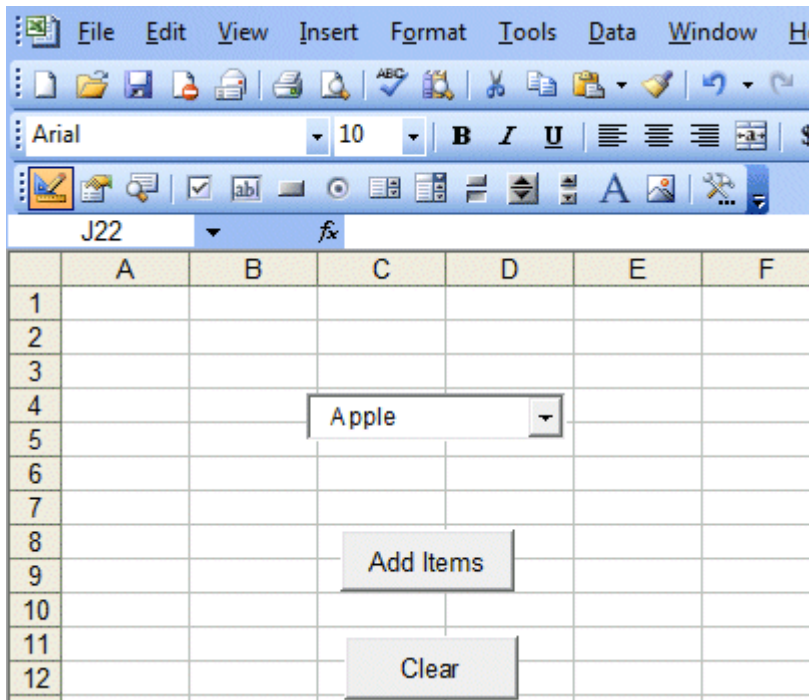



Figure 12.8

12.6 Toggle Button

Toggle button lets the user switches from one action to another alternatively. When the Toggle button is being depressed, the value is true and when it is not depressed, the value is false. By using the If and Else code structure, we can thus switch from one action to another by pressing the toggle button repeatedly.

Example 12.8

In this example, the user can toggle between apple and orange as well as font colors.

```
Private Sub ToggleButton1_Click ()
    If ToggleButton1.Value = True Then
        Cells (1, 1) = "Apple"
        Cells (1, 1).Font.Color = vbRed
    Else
        Cells (1, 1) = "Orange"
        Cells (1, 1).Font.Color = vbBlue
    End If
End Sub
```

Chapter 13

VBA Procedures Part 1-Functions

13.1 The Concept of Functions

A VBA procedure is a block of code that performs certain tasks. We have actually learned about VBA procedures in our previous chapters, but all of them are event procedures. Event procedures are VBA programs that are associated with VBA objects such as command buttons, checkboxes, and radio buttons. However, we can also create procedures that are independent from the event procedures. They are normally called into the event procedures to perform certain tasks. There are two types of the aforementioned procedures, namely Functions and Sub Procedures. In this chapter, we will discuss functions. We will deal with Sub Procedures in the next chapter.

13.2 Types of Functions

There are two types of Excel VBA functions; the built-in functions and the user-defined functions. We can use built-in functions in Excel for automatic calculations. Some of the Excel VBA built-in functions are Sum, Average, Min (to find the minimum value in a range), Max (To find the maximum value in a range), Mode, Median and more. However, built-in functions can only perform some basic calculations, for more complex calculations, user-defined functions are often required. User-defined functions are procedures created independently from the event procedures. A Function can receive arguments passed to it from the event procedure and then return a value in the function name. It is usually used to perform certain calculations.

13.3 Writing Function Code

VBA Function begins with a Function statement and ends with an End Function statement. The program structure of a Function is as follows:

```
Function FunctionName (arguments) As DataType
    Statements
End Function
```

In Excel VBA, when you type the Function statement, the End Function statement will automatically appear.

Example 13.1

In this example, we create a function to calculate the area of a rectangle. It comprises two arguments, one of them is to accept the value of width and the other is to accept the value of height. Note that the function Area_Rect is called from the event procedure (clicking the command button) and the values to be passed to the arguments are enclosed in the parentheses.

```
Private Sub CommandButton1_Click()
    Dim a As Variant, b As Variant
    a = InputBox("Enter Width of Rectangle")
    b = InputBox("Enter Height of Rectangle")
    MsgBox "The area of the rectangle is" & Area_Rect(a, b)
End Sub

Function Area_Rect(x As Variant, y As Variant) As Variant
    Area_Rect = x * y
End Function
```

We can also create a user-defined function to be used just as the built-in functions by inserting a module in the Visual Basic Editor and enter the function code there. After creating the function, we can then return to the spreadsheet and use this function as any other built-in functions. To insert the module, click on Tools in the menu bar, select Macro and then click on Visual Basic Editor.

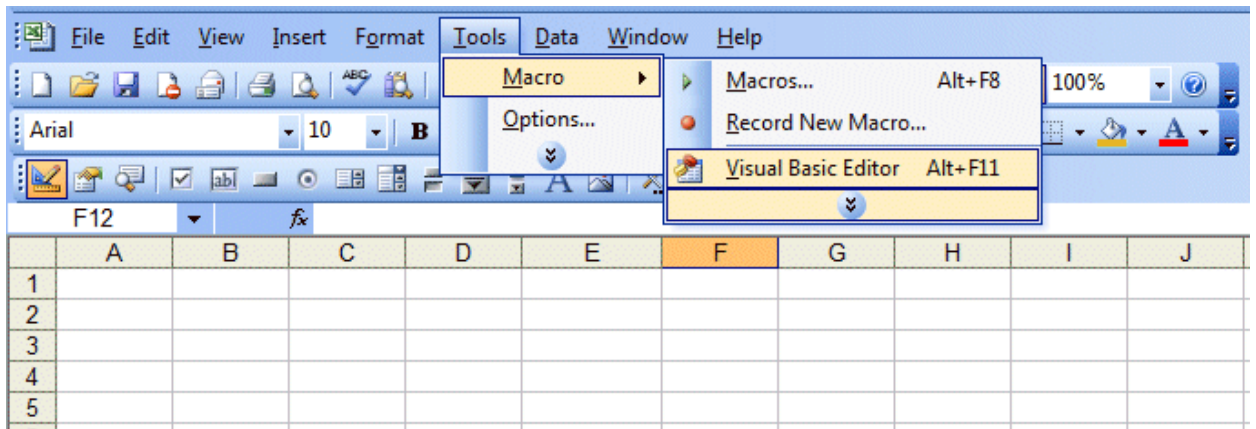


Figure 13.1: Inserting Visual Basic Editor

In the Visual Basic Editor window, insert a module by clicking Insert on the menu bar, and then click on Module.

