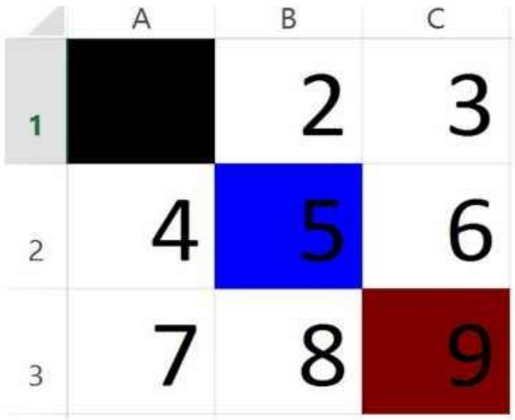
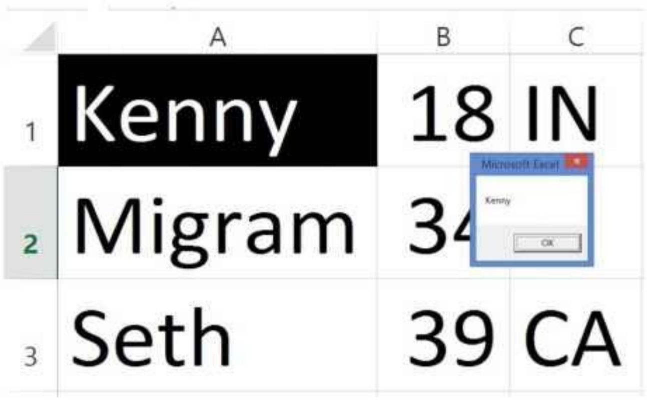
Learning to Love Microsoft Excel Visual Basic for Applications (VBA) Loop Statements Today



	A	B	C
1		2	3
2	4	5	6
3	7	8	9



	A	B	C
1	Kenny	18	IN
2	Migram	34	
3	Seth	39	CA

```
Sub Ken()
For Each kSht In Application.Sheets
Application.Speech.Speak (kSht.Name)
MsgBox kSht.Name
Next kSht
End Sub
```

	A
1	Kenny

	A	B	C
1		2	3
2	4	5	6
3	7	8	9

Kenny L Keys

Table of Contents

Title

Learning to Love Microsoft Excel Visual Basic for Applications (VBA) Loop Statements Today

Copyright

This book is dedicated to Ann Brown Hamilton and the rest of my extended paternal and maternal family, and I'd never forget to dedicate this book to all of my nieces, nephews, great nieces and great nephews. If we have a desire and a determination, we can accomplish many goals that we set for ourselves. We make our dreams happen, with or without the blessing of others.

Preface

This is a continuation of my first tutorial eBook: ***Microsoft Excel VBA Codes Are Fun, Simple, and Easy to Learn in One Hour or Less: VBA for Students, Parents, and Professionals (First Edition)***. This edition walks users through more than twenty (20) Visual Basic codes. This book also includes two additional eBooks that I've published in the past. I've discontinued those eBooks, and I've inserted them at the end of this new book on VBA; thus, you get three books in one.

This book breaks each line of code down, line-by-line. I repeat this process over and over in chapter two; this is done to ensure that users walk away, not only understanding VBA and loop statements, but also able to edit or revise any VBA code that he or she comes across in Microsoft Excel spreadsheets. I've also inserted print screens, so users will see the results of many of the codes in this book.

Exercises: Chapter 3 begins the section that allows users to get more familiar with VBA by changing portions of the original code, i.e., renaming variables, text strings and integer containers.

Combining Microsoft's Like, And, Less Than & Greater Than Operators with Numeric, Single Character & Any Number of Characters Wildcards in VBA Statements: Chapter 4 begins the second book that I retired, so I could consolidate it into this one. This portion of the book will only take users five minutes to learn to use Microsoft's VBA operators (e.g., <, >, Like, And, Or) and its wildcard characters (e.g., #, ? and *) to filter for data in spreadsheet cells. If a user has ever worked with filtering data in Microsoft Access databases, he or she will be very familiar with these operators and wildcard characters and how they're used to extract data, but the user must use these operators and wildcard characters in VBA codes differently than he or she would use them in the Microsoft Access Query View window. Although Microsoft has done a very good job creating unique applications to meet the needs of its users, it knows that some users want more control over their applications; these individuals choose to use VBA to manage them, e.g., Word, Excel, PowerPoint and Access; however, this portion of the book will only cover Microsoft Excel VBA built-in operators.

Using the *Like* operator along with wildcard characters will allow a user to quickly filter for data that she or he needs and wants, within a matter of seconds, without needing to type Microsoft formulas and functions in each corresponding cell for the data that he or she is trying to extract or filter, e.g., *"Sharon Mays lives at 2589 Mathew Lane"* Like *"Sharon Mays*Mathew L???."*

Using the *And* operator allows a user to instruct Microsoft Excel to ensure that two or more criteria are met before a condition or statement can lead to *true*, e.g., *3 > 0 And 3 < 5*. Three is indeed greater than zero, and it's less than five, so both conditions of the *And* operator statement has been met; thus, the result of this statement is *true*; however, if one of the two statements were false, the result of the entire statement would be false. The *Or* operator allows for only one of two or more statements to be true, in order for the full statement to result in true, but I won't be covering the *Or* operator in this book.

Users will learn how to use the greater than (>) and less than (<) operators along with a couple wildcard characters (e.g., *, ?) in this book, and I'll explain one entire VBA code line by line, in this section of the book, so users will gain a clearer understanding of how each line of code in VBA has a specific task or function. When a user completes this section of the book, he or she should feel comfortable writing his or her own VBA code in Microsoft Excel to filter for spreadsheet data.

Microsoft Excel Formulas for Data Extractions: Chapter 5 begins the third book that I retired, so I could consolidate it into this one. This portion of the book will only take users five minutes to learn to use Microsoft Excel data formulas: **left, right, mid, trim, len, find, index, if, and, or, iserror, hlookup, vlookup, concatenate, offset, match, small,** and **large**. In order to create powerful mega-formulas which are used to extract and manipulate data without using VBA or macros; users will learn how to nestle formulas. This book makes it much easier for users to learn how to use Microsoft formulas for tasks that may seem complex; in some cases, these tasks may even seem impossible, but users will become masters of Microsoft Excel spreadsheets and data manipulation in less than five minutes.

The layout of this section of the book consists of formulas accompanied by visual and written results; complete formulas are displayed for each example; they execute into real results; this section will also familiarize users with variables, arrays, and defining content in spreadsheets.

This small book doesn't contain an index, a reference section or a study guide.

Meeting the Master Builder and Learning Some Basics of VBA: Chapter 1

Meeting the Master Builder in a Fairytale about Visual Basic for Applications (VBA): Chapter 1A

I'm going to teach you VBA in a way that will make it easy for you to remember. I'm going to associate VBA, functions and other actions to a container or jar, but before I do this, let's get you in the mood for learning something a bit complex by telling you a story.

Once upon a time, there was a small, yet powerful creature that lived inside of every computer's operating system, browser, console, application and app. This creature was more powerful than anything ever known to man, but its powers were out of reach to humans that were ignorant about its language and purpose. You see, this creature (although powerful) had to serve any human that learned its language and its purpose, so this creature did everything that it could to dissuade humans from believing in their ability to learn program languages like Microsoft Visual Basic for Applications (VBA), JavaScript or C++. This creature would taunt humans by calling them fleshlings and moisture packs. It did everything that it could to belittle humans, but one day, a little boy (about seven years old) just happened to stumble upon an old man; this old man was sitting next to a narrow, dirty creek with his back up against a large, old, majestic tree that was rooted into the ground, about three feet from this narrow creek. The old man had a laptop with him, and he appeared to be typing something inside of it. The little boy eased up behind the old man, and he saw the man typing gibberish; almost immediately, the laptop's screen changed colors and produced a 3D object that was spinning in the center of the laptop's screen display. The little boy wasn't noticed by the old man, because he (the old man) was known to be deaf in both ears. The man reproduced the screen that had the gibberish, and he typed in more gibberish; a few second later, the same screen appeared with the 3D object spinning, but this time, there was also a dancing stick man to the left of the object. The man seemed pleased with himself, and a few second, after he closed his laptop, the little creature appeared and bowed its head to the old man, and it looked the old man straight in the eyes and slowly moved its lips, as if to allow the old man to read what it was saying to him. The creature asked this question: Master, have I served you well? The old man replied: you have done as you always have, being the slave of the operating system, browser and console, you have served me well, as always. The little boy had heard stories (or fairytales) about a human that was wiser than the little creature that loved to taunt humans. It is said that the little creature's true home is not of this world, but inside of an electronic device that allows humans to type and create their life stories, wishes and desires; however, this creature leaves its home, often, to venture into the human world to taunt those who hasn't figured out its secrets. You see, it is said that this creature is like a genie; it must grant your every wish, if you learn its language and its purpose; however, unlike a genie who only grants you three (3) wishes, this creature becomes a slave to any human that learns to speak, write and understand its language.