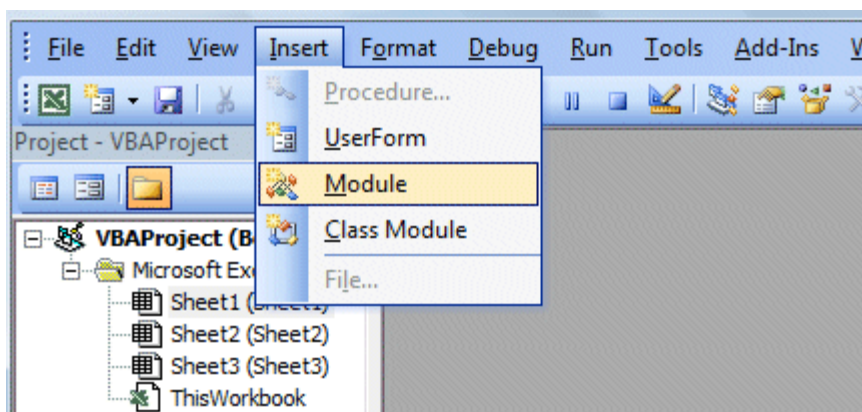


Figure 13.2: Inserting Module

In the module environment, key in the function code for the function Area_Rect , as shown in the diagram below.

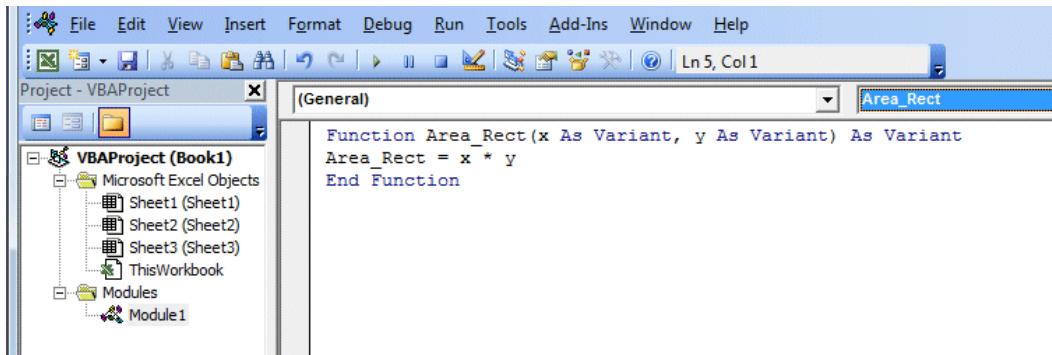


Figure 13.3: Key in the VBA code for the function.

Now, you can return to the Excel spreadsheet and enter the function in any cell. In this Example, the function is entered in cell C1 and the values of width and height are entered in cell A1 and cell B1 respectively. Notice that the value of area is automatically calculated and displayed in cell C1.

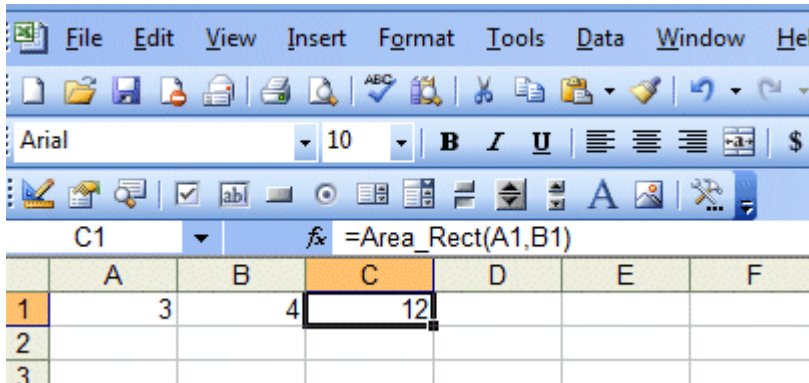


Figure 13.4

The formula can be copied and updated to other cells by using the autofill method, i.e. by dragging the place holder on the bottom right corner of the cell, as shown in Figure 13.5 below.

	A	B	C	D	E
1	3	4	12		
2	4	5	20		
3	5	6	30		
4	6	7	42		
5	7	8	56		
6	8	9	72		
7	9	2	18		
8	9	6	54		
9	5	7	35		
10	3	10	30		
11					
12					
13					
14					
15					

Figure 13.5: using autofill method to update the formula.

The user-defined function not only calculates numerical values, it can also return a string, as shown in Example 13.2 below:

Example 13.2

This program computes the grades of an examination based on the marks obtained. It employed the Select Case.....End Select code structure. The code is shown on next page.

Function grade(mark As Single) As String

Select Case mark

Case 0 To 20

grade = "F"

Case 20 To 29

grade = "E"

Case 30 To 39

grade = "D"

Case 40 To 59

grade = "C"

Case 60 To 79

grade = "B"

Case 80 To 100

grade = "A"

Case Else

grade = "Error!"