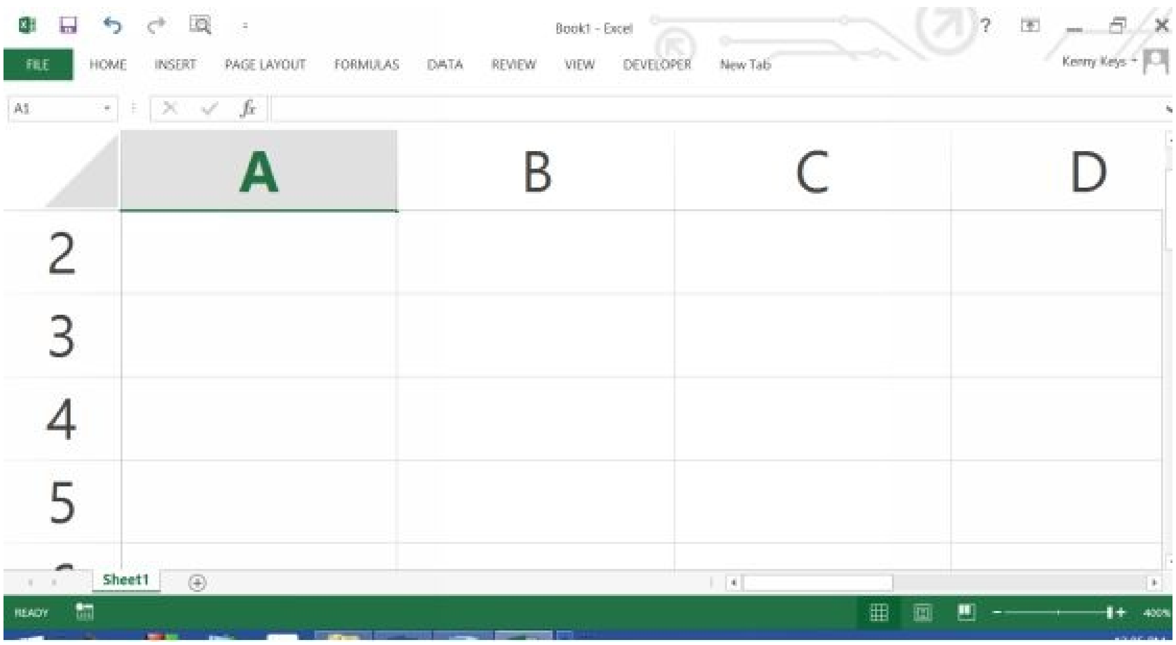
The next day, the little boy arrived at the creek to spy on the old man again, but this time, when he eased up behind the man, he was greeted with words that he didn’t expect: I’ve been waiting for you. The little boy looked around, but he saw no one. The old man had not even turned to look directly at the little boy, when he spoke again: I’m talking to you, little one. I know that you were here yesterday, and I also know that you’re curious about my relationship with the little creature that you saw me talking to, so I’ll tell you everything about my relationship with the “Master Builder” today. Before the old man spoke again, he turned his head to look the little boy in the eyes, and he continued: I’m old, and I need to pass my knowledge on to someone who will carry on the relationship with the Master Builder. The little boy asked: who is the Master Builder? Why it’s the creature that you saw, yesterday, replied the old man. He lives in all computers’ operating systems; he lives in all browsers, and he lives in all consoles, applications and apps. The old man instructed the boy: come sit down beside me, so we can get started, because I don’t have much time left. The little boy did as he was told. The old man opened up his laptop, and he called on the creature to appear: Master Builder, I command you to show yourself. Not more than a second later, the creature appeared. The old man informed the creature that it was time that he had a new master: you have served me for more than 65 years, but unlike you, I can’t and won’t live forever. You will live as long as a human needs your services; therefore, I’ve chosen someone that will keep you alive. We both know, despite your taunts towards humans, that you need us to keep you alive. You’ll never die, but you’ll no longer exist with your powers, if a human doesn’t request your services. As much as the creature hated serving humans, it knew what the old man had stated was true, and the creature didn’t want to cease to exist without its amazing powers, just because it despised (in its opinion) inferior humans having total dominion over it.

We’ll need to start from the beginning stated the old man. Son, as I type, the Master Builder will demonstrate his powers for you. He is more than just a magical creature, he is also an emperor of all things that lives inside of computers, but we are going to start out with something simple, a Microsoft Excel spreadsheet. The Master Builder speaks many languages, within the computer world, and you’ll need to be fluent in whichever one you’ll be using to request the services of the Master Builder. Since we are starting with Microsoft Excel spreadsheets, you’ll be learning the language called VBA. VBA stands for Visual Basic for Applications. The application is Microsoft Excel; VBA is a Microsoft language. From this point forward, the Master Builder will be known, simply as the application. Whenever the application is mentioned, it will be the Master Builder working his magic to grant us what we’ve instructed by way of our lines of code. Now, let’s get started. Shall we? The little boy nodded agreeably.

Getting Acquainted with VBA Integers, Variants, Text Strings, Ranges, Cells, Message Boxes and Input Boxes: Chapter 1B

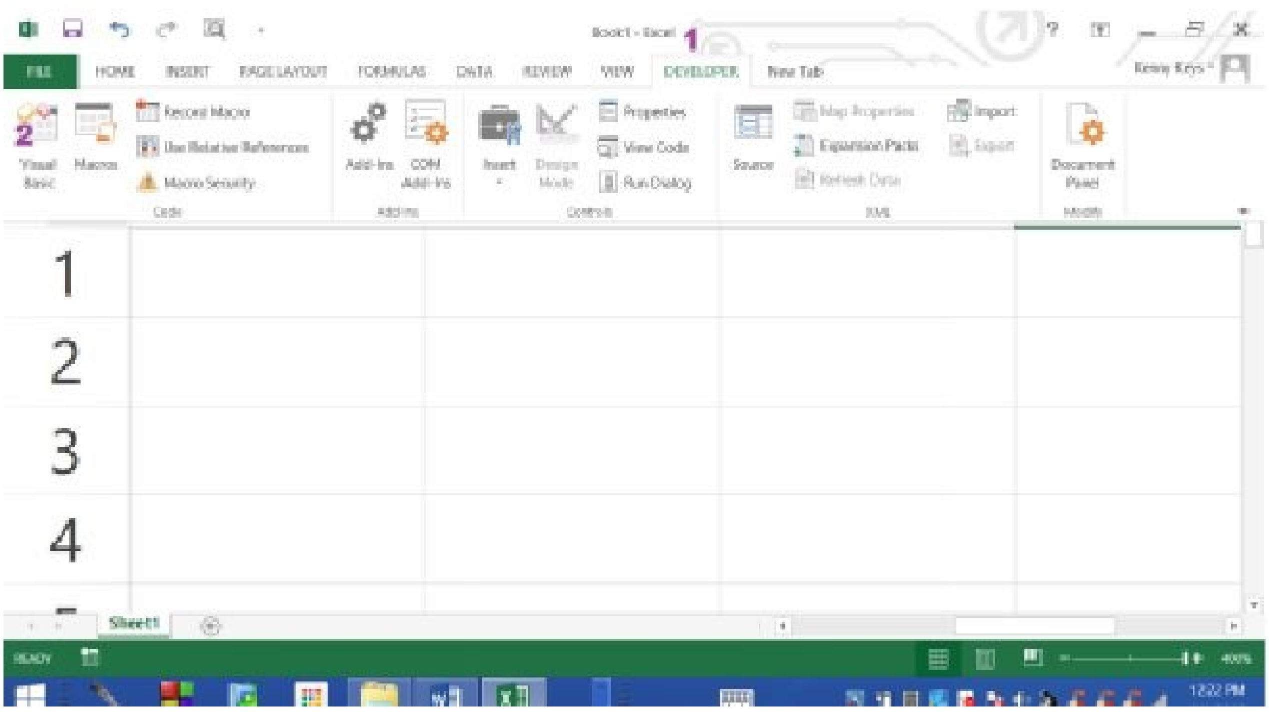
Our first lesson begins now, said the old man. When you first open up an Excel workbook, you'll automatically see sheet 1, as shown in Sample Object 1A.

Note: before going any further, I'd like for you to ***download*** this text file that contains some of the codes in **Chapter 2** and **Chapter 3**. Some of these codes will be long, and they'll wrap in this eBook, but they ***MUST NOT*** wrap in the **VBA Module window**; thus, I've remedied this issue by pasting all of the long codes into a **text file**; this text file will allow users to just ***copy and paste the full code directly into the VBA Module window and run it to see the correct results***. Users ***must*** still follow my instructions, before running each of the codes, so they work as they were intended. Here is the **link**.



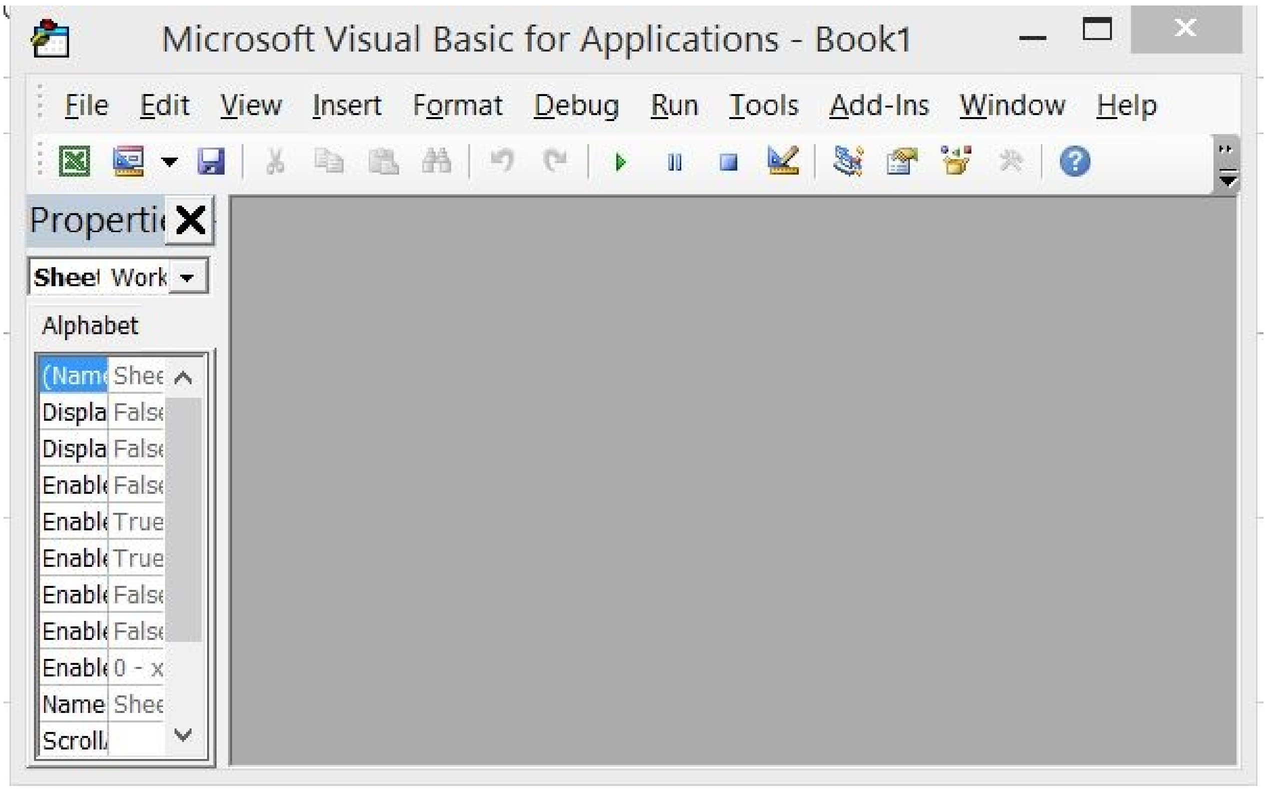
Sample Object 1A: this shows a snippet of a newly opened workbook that displays Sheet1.

Next, we need to access the screen that will allow us to write our own unique VBA code, so we'll need to click on the 9th top tab from the left; it is called the **Developer** tab, as shown in Sample Object 1B.



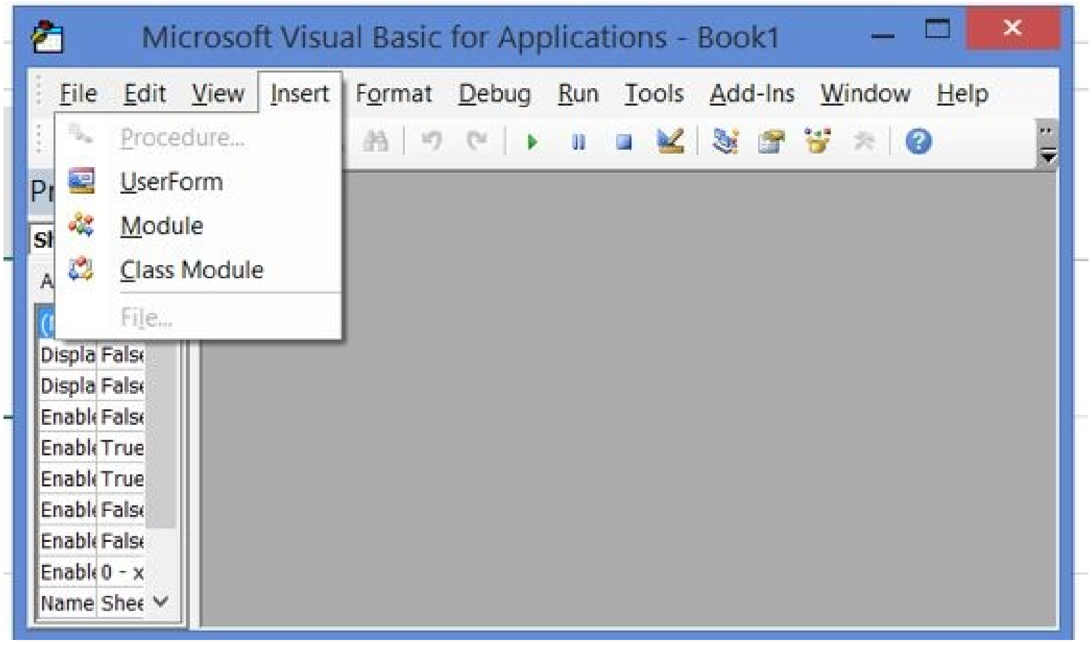
Sample Object 1B: this shows users how to access the **Developer** tab.

I've typed a "1" and "2" above both the **Developer** tab and the Visual Basic portal or gateway. You must access the portal or gateway, in order to speak to the Master Builder. It and I have known each other for so long, that it (sometimes) takes vacations in my brain; this is why I can call on it anytime that I need its services, but you won't be able to call on the Master Builder (aka the Application), unless you open the VBA portal first. Now, after you click on the **Developer** tab, you'll move down to the next level and all the way over to the left-most side of the menu bar; you should see a **Visual Basic icon**; you'll need to click on it (Visual Basic icon), and you'll see the **Microsoft Visual Basic for Applications (VBA) – Book 1** window open for you, as shown in Sample Object 1C.



Sample Object 1C: this shows a newly opened Visual Basic for Applications (VBA) window.

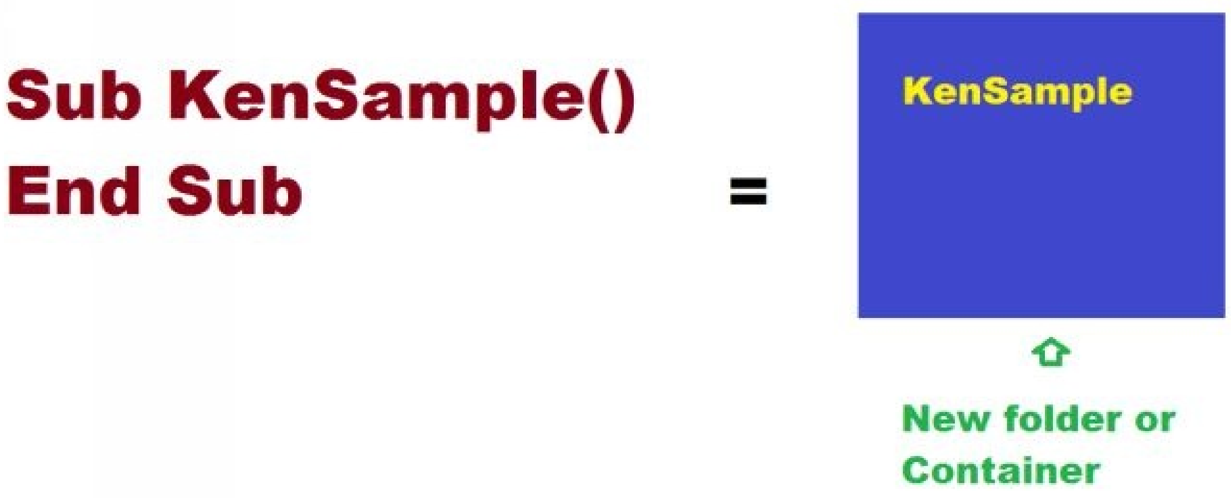
Once you’re in the VBA window, you’ll click on the **Insert** drop-down menu option and move down and select **Module**, as shown in Sample Object 1D.


Microsoft Visual Basic for Applications - Book1
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
Procedure...
UserForm
Module
Class Module
File...

Sample Object 1D: this shows the Insert menu option that contains the Module option.

Sample Object 1E displays a newly opened **Module** window; this is where you’ll type your command for the Master Builder (aka the application); it’ll execute your wishes, line by line, starting from the top line of code and working downwards, line by line. In

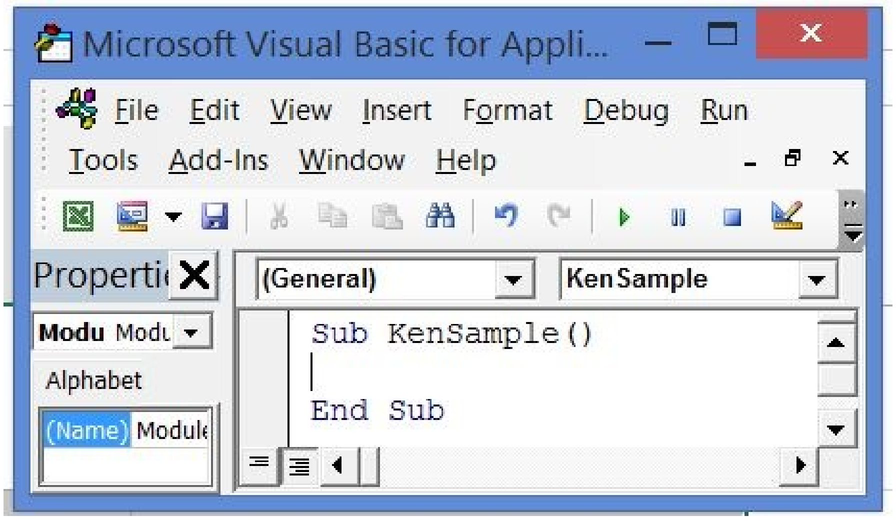
Sample Object 1E, you see how to begin your instructions for the Master Builder (aka application). You'll always start your instructions to it (the application) by typing the word **Sub** and then a single space followed by any name for this container that will hold all of your demands, commands or instructions for the application to execute for you; the **Sub** container name could be **oogashooka()**, **Mokilidee()**, **PegLeg()**, **CaptainTroubleCrook()**, **LKJLJLJDFDJF()**, **k3_34er()** or whatever name that you desire. The name will be for this main container or folder that will hold each line of your instructions to be executed by the application. To make it easy to understand, I've made a visualization of what ***Sample Object 1E*** means; it can be seen in the *SubFolderNew* image.



SubFolderNew: this image shows how a user creates a brand new, empty container or folder that will hold all requests to be executed by the application.

Sub KenSample()

End Sub


Microsoft Visual Basic for Appli...
File Edit View Insert Format Debug Run
Tools Add-Ins Window Help
Properti
Modu Modu
Alphabet
(Name) Module
(General)
KenSample

```
Sub KenSample()

End Sub
```

Sample Object 1E: this shows a newly added Module that allows user to start typing VBA code.

After you create your main folder or container, and you name it, as you've already done in ***Sample Object 1E***, you'll be ready to create additional, smaller folders or sub-containers to go inside your main folder or container; a sub-container name isn't followed by the two parentheses, e.g., (); instead it will be followed by the word As, e.g., As Integer, As Variant, As String, As Long, etc…; once it is created, it will always be followed by an equal sign, e.g., (=), when it is being filled with something, e.g., KenContainer = 1, KenContainer = "I love you", KenContainer = InputBox("Type something",""), etc…; this is how you will always know that a new sub-container is being created and filled with something. It's just like any filling system where you have a main folder and sub folders inside the main folder. The main folder is the name that follows the opening **Sub** tag, e.g., **KenSample()**, and the sub-containers are the containers that have equal signs after them, e.g., **KenContainer = 1**.