End Select

End Function

In the Excel spreadsheet environment, key in the marks in column A and key in the grade function in column B. Notice that the grades will be automatically updated in column B as marks are entered or updated in column A, as shown in Figure 13.6

	A	B	C	D
1	15	F		
2	30	D		
3	45	C		
4	67	B		
5	80	A		
6	30	D		
7	101	Error!		
8	-12	Error!		
9	36	D		
10	100	A		
11	0	F		
12				
13				

Figure 13.6: The grade function

Example 13.3

In this example, we create a function that calculates commissions payment based on the commissions payment table below. We can use the If....Then...Elseif program structure to write the function code.

Commissions Payment Table

Sales Volume(\$)	Commissions
<500	3%
<1000	6%
<2000	9%
<5000	12%
>5000	15

Function `Comm(Sales_V As Variant) as Variant`

If `Sales_V < 500` Then

`Comm = Sales_V * 0.03`

Elseif `Sales_V >= 500 and Sales_V < 1000` Then

`Comm = Sales_V * 0.06`

Elseif `Sales_V >= 1000 and Sales_V < 2000` Then

`Comm = Sales_V * 0.09`

Elseif `Sales_V >= 200 and Sales_V < 5000` Then

`Comm = Sales_V * 0.12`

Elseif `Sales_V >= 5000` Then

`Comm = Sales_V * 0.15`

End If

End Function

After creating the *Comm* Function, we can then enter the sales volume in one column and enter the formula based on the function *Comm* in another column. The commissions will be automatically computed and updated accordingly.

The screenshot shows a Microsoft Excel window titled "Sales Volume". The spreadsheet has columns A through F and rows 1 through 13. Column B is labeled "Sales Volume" and column C is labeled "Commissions". The data in column B is highlighted in yellow, and the data in column C is highlighted in cyan. The formula bar shows the formula `=Comm(B4)` for cell C4.

	A	B	C	D	E	F
1						
2						
3		Sales Volume	Commissions			
4		1000	90			
5		300	9			
6		10000	1500			
7		20000	3000			
8		500	30			
9		10	0.3			
10		1700	153			
11		2000	240			
12		4900	588			
13						

Figure 13.7

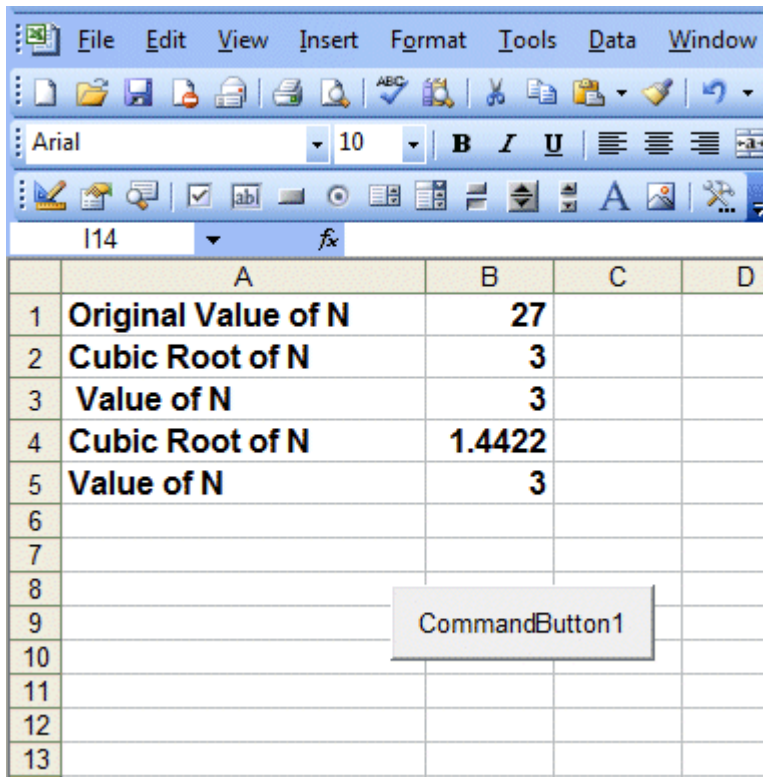
13.4 Passing variables by reference and by Value in a Function

Variables in a function can be passed by reference or by value, using the keywords *ByRef* and *ByVal* respectively. The main difference between *ByRef* and *ByVal* is *ByRef* will change the value of the variable while *ByVal* will retain the original value of the variable. By default, the function uses *ByRef* to pass variables.

Example 13.4

```
Private Sub CommandButton1_Click()  
    Dim N As Variant  
    N = 27  
    Range("A1") = CRoot(N)  
    Range("A2") = N  
    Range("A3") = CRoot2(N)  
    Range("A4") = N  
End Sub  
  
Function CRoot(ByRef r As Variant)  
    r = r ^ (1 / 3)  
    CRoot = r  
End Function  
  
Function CRoot2(ByVal w As Variant)  
    w = w ^ (1 / 3)  
    CRoot2 = w  
End Function
```

In this example, we created two similar functions, CRoot and CRoots respectively. However, the first function uses the ByRef keyword and the second function uses the ByVal keyword. Notice that the value of N has changed to 3 by the function CRoot, as shown in cell B3. Now the function CRoot2 compute the cubic root of N based on the new value of N, i.e. 3, and shows result in cell B4. However, it does not change the value of N, it remains as 3, as shown in cell B5.



The screenshot shows the Microsoft Excel interface with a spreadsheet. The menu bar includes File, Edit, View, Insert, Format, Tools, Data, and Window. The toolbar contains various icons for file operations and editing. The font is set to Arial, size 10, with bold, italic, and underline options. The active cell is I14, and the formula bar shows a function symbol (fx). The spreadsheet data is as follows:

	A	B	C	D
1	Original Value of N	27		
2	Cubic Root of N	3		
3	Value of N	3		
4	Cubic Root of N	1.4422		
5	Value of N	3		
6				
7				
8				
9				
10				
11				
12				
13				

A command button labeled "CommandButton1" is located in cell B9.