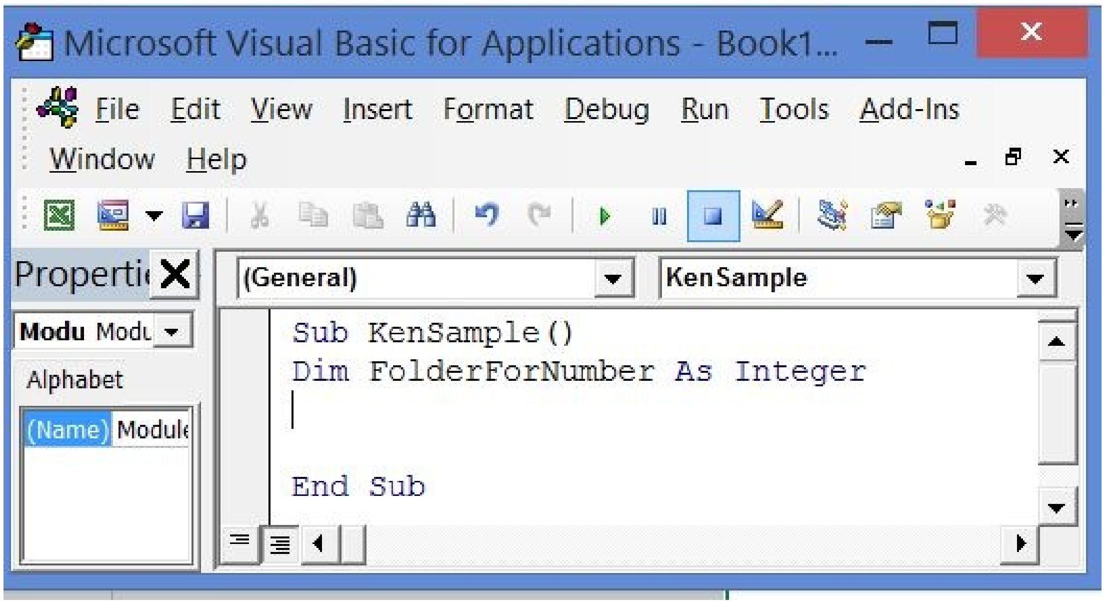
Dim means to create a new folder or sub-container; next, you type the new name followed by the word "As," and lastly, you decide if the new folder or sub-container will hold a number only, a text string only or numbers, text strings and objects like input boxes.

If your new folder or sub-container will only contain a numeric character, e.g., 1, 3, 45, etc…, you'll indicate this by typing **Integer**, as in **Dim KSample As Integer**.

Sub KenSample()

Dim FolderForNumber As Integer

End Sub


Microsoft Visual Basic for Applications - Book1...
File Edit View Insert Format Debug Run Tools Add-Ins
Window Help
Properti
Modu Modu
Alphabet
(Name) Module
(General)
KenSample

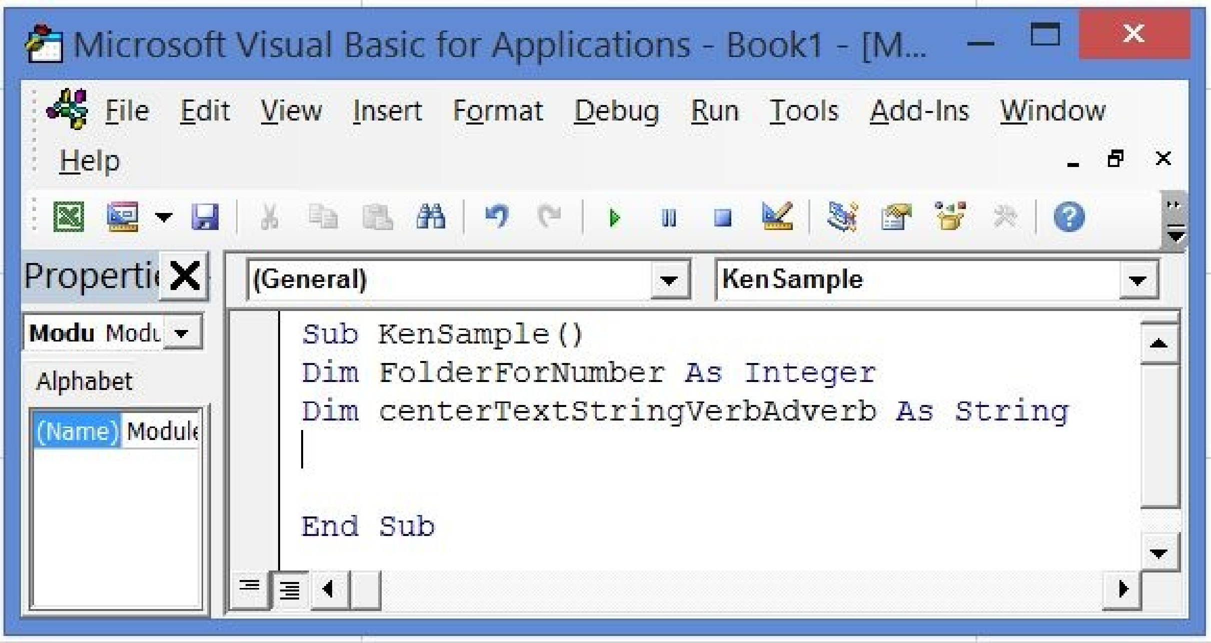
```
Sub KenSample()
Dim FolderForNumber As Integer

End Sub
```

Sample Object 1F: this shows user creating (e.g., Dim) the first container or folder that will hold a numeric character or number for VBA subroutine or code.

Now, we'll create another sub-container within the main container named **KennySample()**; the second, new, sub-container will be named **centerTextStringVerbAdverb**; since it will hold a text string, we indicate this by typing the new text string folder like this:

Dim centerTextStringVerbAdverb As String, as show in ***Sample Object 1G***.

```
Sub KenSample()
Dim FolderForNumber As Integer
Dim centerTextStringVerbAdverb As String
End Sub
```


Microsoft Visual Basic for Applications - Book1 - [M...
File Edit View Insert Format Debug Run Tools Add-Ins Window
Help
Properti
Modu Modu
Alphabet
(Name) Module
(General)
KenSample

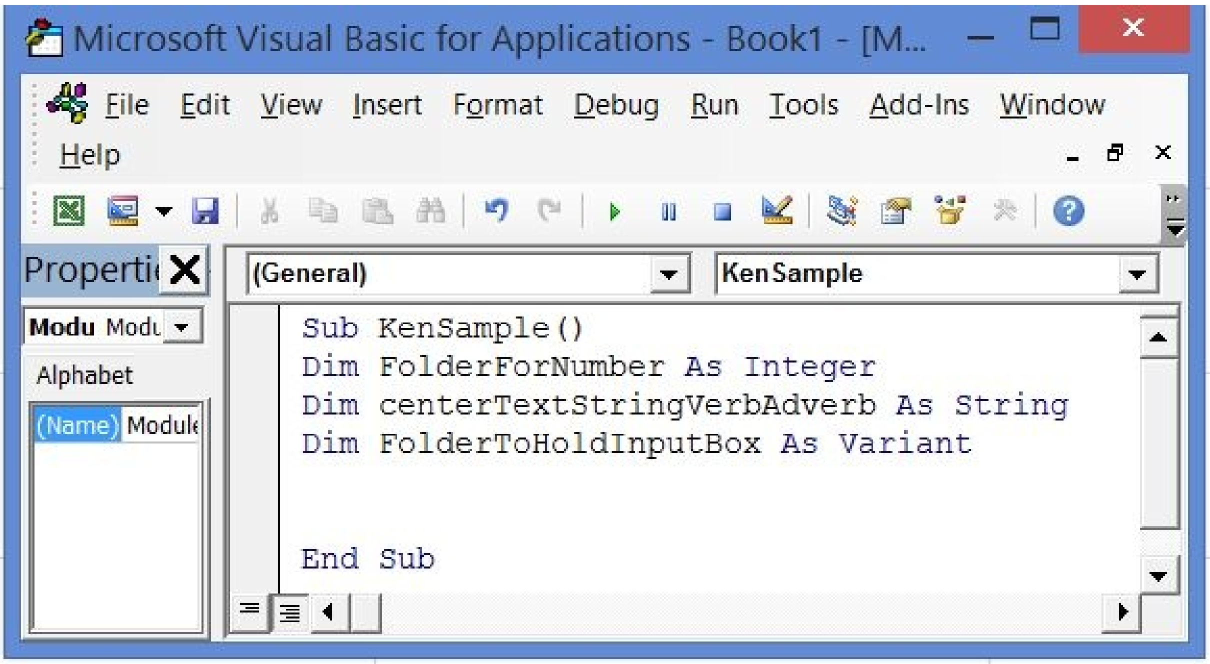
```
Sub KenSample()
Dim FolderForNumber As Integer
Dim centerTextStringVerbAdverb As String


End Sub
```

Sample Object 1G: this shows user creating (e.g., Dim) the second sub-container or folder that will hold a text string, e.g, "hello", "Me34y is my usersname", 123, "he he", etc…

It's time to create another sub-container within the main folder or container named **KennySample()**; the third, new, sub-container will be named **FolderToHoldInputBox**; since it will hold an InputBox, we indicate this by typing the folder like this:

Dim FolderToHoldInputBox As Variant, as shown in ***Sample Object 1H***. A variant is basically a variable. Cells A1, A4, etc… are variables. An input box stores whatever is typed into it; thus it becomes a variable; a variable aka a variant can be numeric, strings or even objects and arrays. Variants can hold other variants combined with strings or variants combined to other variants. Variants use lots of memory; strings use very little memory.

```
Sub KenSample()
Dim FolderForNumber As Integer
Dim centerTextStringVerbAdverb As String
Dim FolderToHoldInputBox As Variant
End Sub
```



```
Sub KenSample()
Dim FolderForNumber As Integer
Dim centerTextStringVerbAdverb As String
Dim FolderToHoldInputBox As Variant


End Sub
```

Sample Object 1H: this shows user creating (e.g., Dim) the third container or folder that will hold an InputBox.

It's time to put something inside of each of our newly created sub-containers. By using the **Dim** Statement, we've created three (3) different types of empty folders (Integer, String and Variant), but these are useless, until we fill them with something. Since an integer requires a numeric character (only), we'll put a number one (1) in the folder called FolderForNumber: **FolderForNumber = 1**.

We'll put a string in our second sub-container named centerTextString: **centerTextString = "is our # . "**

And, we'll put an input box inside of our third sub-container defined as a variant: **FolderToHoldInputBox = InputBox("Type a word ", " ")**.

Going forward, a sub-container will also be synonymous with subfolder.

CODE 1: objective is to use a prompt box to obtain a numeric value to select the font color; once this is done, the text that is already typed in cells A1 and B1 will be combined and the font color will be changed based on a number typed into the prompt or input box; the text will read: content in cell A1 ***is our # 1*** content in cell B1, i.e., if Kenny Keys is in cell A1, and Teacher is in cell B1, cell B1 will read: Kenny Keys is our #1 Teacher.

```
Sub KenSample()
Dim FolderForNumber As Integer
Dim centerTextStringVerbAdverb As String
Dim FolderToHoldInputBox As Integer
FolderForNumber = 1
centerTextString = ” is our #”
FolderToHoldInputBox = InputBox(“Type a number from 1 to 57 for font color”, ””)
MsgBox Range(“A1”).Value & centerTextString & FolderForNumber & ” ” &
Range(“B1”).Value
Range(“A2”).Value = Range(“A1”).Value & centerTextString & FolderForNumber
& ” ” & Range(“B1”).Value
Range(“A2”).Font.ColorIndex = FolderToHoldInputBox
End Sub
```

Let’s break **Code 1** down, line by line:

This line of code tells your application (Microsoft Excel) to create a new main container that will hold all of your requests and your commands; once it has done that, it is to name the container: KenSample(). By naming the container, it gives the container a unique name; this allows the application to find the storage container instantly. This will be the first task or command that will be fulfilled by your application, upon running your code. In programming/coding, each line of code is fulfilled by your application line by line (starting from the top line and doing down, line by line, until it reaches the very bottom or last line of code).

```
Sub KenSample()
```

This line of code tells your application (Microsoft Excel) to create a new subfolder or sub-container that will hold a single numeric integer, e.g., 1, 4, 56, 6464, etc…; this sub-container will be named ForlderForNumber. By naming the sub-container, it gives the container a unique name; this allows the application to find the storage container instantly; this will be the second task or command that will be fulfilled by your application, upon running your code. In programming/coding, each line of code is fulfilled by your application line by line (starting from the top line and doing down, line by line, until it reaches the very bottom or last line of code).

Dim FolderForNumber As Integer

This line of code tells your application (Microsoft Excel) to create a new subfolder or sub-container that will hold a single text string, e.g., 1, 4, 56, 6464, "My name is Kenny","pup", "d5k895kfdh", "purple6743", etc…; this sub-container will be named conterTextStringVerbAdverb. When you name a main container or sub-containers, you must never put spaces between characters; these are no no's: Sub Kenny Keys(), Sub myText 897(), kencontainer for integer, etc…; instead, if you want to indicate spaces in your name, you must not put a space, but you can show separation of characters by using an underscore, e.g., Sub Kenny_Keys(), Sub myText897(), kencontainer_for_integer, etc…. By naming the sub-container, it gives the container a unique name; this allows the application to find the storage container instantly; this will be the third task or command that will be fulfilled by your application, upon running your code. In programming/coding, each line of code is fulfilled by your application, line by line (starting from the top line and doing down, line by line, until it reaches the very bottom or last line of code).

Dim centerTextStringVerbAdverb As String

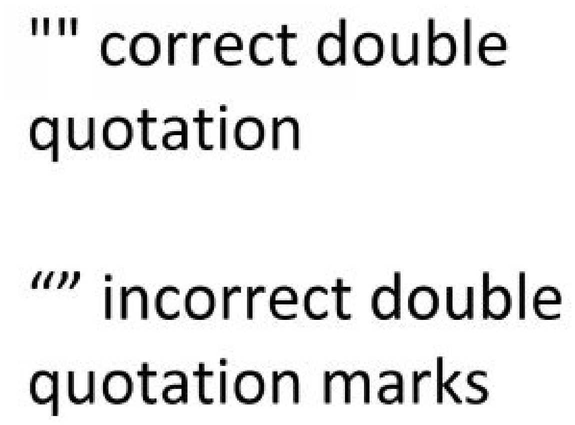
This line of code tells your application (Microsoft Excel) to create a new subfolder or sub-container that will hold a single numeric character, e.g., 1, 4, 56, 6464, etc…; this sub-container will be named FolderToHoldInputBox. When you name a main container or sub-containers, you must never put spaces between characters; these are no no's: Sub Kenny Keys(), Sub myText 897(), kencontainer for integer, etc…; instead, if you want to indicate spaces in your name, you must not put a space, but you can show separation of characters by using an underscore, e.g., Sub Kenny_Keys(), Sub myText897(), kencontainer_for_integer, etc…. By naming the sub-container, it gives the container a unique name; this allows the application to find the storage container instantly; this will be the fourth task or command that will be fulfilled by your application, upon running your code. In programming/coding, each line of code is fulfilled by your application, line by line (starting from the top line and doing down, line by line, until it reaches the very bottom or last line of code).

Dim FolderToHoldInputBox As Integer

This line of code will be the fifth line to be fulfilled by the application. We've already created an empty folder to hold a numeric character, e.g., 1, 4, 59, 453, etc…; now, we need to put something in that container, or it's useless. It's like having an empty bottle with the name "Kenny's Delicious Red Drink" on the label, but the bottle is completely empty. We want to start our loop and a specific row or column, in the spreadsheets, so this number must start with 1 or higher to be effective for looping in the spreadsheet.

FolderForNumber = 1

This line of code will be the sixth line to be fulfilled by the application. We've already created an empty folder to hold a text string, e.g., 1, 4, 59, 453, "kenny", "my 1st date", "Mia 6789", etc.… A text string can be numbers, letters, special characters, spaces with letters and numbers, etc.… The only difference with how to type a text string for all numbers versus for text and numbers is this: numbers don't require double quotation marks around them, unless they include alpha or special characters and spaces; any other string that isn't all numeric characters will need to be enclosed between the double quotation marks found in the Notepad or in the VBA Module window. Microsoft Word's double quotation marks will not work.



"" correct double quotation

“” incorrect double quotation marks

Sample Object 1I: This image shows the correct and incorrect quotation marks to use for coding in the VBA Module window.

Now, we need to put something in our new text string container. We've chosen to start the text string off with an empty space and end it with an empty space. This way, when we connect it to other text strings and numbers entered via the input box and the integer number, we don't have to worry about words running together. We'll be putting ” is our # ” in the container called centerTextString, as shown below:

centerTextString = ” is our # “

This line of code will be the seventh line to be fulfilled by the application. We've already created an empty folder to hold an integer, e.g., 1, 4, 59, 453, etc.… This integer will be collected by a prompt box requesting that the user enter a number from 1 – 56; this number will correspond to a color from the built-in color pallet built into the Microsoft Excel application.

FolderToHoldInputBox = InputBox(“Type a number from 1 to 56 for font color”, ””)

This line of code will be the eighth line to be fulfilled by the application. We've already created all the empty sub-containers, and we've already filled them with either numeric characters or text strings. In cell location A1, you will need to type someone's name, e.g., Margaret, Kenny Keys, Harold, etc…; in cell location B1, you'll need to type a career title or job title, e.g., teacher, musician, doctor, etc…. Now, let's say that we did the following:

Cell A1 = Betty Carter

Cell B1 = music teacher

FolderForNumber = 1

centerTextString = " is our # "

So, the message box will be combining these together using the ampersand (&) symbol. An ampersand is only allowed to touch a variable container name, an array container name or a cell location, e.g., A1, A45, E54, etc…, without them being between double quotation marks; it can only touch text strings that haven't been put into cell locations or pre-named containers by putting them between double quotation marks (" "), e.g., "Mavis", "Henry is 2 years old ", "JRam 67*33:,U9", etc… In this case, we only need an empty space to put between the number 1 that is put into the container named FolderForNumber and the cell contents already typed into cell B1. Whenever you combined containers and cell locations, you will only see what is inside those containers and cells; you won't see cell names in your worksheet, Message Boxes, Input Boxes or Confirmation Boxes. This is the result of this line of code: **Betty Carter is our # 1 music teacher**

This is the eighth line of code that results in the example above being shown in a message box display:

MsgBox Range("A1").Value & centerTextString & FolderForNumber & " " & Range("B1").Value

This is the ninth line of code that will be fulfilled by the application. I won't explain this one, because it will be the same result as the eight line of code above it; however, this will be written out in cell location **C1** of your active spreadsheet. It will read as follows: **Betty Carter is our # 1 music teacher**

Below, you see the ninth line of code that results in the example shown above. The color coding (above) is just so the user can easily identify different container contents and cell location contents in the above example.

Range("A2").Value = Range("A1").Value & centerTextString & FolderForNumber & " " & Range("B1").Value

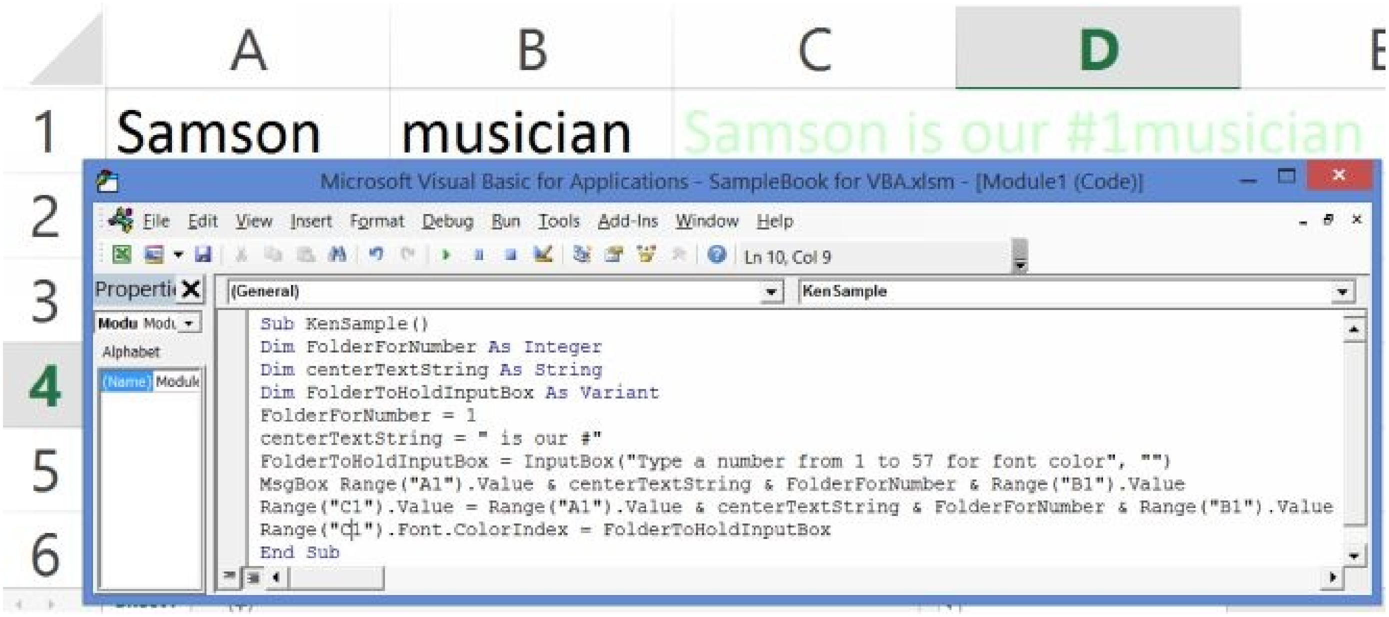
Below, you see the tenth line of code. This line instructs your application to find the contents in spreadsheet cell location **C1**; after found, the application is to change the font color based on the built-in color index chart for the numeric character found in the container named **FolderToHoldInputBox**. Example: if the folder contains a number 1; that number corresponds to the color BLACK in the color index pallet; thus, the text will be changed to black.