Chapter 14

VBA Procedures Part 2-Sub Procedures

A sub procedure is a procedure that performs a specific task and to return values, but it does not return a value associated with its name. However, it can return a value through a variable name. Sub procedures are usually used to accept input from the user, display information, print information, manipulate properties or perform some other tasks. It is a program code by itself and it is not an event procedure because it is not associated with a runtime procedure or a VBA control such as a command button. It is called by the main program whenever it is required to perform a certain task. Sub procedures help to make programs smaller and easier to manage.

A Sub procedure begins with a Sub statement and ends with an End Sub statement. The program structure of a sub procedure is as follows:

```
Sub ProcedureName (arguments)
    Statements
End Sub
```

Example 14.1

In this example, a sub procedure *ResizeFont* is created to resize the font in the range if it fulfills a value greater than 40. There are two parameters or arguments associated with the sub procedure, namely x for font size and Rge for range. This sub procedure is called by the event procedure Sub CommandButton1_Click () and passed the values 15 to x (for font size) and Range ("A1:A10") to Rge (for range) to perform the task of resizing the font to 15 for values>40 in range A1 to A10.

```
Private Sub CommandButton1_Click()
```

```

        ResizeFont 15, Range("A1:A10")
    End Sub

Sub ResizeFont(x As Variant, Rge As Range)
    Dim cel As Range
    For Each cel In Rge
        If cel.Value > 40 Then
            cel.Font.Size = x
        End If
    Next cel
End Sub

```

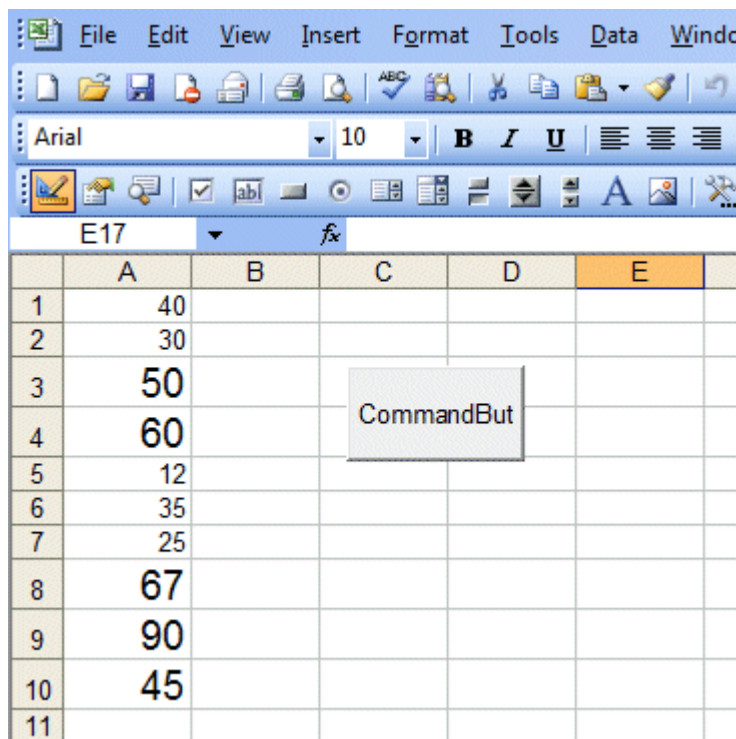


Figure 14.1: Output for Example 14.1

To make the program more flexible and interactive, we can modify the above program to accept input from the user. The values input by the user through the input

boxes will be passed on to the procedure to execute the job, as shown in Example 14.2.

Example 14.2

```
Private Sub CommandButton1_Click()  
    Dim rng As String  
    rng = InputBox("Input range")  
    x = InputBox("Input Font Size")  
    ResizeFont x, Range(rng)  
End Sub  
  
Sub ResizeFont(x As Variant, Rge As Range)  
    Dim cel As Range  
    For Each cel In Rge  
        If cel.Value > 40 Then  
            cel.Font.Size = x  
        End If  
    Next cel  
  
End Sub
```

Chapter 15

String Handling Functions

Excel VBA handles strings similar to the stand-alone Visual Basic program. All the string handling functions in Visual Basic such as Left, Right, Instr, Mid and Len can be used in Excel VBA. Some of the string handling functions are listed and explained below:

15.1 InStr

InStr is a function that looks for the position of a substring in a phrase.

InStr (phrase,"ual") will find the substring "ual" from "Visual Basic" and then return its position; in this case, it is fourth from the left.

15.2. Left

Left is a function that extracts characters from a phrase, starting from the left. Left (phrase, 4) means four characters are extracted from the phrase, starting from the leftmost position.

15.3. Right

Right is a function that extracts characters from a phrase, starting from the Right. Right (phrase, 5) means 5 characters are extracted from the phrase, starting from the rightmost position.

15.4. Mid

Mid is a function that extracts a substring from a phrase, starting from the position specified by the second parameter in the bracket. Mid (phrase, 8, 3) means a substring of three characters are extracted from the phrase, starting from the 8th position from the left.

15.5. Len

Len is a function that returns the length of a phrase.

Example 15.1

In this example, we insert five command buttons and change the names to cmdInstr, cmdLeft, cmdRight, cmdLeft, cmdMid and cmdLen respectively.

```
Private Sub cmdInstr_Click ()  
    Dim phrase As String  
    phrase = Cells (1, 1).Value  
    Cells (4, 1) = InStr (phrase, "ual")
```

```
End Sub
```

```
Private Sub cmdLeft_Click ()  
    Dim phrase As String  
    phrase = Cells (1, 1).Value  
    Cells (2, 1) = Left (phrase, 4)
```

```
End Sub
```

```
Private Sub cmdLen_Click ()  
    Dim phrase As String  
    phrase = Cells (1, 1).Value  
    Cells (6, 1) = Len (phrase)
```

```
End Sub
```

```
Private Sub cmdMid_Click ()
```



```

Dim phrase As String
phrase = Cells (1, 1).Value
Cells (5, 1) = Mid (phrase, 8, 3)
End Sub

Private Sub cmdRight_Click ()
    Dim phrase As String
    phrase = Cells (1, 1).Value
    Cells (3, 1) = Right (phrase, 5)
End Sub