Range("A2").Font.ColorIndex = FolderToHoldInputBox

This is the eleventh line of code, and it commands your application to stop accepting further instructions or commands for the main container called KenSample(), e.g., Sub KenSample().

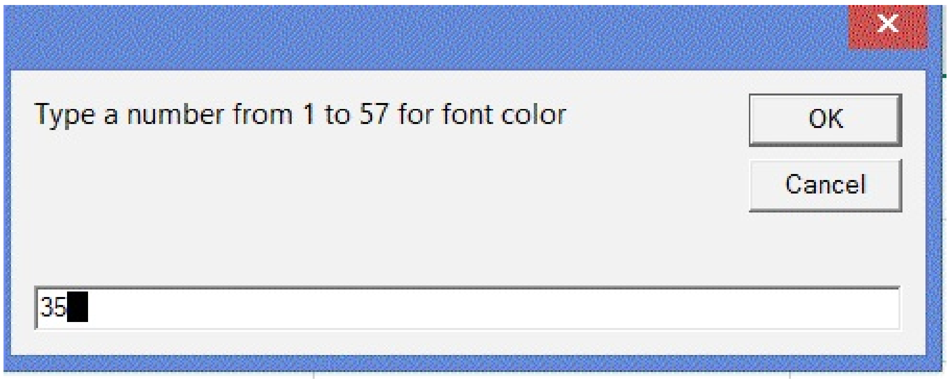
End Sub

Now, it's time to see the results of our code; to do this, we select **Run** from the drop-down menu option and choose **Run Sub/UserForm**, as shown in ***Sample Object 1J***. You can see the result of running or executing this code in ***Sample Object 1K-1M***.

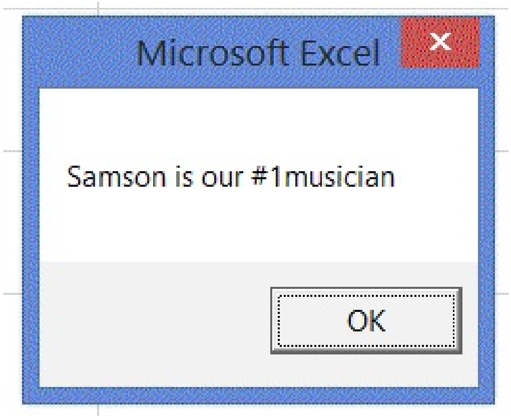

A
B
C
D
E
1
Samson
musician
Samson is our #1musician
2
3
4
5
6
Microsoft Visual Basic for Applications - SampleBook for VBA.xlsm - [Module1 (Code)]
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
Ln 10, Col 9
Properties
(General)
KenSample

```
Sub KenSample()
Dim FolderForNumber As Integer
Dim centerTextString As String
Dim FolderToHoldInputBox As Variant
FolderForNumber = 1
centerTextString = " is our #"
FolderToHoldInputBox = InputBox("Type a number from 1 to 57 for font color", "")
MsgBox Range("A1").Value & centerTextString & FolderForNumber & Range("B1").Value
Range("C1").Value = Range("A1").Value & centerTextString & FolderForNumber & Range("B1").Value
Range("C1").Font.ColorIndex = FolderToHoldInputBox
End Sub
```

Sample Object 1K: this shows user running or executing the code by pressing the drop-down menu option named Run and clicking selecting Run Sub/UserForm.

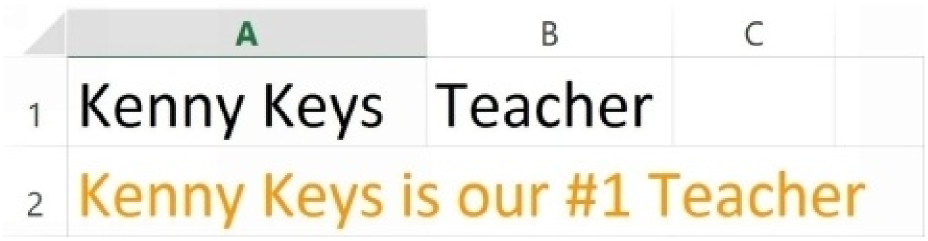


Sample Object 1L: this shows the InputBox, after user runs or executes the VBA code.



Samson is our #1musician

Sample Object 1M: this shows the MsgBox after user executes code.

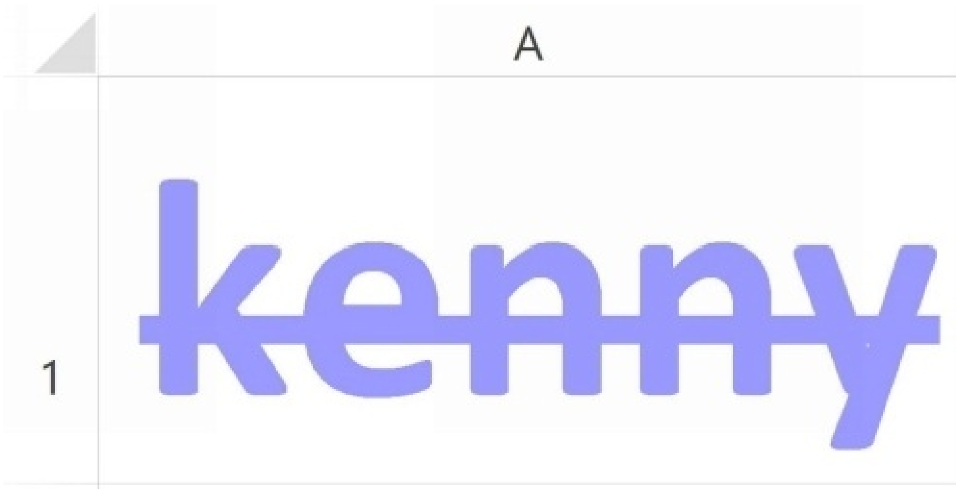


	A	B	C
1	Kenny Keys	Teacher	
2	Kenny Keys is our #1 Teacher		

Sample Object 1N: this shows the third and fourth instructions of the code being executed; the number entered into the InputBox determines the font color. This print screen was produced by entering different info.

CODE 2:

```
Sub kit()
Selection.Value = “kenny”
Set kenshortcut = Selection.Font
kenshortcut.ColorIndex = 17
kenshortcut.Bold = True
kenshortcut.Strikethrough = True
End Sub
```



Sample Object 1O: the shows the result of CODE 2, after running it.

Code 2 will allow the user to select one or more cells, simultaneously; after running the code, the cell or cells will have the name kenny typed into them; the font will be set to number 17 in the color index, built-in color pallet; the text will be bolded, and a strikethrough line will be drawn through the name kenny, e.g., ~~kenny~~.

I'll explain it, line by line: this creates a main container like a file cabinet to store all of your requests or commands to the Microsoft Excel application.

Sub kit()

This tells the application to allow the user to select one or more cells simultaneously to be impacted by the code instructions that follows, e.g., A1, A1 and A4 and G6, etc… Whatever cell or cells are selected will have the name kenny typed into them.

Selection.Value = "kenny"

This tells the application to create a container to hold the 1st portion of the folder path for font attributes. In VBA, in order to instruct your application to do something for you, you have to tell it what you want; this means that if you want your font impacted by your command, you have to start with the file location of that command. For font, it always starts off with Font as the parent folder, followed by the specific subfolder that you want. I'll continue explaining in the next line of code, but for this line of code, I want whatever cells are selected in this spreadsheet to pre-store the Font property, so we don't have to keep typing it for each line of code. We give this invisible container the name **kenshortcut;** this container can only be seen by the computer. Now, each time that we type kenshortcut, it is to be replaced by the file path Selection.Font.

Set kenshortcut = Selection.Font

This line of code tells your application (Microsoft Excel) to replace kenshortcut with Selection.Font and add on the ColorIndex subfolder, and set the color to 17, based on the internal, built-in color pallet. Anytime that you see an equal sign, you know that whatever is on the right of it is what you'll see displayed in your message boxes, prompt boxes, spreadsheet cell locations, etc…, e.g., color, size, bold, etc….

kenshortcut.ColorIndex = 17

This line of code tells your application (Microsoft Excel) to replace kenshortcut with Selection.Font and add on the Bold subfolder, and set the Bold to True. Anytime that you see an equal sign, you know that whatever is on the right of it is what you'll see displayed in your message boxes, prompt boxes, spreadsheet cell locations, etc…, e.g., color, size, bold, etc….

kenshortcut.Bold = True

This line of code tells your application (Microsoft Excel) to replace kenshortcut with Selection.Font and add on the Strikethrough subfolder, and set it to True, i.e., as long as you don't type false, the application will default to True. Anytime that you see an equal sign, you know that whatever is on the right of it is what you'll see displayed in your message boxes, prompt boxes, spreadsheet cell locations, etc…, e.g., color, size, bold, etc.…

kenshortcut.Strikethrough = True

'This line of code tells your application (Microsoft Excel) that you are done making requests or demands for now.