```

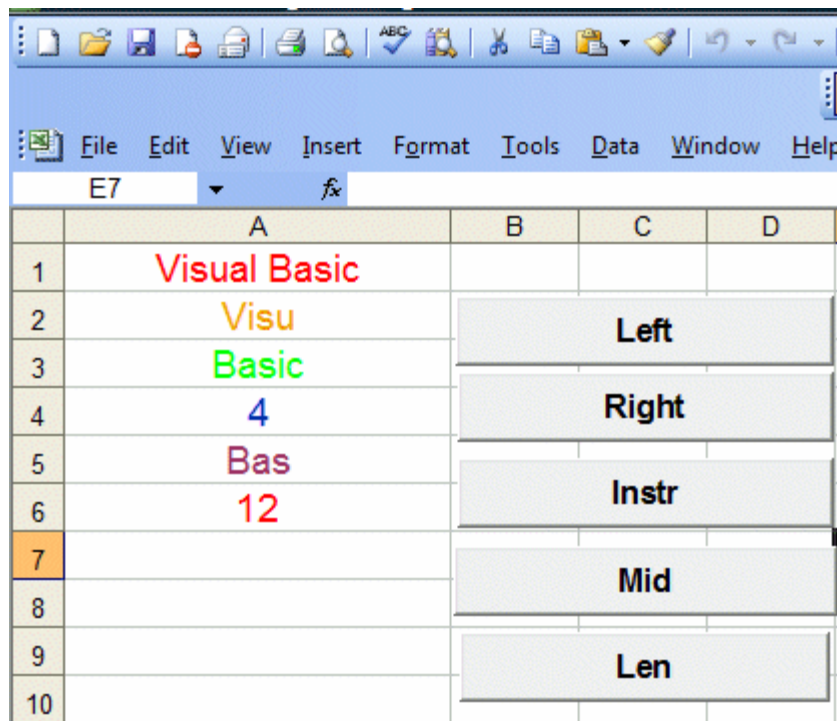


Figure 15.1: The end results after clicking all the command buttons.

Chapter 16

Date and Time Functions

Excel VBA can be programmed to handle Date and Time, adding extra capabilities to time and date handling by MS Excel. We can use various built-in date and time handling functions to program Excel VBA date and time manipulating programs.

16.1 Using the Now () Function

The Now () function returns the current date and time according to your computer's regional settings. We can also use the Format function in addition to the function Now to customize the display of date and time using the syntax `Format (Now, "style argument")`. The usage of Now and Format functions are explained in the table below:

Table 16.1: Various Date and Time Formatting with Different Style Arguments

Formatting with various style arguments	Output
<code>Format(Now, "s")</code>	Current Time in seconds
<code>Format(Now, "n")</code>	Current Time in minutes
<code>Format(Now, "h")</code>	Current Time in hours
<code>Format(Now, "m")</code>	Current Month in numeric form
<code>Format(Now, "mmm")</code>	Current Month in short form
<code>Format(Now, "mmmm")</code>	Current Month in full name
<code>Format(Now, "y")</code>	Number of days to date in current year
<code>Format(Now, "yyyy")</code>	Current Year

Example 16.1

```
Private Sub CommandButton1_Click ()
```

Cells (1, 1).Value = Now ()

Cells (2, 1).Value = Format (Now, "s")

Cells (3, 1).Value = Format (Now, "n")

Cells (4, 1).Value = Format (Now, "h")

Cells (5, 1).Value = Format (Now, "m")

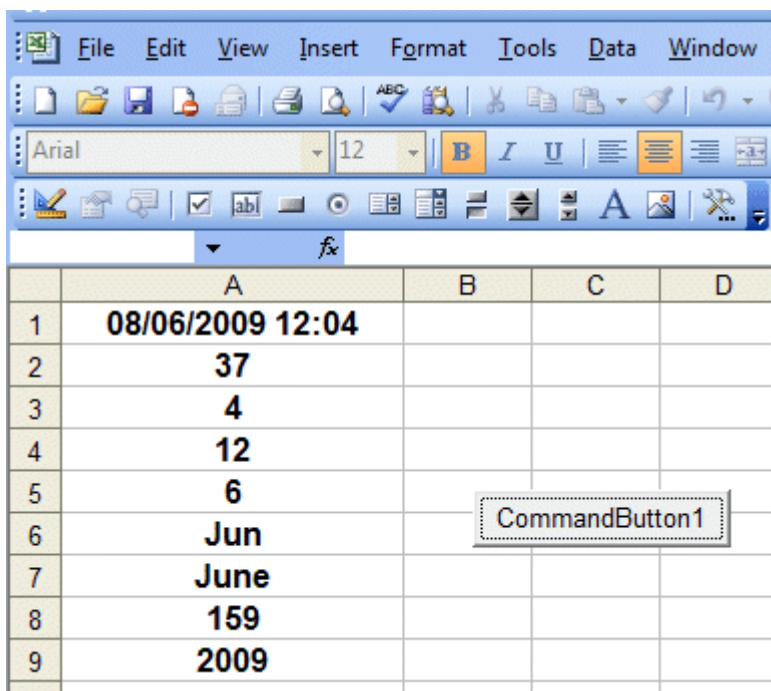
Cells (6, 1).Value = Format (Now, "mmm")

Cells (7, 1).Value = Format (Now, "mmmm")

Cells (8, 1).Value = Format (Now, "y")

Cells (9, 1).Value = Format (Now, "yyyy")

End Sub



	A	B	C	D
1	08/06/2009 12:04			
2	37			
3	4			
4	12			
5	6			
6	Jun	CommandButton1		
7	June			
8	159			
9	2009			

Figure 16.1: Output of various date and time formats

16.2 Date, Day, Weekday, WeekdayName, Month, MonthName and Year Functions

The usage of these functions is illustrated in the following table:

Table 16.2: Various Date and Time functions

Function	Output
Date	Current date and time
Day(Date)	Day part of the current date
Weekday(Date)	Weekday of the current week in numeric form.
WeekdayName(Weekday(Date))	Weekday name of the current date
Month(Date)	Month of the current year in numeric form
MonthName(Month(Date))	Full name of the current month
Year(Date)	Current year in long form