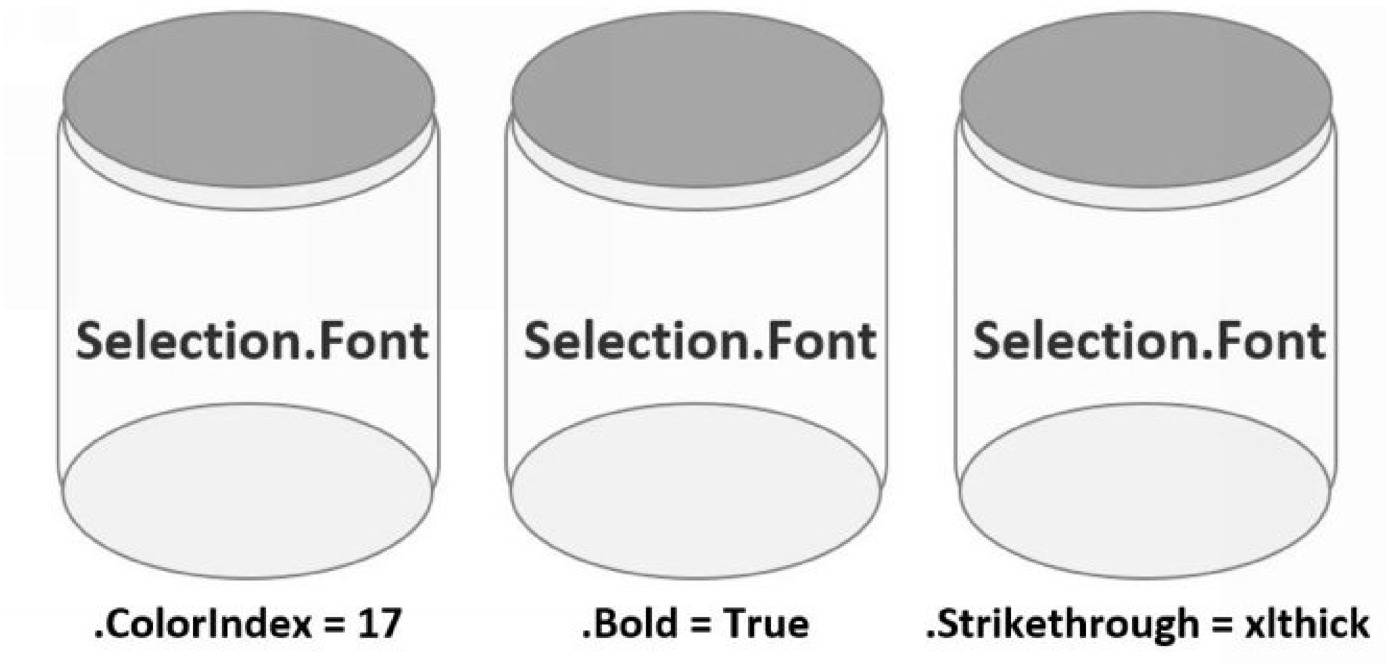
End Sub

Now that I've explained two (2) complete VBA codes in detail, line by line, it's time to manifest my container version of simplifying VBA in a visual form, so users understand my way of tutoring or teaching VBA even better.

Below, I've inserted an image of factitious containers to better help new learners of VBA identify VBA to something that we all know so well—containers. Let us assume that I took a glass gar from the cabinet, and I wanted to store single lines of VBA codes in each jar. In this case, each jar would hold the 1st folder path to accessing font properties. In VBA, every actions is commanded by providing the application (Microsoft Excel) with the exact folder location of the action that users wants to be executed or fulfilled. Selection simply tells your application that you don't want to have to name an exact spreadsheet cell location that will be impacted by your code; instead, you want to choose whatever cell location you desire; this can change each time. You can select a single cell location, e.g. A1, or you can select multiple cell locations, simultaneously, e.g., A1, G5, F34, A3:H10, etc…. Once you've selected the cell or cells that you want to have your code impact, the next folder path is to choose the main font folder that holds many subfolders inside of it; each subfolder is responsible for different actions that affects your spreadsheet's cell location font.

So, Selection.Font will be given the name kenshortcut; we do this so we don't have to retype the same beginning file path locations two (2) or more times. The container or imaginary empty glass jar will be filled with the first portion of the file path Selection.Font; on the outside of the jar, we'll put an imaginary label on it, and it will read: kenshortcut.

Once we've created our 1st jar, we'll just do some magic and tell that jar to duplicate itself, each time that we want to go into another subfolder for the font property, e.g., ColorIndex, Size, Strikethrough, Bold, Italic, etc…



Sample Object 1P: This image is a fictional depiction of how your application (Microsoft Excel) stores the first part of a font folder pathway, so the user doesn't have to keep retyping that part of the file pathway each time.

Now that we've created the container that will be storing the first portion of our Font property file pathway, we duplicate that jar two (2) more times, until we have three (3) jars that look exactly the same.

```
.ColorIndex = 17
.Bold = True
.Strikethrough = True
```

As we've already discussed earlier in this chapter, whatever is on the left of the equal (=) sign is the folder name to be filled with instructions; since these are Microsoft VBA built-in folder pathways that user don't create themselves, but instead they access, we have to fill them with the choices or selections that Microsoft has provided to us for the Microsoft Excel spreadsheet Font properties. In these cases, we want to change the color of the font to the color on the built-in color pallet that corresponds to the number 17. And, we want the text to be bold, so we type TRUE; even if we typed something else besides True, the application would read it as True, as in xlThin or xlThick; however, if we didn't want the text to be bolded, we'd type FALSE. Last, we want to strike out the text, so we tell the application to do this the same as we did with the Bold, i.e., True for strikethrough, and False for no strikethrough. You can also type something like xlThin or xlThick to substitute for True, as shown in my example image.

The next step is to type three (3) different subfolder pathways under each jar:

Selection.Font

Selection.Font

Selection.Font

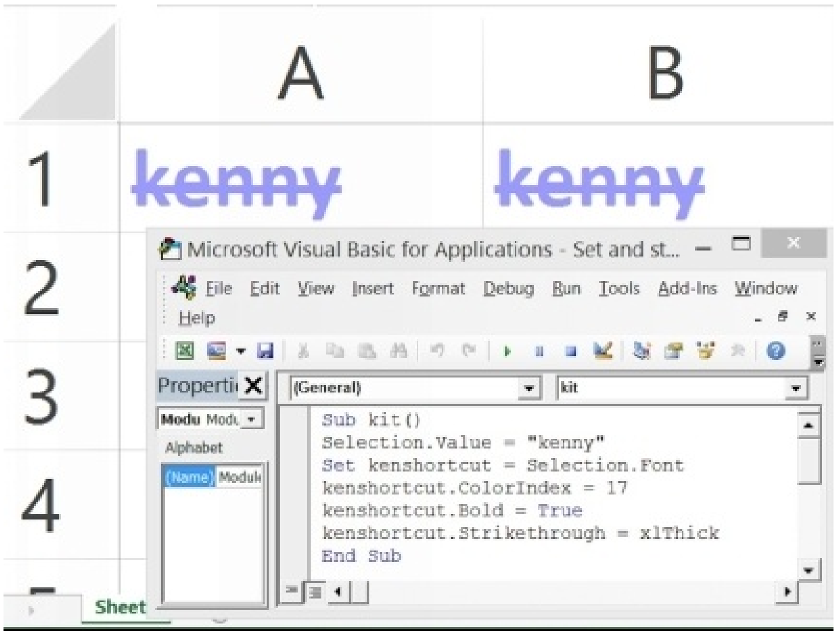
Next, we add on the rest of the font pathways to each of the samples above, as shown below:

Selection.Font.ColorIndex = 17

Selection.Font.Bold = True

Selection.Font.Strikethrough = True

The result of the code can be seen in Sample Object 1Q.


A
B
1
2
3
4
kenny
kenny
Microsoft Visual Basic for Applications - Set and st...
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
Properti
Modu Modu
Alphabet
(Name) Modul
(General)
kit
Sheet

```
Sub kit()
Selection.Value = "kenny"
Set kenshortcut = Selection.Font
kenshortcut.ColorIndex = 17
kenshortcut.Bold = True
kenshortcut.Strikethrough = xlThick
End Sub
```

Sample Object 1Q: This image shows the result of CODE 2.

Now that we've learned how to use the **Selection** property, we'll learn how the **ActiveCell** property is used next. The **ActiveCell** property works exactly like the **Selection** property; the only difference is that it is written differently. In the case of **Code 3**, we're also going to learn how to use the **With** Statement in conjunction to the **ActiveCell** property.

CODE 3: objective is to put the cursor on any cell in the spreadsheet, e.g., A2, and have the text string **"Hee Hee"** typed in cell A2 by the application; that same cell will have the text string in cell A2 bolded with a strikethrough line through it.

```
Sub KenWithStatement()
ActiveCell.Value = "Hee Hee"
With ActiveCell.Font
.ColorIndex = 33
.Bold = True
.Strikethrough = True
End With
End Sub
```

It's time to break **Code 3** down, line by line:

We already know that this first line of code creates a new main container that will hold all sub-containers; all sub-containers will store integers, variants and text strings.

Sub KenWithStatement()

This second line of code assigns a value to whatever spreadsheet cell that your cursor is on, when you run this complete code. Example: if your cursor is on cell B12, the word Hee Hee will appear in that cell, after this code is ran.

ActiveCell.Value = “Hee Hee”

This third line of code uses the application’s (Microsoft Excel’s) built-in statement called the **With** Statement. The **With** Statement in conjunction with the **ActiveCell** property allows the user to forgo retyping the first portion of the folder pathways for accessing actions and command needed for instructing the application to do as the user (e.g., you) demands or commands. So, using this third line of code, the application grants us permission to forgo retyping **ActiveCells.Font** for any other lines below this one; this only ends when the user ends the **With** Statement by typing **End With**.

With ActiveCell.Font

This fourth line of code only requires us to continue the subfolder pathways to access specific Font property commands or instructions. In this case, we are continuing to instruct the application to enter the built-in Font subfolder named **ColorIndex**; once we enter this folder, we tell the application to search the built-in color pallet index and select the color that corresponds to the number **33**; whatever color corresponds to number 33 will be the color of the font located in whatever spreadsheet cell location that our cursor is in when we run this complete code.

.ColorIndex = 33

This fifth line of code only requires us to continue the subfolder pathways to access specific Font property commands or instructions. In this case, we are continuing to instruct the application to enter the built-in Font subfolder named **Bold**; once we enter this folder, we have two options: True or False. If we type **true**, after the equal (=) sign, the text in whatever cell our cursor is in when this complete code is ran will be bolded; if we type **False**, after the equal (=) sign, the text in whatever cell our cursor is in when this complete code is ran will not be bolded.