Example 16.2

```

Private Sub CommandButton1_Click()
    Cells(1, 1) = Date
    Cells(2, 1) = Day(Date)
    Cells(3, 1) = Weekday(Date)
    Cells(4, 1) = WeekdayName(Weekday(Date))
    Cells(5, 1) = Month(Date)
    Cells(6, 1) = MonthName(Month(Date))
    Cells(7, 1) = Year(Date)
End Sub

```

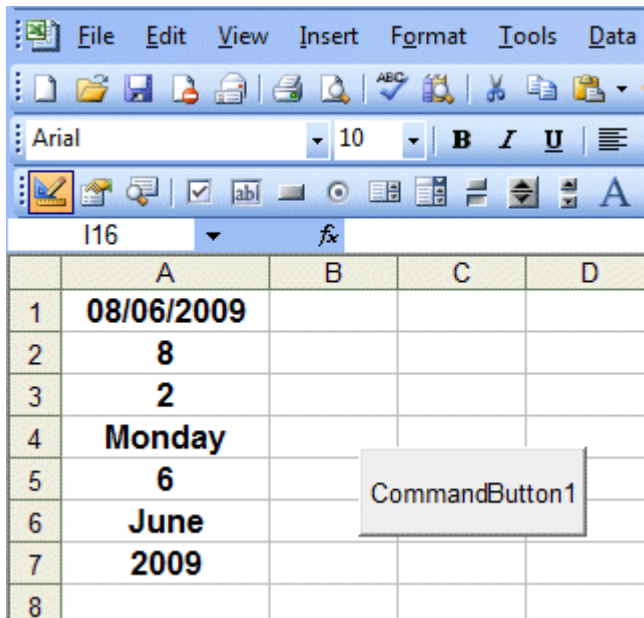


Figure 16.2: Output of various date and time formats.

16.3 DatePart Function

The *DatePart* function is used together with the Now function to obtain part of date or time specified by the arguments. The *DatePart* function is generally written as

DatePart (Part of date to be returned, Now)

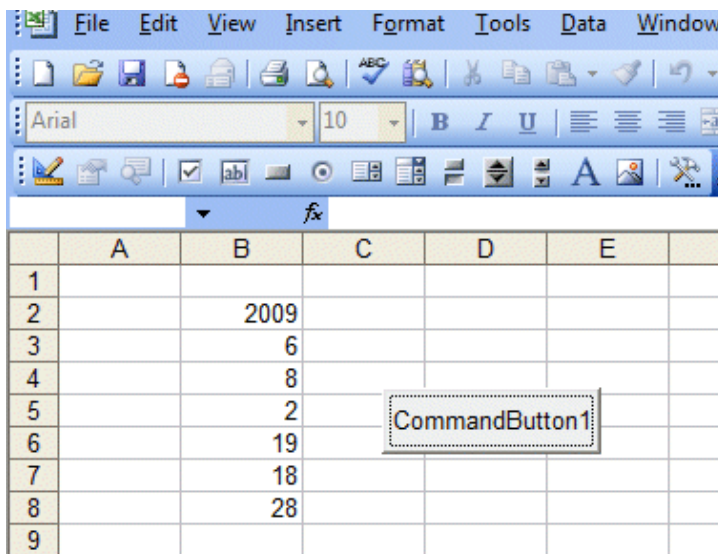
Various DatePart expressions and the corresponding outputs are shown in Table 16.3

Table 16.3: DatePart Expressions

DatePart Expression	Part of Date /Time Returned
DatePart("s",Now)	Current second
DatePart("n",Now)	Current minute
DatePart("h",Now)	Current hour
DatePart("w",Now)	Current weekday
DatePart("m",Now)	Current month
DatePart("y",Now)	Current day of the year
DatePart("yyyy",Now)	Current year

Example 16.3

```
Private Sub CommandButton1_Click ()  
    Cells (2, 2) = DatePart ("yyyy", Now)  
    Cells (3, 2) = DatePart ("m", Now)  
    Cells (4, 2) = DatePart ("d", Now)  
    Cells (5, 2) = DatePart ("w", Now)  
    Cells (6, 2) = DatePart ("h", Now)  
    Cells (7, 2) = DatePart ("n", Now)  
    Cells (8, 2) = DatePart ("s", Now)  
  
End Sub
```

**Figure 16.3: DatePart Function**

16.4 Adding and Subtracting Dates

Dates can be added using the *DateAdd* function. The syntax of the *DateAdd* function is

```
DateAdd (interval, value to be added, date)
```

Where interval=part of date to be added. For example, *DateAdd* ("yyyy", 3, Now) means 3 years will be added to the current year. Similarly, Dates can be subtracted using the *DateDiff* function. The syntax of the *DateDiff* function is

```
DateDiff (interval, first date, second date)
```

Where interval=part of date to be subtracted. For example, *DateDiff* ("yyyy", Now, "6/6/2012") means 3 years will be subtracted from the current year. Both the aforementioned functions use the argument "s" for second, "n" for minute, "h" for hour, "d" for day, "w" for week, "m" for month and "yyyy" for year.

Example 16.4

```
Private Sub CommandButton1_Click ()
```

```
Cells (1, 1) = Date
Cells (2, 1) = DateAdd ("s", 300, Now)
Cells (3, 1) = DateAdd ("n", 30, Now)
Cells (4, 1) = DateAdd ("h", 3, Now)
Cells (5, 1) = DateAdd ("d", 2, Now)
Cells (6, 1) = DateAdd ("m", 3, Now)
Cells (7, 1) = DateAdd ("yyyy", 2, Now)
Cells (8, 1) = DateDiff ("yyyy", Now, "8/6/2012")
Cells (9, 1) = DateDiff ("d", Now, "13/6/2009")
Cells (10, 1) = DateDiff ("m", Now, "8/10/2011")
Cells (11, 1) = DateDiff ("d", Now, "8/10/2009")
Cells (12, 1) = DateDiff ("n", Now, "8/10/2009")
```

```
Cells(13, 1) = DateDiff("s", Now, "8/10/2009")
```

```
End Sub
```

	A	B	C	D	E
1	08/06/2009				
2	08/06/2009 20:15				
3	08/06/2009 20:40				
4	08/06/2009 23:10				
5	10/06/2009 20:10		CommandButton1		
6	08/09/2009 20:10				
7	08/06/2011 20:10				
8	3				
9	5				
10	28				
11	122				
12	174470				
13	10468164				
14					
15					

Figure 16.5: DateAdd and DatePart functions

Chapter 17

Sample Excel VBA Programs

17.1 BMI Calculator

Body Mass Index (BMI) is so popular today that it has become a standard measure for our health status. If your BMI is too high, it means you are overweight and would likely face a host of potential health problems associated with high BMI, such as hypertension, heart diseases, diabetics and many others. The formula for calculating BMI is

$$\text{BMI} = \text{weight} / (\text{height})^2$$

The Excel VBA code for BMI calculator is illustrated below:

```
Private Sub CommandButton1_Click()  
    Dim weight, height, bmi, x As Single  
    weight = Cells(2, 2)  
    height = Cells(3, 2)  
    bmi = (weight) / height ^ 2  
    Cells(4, 2) = Round(bmi, 1)  
    If bmi <= 15 Then  
        Cells(5, 2) = "Under weight"  
    ElseIf bmi > 15 And bmi <= 25 Then  
        Cells(5, 2) = "Optimum weight"  
    Else  
        Cells(5, 2) = "Over weight"  
    End If  
  
End Sub
```

The function Round is to round the value to a certain decimal places. It takes the format Round(x, n), where x is the number to be rounded and n is the number of decimal places. The second part of the program uses the If...Then.... Else statement to evaluate the weight level. The output is shown in Figure 17.1

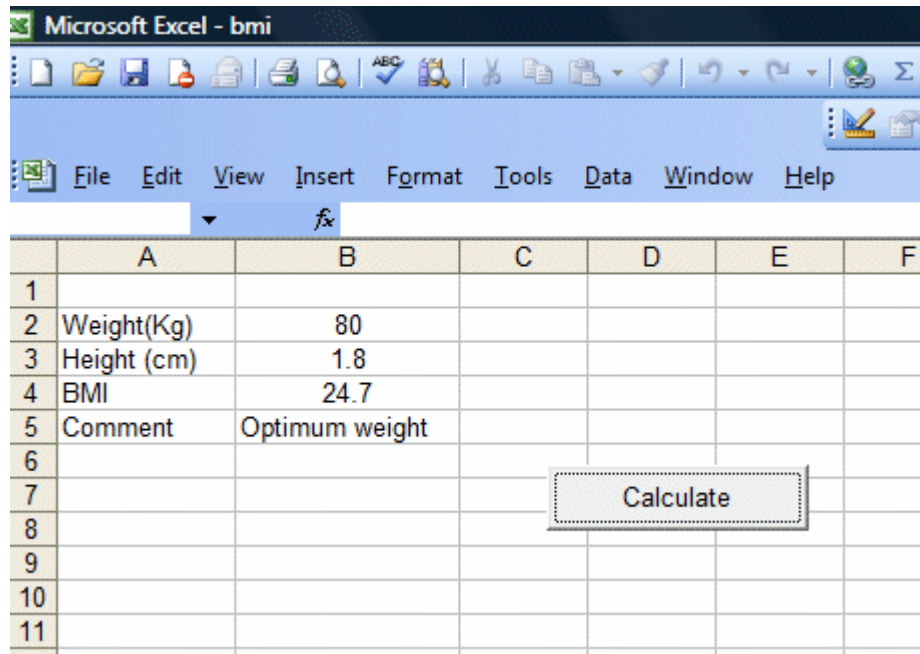


Figure 17.1: BMI Calculator

17.2: Financial Calculator

This is an Excel VBA program that can calculate monthly payment for the loan taken from the bank. The formula to calculate periodic payment is shown below, where PVIFA is known as present value interest factor for an annuity.

$$\text{Payment} = \text{Initial Principal} / \text{PVIFA},$$

The formula to compute PVIFA is

$$1/i - 1/i (1+i)^{-n}$$

where n is the number of payments. Normally you can check up a financial table for the value of PVIFA and then calculate the payments manually. The function Format is to determine the number of decimal places and the use of the \$ sign. Below is the Excel VBA code for the financial calculator:

```
Private Sub CommandButton1_Click()

    Dim N As Integer
    Dim p, pmt, rate, I, PVIFA As Double
    p = Cells(2, 2)
    rate = Cells(3, 2)
    N = Cells(4, 2) * 12
    I = (rate / 100) / 12
    PVIFA = 1 / I - 1 / (I * (1 + I) ^ N)
    pmt = p / PVIFA
    Cells(5, 2) = Format(pmt, "$#,###0.00")

End Sub
```

The above financial VBA calculator can also be programmed using the built-in worksheet function, PMT. It is very much easier to program than the previous one.

The format of this function is

`WorksheetFunction.pmt (rate, N, amount)`

Where rate is the interest rate, N is the period of payments (of number of periodic payments) and amount is the amount borrowed.

People usually key in the annual interest rate as an integer rather than in decimal form, so we need to divide the rate by 100 and then divide again by 12 to get the monthly rate.

The negative sign is placed in front of the amount borrowed because this is the amount the borrower owed the financial institute,. If we don't put the negative sign, the payment will have a negative sign.

The VBA code is shown below:

```
Private Sub CommandButton1_Click ()
```

```
    Dim rate, N As Integer
```

```
    Dim amt, payment As Double
```

```
    amt = Cells(2, 2)
```

```
    rate = (Cells(3, 2) / 100) / 12
```

```
    N = Cells(4, 2) * 12
```

```
    payment = WorksheetFunction.pmt(rate, N, -amt)
```

```
    Cells(5, 2) = Format(payment, "$###,###.00")
```

```
End Sub
```

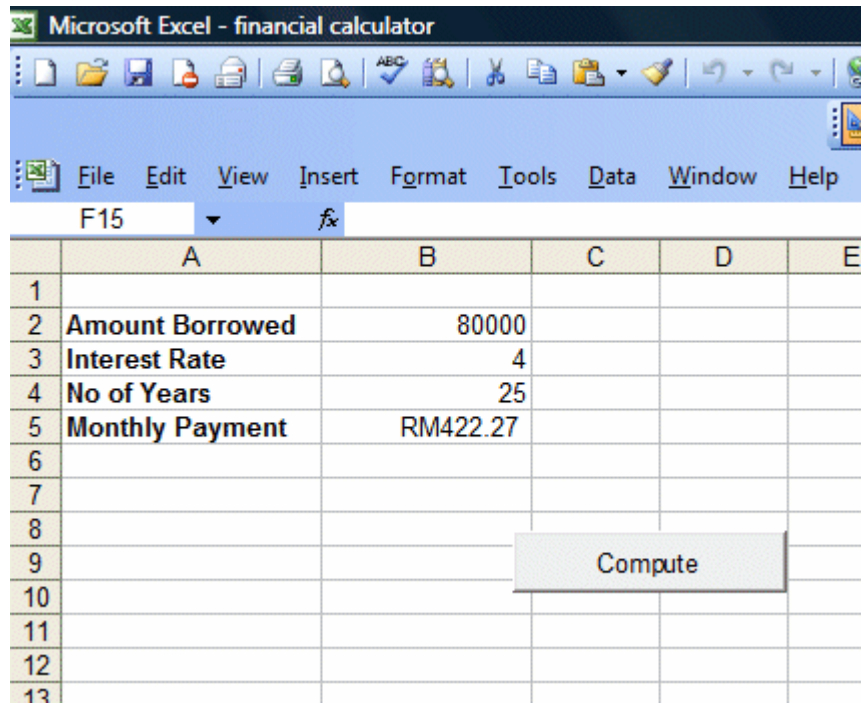


Figure 17.2: Financial Calculator

17.3: Investment Calculator

In order to get one million dollars in the future, we need to calculate the initial investment based on the interest rate and the length of a period, usually in years.

The formula is

`WorksheetFunction.PV (rate, N, periodic payment, amount, due)`

Where rate is the interest rate, N is the length of the period and amount is the amount borrowed. Below is the Excel VBA code for the investment Calculator:

```
Private Sub CommandButton1_Click ()
```

```
    Dim F_Money, Int_Rate, Investment As Double
```

```
    Dim numYear As Single
```

```
    F_Money = Cells(2, 2)
```

```
    Int_Rate = (Cells(3, 2) / 100)
```

```
    numYear = Cells(4, 2)
```

```
    Investment = PV(Int_Rate, numYear, 0, F_Money, 1)
```

```
    Cells(5, 2) = Format(-Investment, "$###,###,##0.00")
```

```
End Sub
```

	A	B	C
1			
2	Future Value	1000000	
3	Interest Rate	5	
4	Number of Years	30	
5	Initial Investment	RM231,377.45	
6			
7			
8			
9			
10		Calculate	
11			
12			

Figure 17.3: Investment Calculator

17.4: Prime Number Tester

This Excel VBA program will test whether a number entered by the user is a prime number or not. Prime number is a number that cannot be divided by other numbers other than itself, it includes 2 but exclude 1 and 0 and all the negative numbers.

In this program, we use the *Select CaseEnd Select* statement to determine whether a number entered by a user is a prime number or not. For case 1, all numbers that are less than 2 are not prime numbers. In Case 2, if the number is 2, it is a prime number. In the last case, if the number N is more than 2, we divide this number by all the numbers from 3,4,5,6,.....up to N-1, if it can be divided by any of these numbers, it is not a prime number, otherwise it is a prime number. We use the *Do.....Loop While* statement to control the program flow. Besides, we also used a tag="Not Prime' to identify the number that is not prime, so that when the routine exits the loop, the label will display the correct answer. Below is the code:

```
Private Sub CommandButton1_Click ()

    Dim N, D As Single
    Dim tag As String
    N = Cells (2, 2)
    Select Case N
    Case Is < 2
        MsgBox "It is not a prime number"
    Case Is = 2
        MsgBox "It is a prime number"
    Case Is > 2
        D = 2
        Do
            If N / D = Int(N / D) Then
```

```
MsgBox "It is not a prime number"  
tag = "Not Prime"  
Exit Do  
End If  
D = D + 1  
Loop While D <= N - 1  
  
    If tag <> "Not Prime" Then  
        MsgBox "It is a prime number"  
    End If  
End Select  
  
End Sub
```

17.5 Selective Summation

This is an Excel VBA program that can perform selective summation according to a set of conditions. For example, you might just want to sum up those figures that have achieved sales target and vice versa. This VBA program can sum up marks that are below 50 as well as those marks which are above 50.

In this program, rng is declared as range and we can set it to include certain range of cells, here the range is from A1 to A10.

Then we used the ForNext loop to scan through the selected range

rng.Cells(i).Value read the value in cells(i) and then passed it to the variable mark.

To do selective addition, we used the statement Select Case....End Select

Finally, the results are shown in a message box

Here is the code:

```
Private Sub CommandButton1_Click ()

    Dim rng As Range, i As Integer
    Dim mark, sumFail, sumPass As Single
    sumFail = 0
    sumPass = 0
    Set rng = Range("A1:A10")
    For i = 1 To 10
        mark = rng.Cells(i).Value
        Select Case mark
            Case Is < 50
                sumFail = sumFail + mark
            Case Is >= 50
                sumPass = sumPass + mark
        End Select
    Next i

    MsgBox "The sum of Failed marks is" & Str(sumFail) & vbCrLf &
        "The sum of Passed marks is" & Str(sumPass)

End Sub
```