.Bold = True

This sixth line of code only requires us to continue the subfolder pathways to access specific Font property commands or instructions. In this case, we want to enter the application’s (Microsoft Excel’s) built-in Font subfolder called **Strikethrough**; once inside this folder, we can tell the application to make a strike through thetext in our active cell by typing **True** after the equal (=) sign; to remove the strike from the text, we type **False**, after the equal (=) sign.

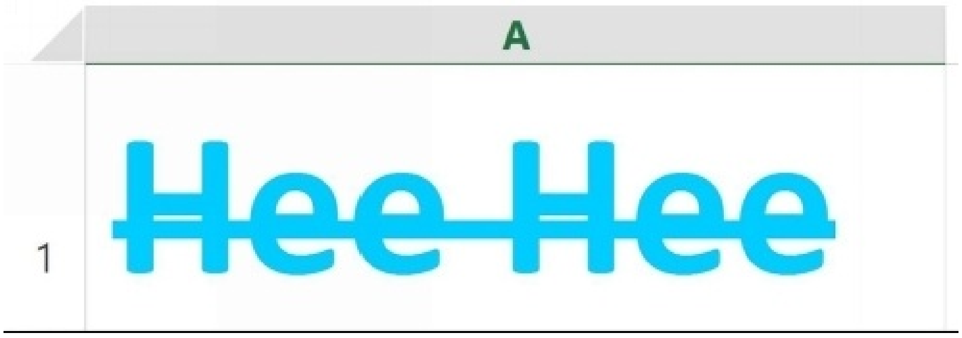
.Strikethrough = True

This seventh line of code tells the application that you no longer need to add any more Font subfolder requests or commands for your application to execute.

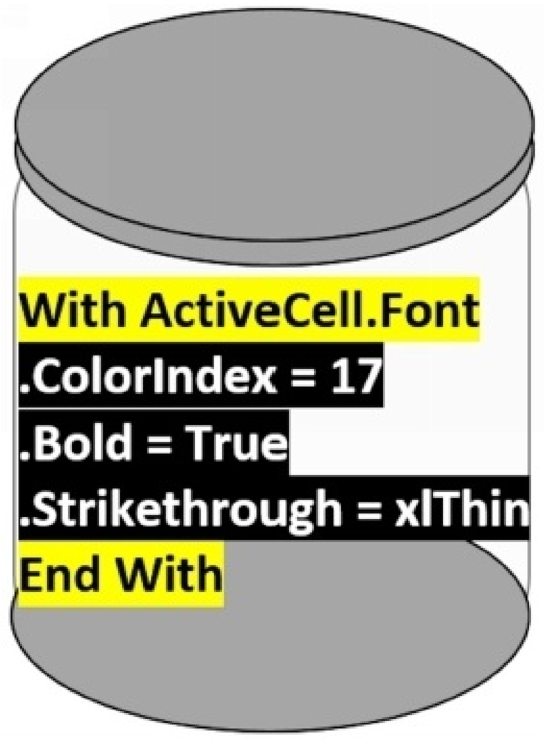
End With

Woo Hoo! This eighth line of code tells your application that it can take a break, because we don't have any more instructions to place inside this main command container named **KenWithStatement()**. In other words, this eighth and last line of code tells your application that the full code is complete and ready to be executed upon your request.

End Sub



Sample Object 1R: This image shows the result of CODE 3.



```
With ActiveCell.Font
.ColorIndex = 17
.Bold = True
.Strikethrough = xlThin
End With
```

Sample Object 1S: This image shows how the **With** Statement in conjunction with the **ActiveCell** and **Font** property code would look, if it was placed in a glass jar to be used by the user, when he or she was ready to use it. It's like storing your family's jam and jelly, until you need it.

Well, everyone, it’s been nice, but this chapter is complete. I’ll meet you all in the next chapter on Microsoft Excel Loop Statements. I’ll see you there! Until then, A bientot! (See you soon!)

Loop Statements: Chapter 2

Welcome to the chapter on VBA Loop Statement; in this chapter, you'll learn how to use five different types of loop statement: While Wend (my favorite), For Next, Do While, Do Until and For Each loop statements.

While Wend Loop Statements: Chapter 2A

Welcome to section A. This is the section on While Wend Loop Statements.

This first While Wend code will be explained line by line; however, it won't be necessary for me to over explain each line of this code, because I've already done that in the previous chapter on ***Getting Acquainted with VBA Integers, Variants, Text Strings, Ranges, Cells, Message Boxes and Input Boxes***. What I'll be explaining in detail are the lines that effect the actual loop statement. Now, let's get started.

CODE 4: objective is to loop through cells in column A, e.g., A1: A4, and change the cell's background color to 33, in each cell, within column A, to correspond with the built-in color index chart for this application; this will be done as long as there are no empty cells within the range. The cell that is empty will stop the loop. Before running this code, you'll need to type something in cells A1:A6.

```
Sub KenRowLoop()
Dim RowKenLoop As Integer
RowKenLoop = 1
While Cells(RowKenLoop, 1) <> ””
Cells(RowKenLoop, 1).Interior.ColorIndex = 33
RowKenLoop = RowKenLoop + 1
Wend
End Sub
```

This first line of code instructs the application to create a main container to hold all of your sub-containers (see Chapter 2).

```
Sub KenRowLoop()
```

This second line of code instructs your application to create a new container and give it the name RowKenLoop; this container will only hold a single numeric character.

Dim RowKenLoop As Integer

This third line of code instructs your application to fill the newly created sub-container named RowKenLoop with the number 1.

RowKenLoop = 1

This fourth line of code will be explained in detail; this is the beginning of the **While Wend** statement. While is always to be proceeded by a criteria or logical statement. In other words, imagine that I wanted to give my nephew $300 each month, as long as he completed a certain amount of choirs each day; the total amount of choirs each day that he must complete are 2 choirs each day for (at least) 28 days out of each month. If he neglects to complete 2 choirs on the 28th day, he forfeits his allowance of $300, even though he successfully completed his 2 choirs the other 27 days. In computer language, the application is absolute about following your commands. It's not a human, so it doesn't sympathize with my nephew for falling short of completing his choirs 1 day out to the required 28 days, in order to receive his allowance for the month. In the case of the code below, we are instructing the application to execute this loop, as long as the content in cells 1, column 1, which is cell A1, is not equal to empty; this is what the empty set of double quotation marks mean. Remember that **RowKenLoop** has been set to 1; the Cells() Function in VBA has 2 requirements that are mandatory: rows and columns. This is how it looks: **cells(row, column)**. *Row is always the first criteria*, and *column is always the second criteria*; this never changes. So, for now, RowKenLoop is equal to 1; this means that the cells function is interpreted by your applications like this: **cells(1,1)**. Once again, the first row and first column in any Microsoft Excel spreadsheet is cell **A1**; thus, the **While** statement *is demanding that cell A1 not be empty*, in order to executed the instructions that will following, in the next line of code.

While Cells(RowKenLoop, 1) <> ""

This fifth line of code is instructing your application to go to Row 1, Column 1, which is cell A1, and make the color of the font equal to the number 33 on the application's built-in color index pallet (see Chapter 2).

Cells(RowKenLoop, 1) .ColorIndex = 33

This sixth line will be explained in detail. This lines instructs your application (Microsoft Excel) to find the container named RowKenLoop and add the number that is currently inside it to the number 1; *currently, the number in **RowKenLoop** is the number 1*, so the applications interprets this request as follows: ***1 = 1+1***. Of course, this will change the value inside the RowKenLoop container to the number 2. So now the application will repeat the code above by looking in row 2, column 1, which is cell A2, and it will change the font color in that cell to correspond to number 33 on the built-in color index pallet.

This process will continue to repeat itself, e.g., A3, A4, A5, etc…, until it comes to a row in column A that is empty, i.e., the row will not have anything typed in it. Let's assume that Cells A1:A6 has something typed in them, but cell A7 is empty. Cells A1:A6 will have their font color changed to correspond to number 33 on the color index pallet, but the loop will stop at cell A7; this means A7 and all cells below it will not have their font color changed.

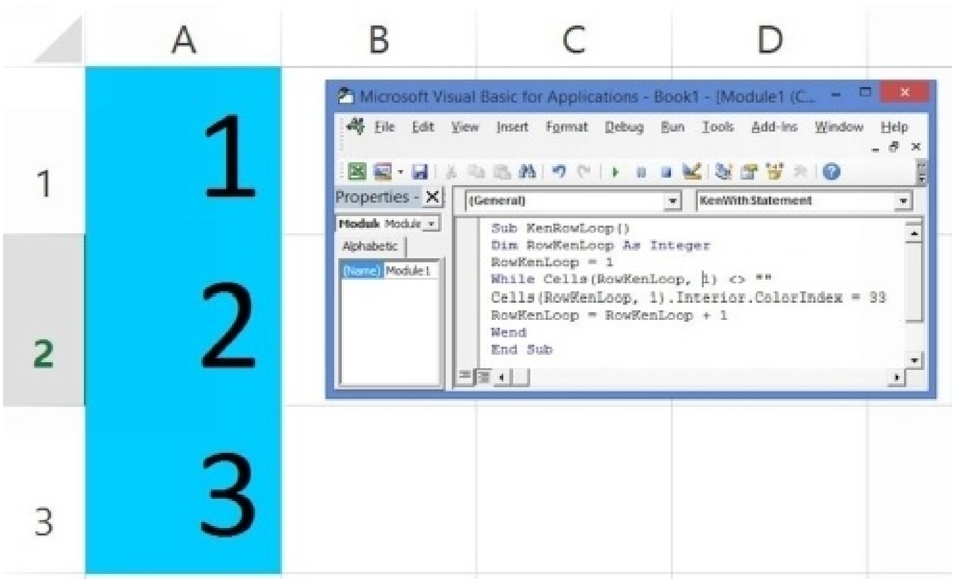
RowKenLoop = RowKenLoop + 1

This seventh line of code simply tells the application (Microsoft Excel) that there are no more instructions to be executed in this particular loop statement. This means end of loop instruction. So your application will only execute loop instructions between the beginning **While** Statement line of code and this **Wend** Statement line of code.

Wend

This eight line instructs your application that there are no more instructions for this main container called KenRowLoop() to be executed.

End Sub


A
B
C
D
1
2
3
1
2
3
Microsoft Visual Basic for Applications - Book1 - [Module1 (C...
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
Properties - Module1
Module Module
Alphabetic
(Name) Module1
(General)
KenWithStatement

```
Sub KenRowLoop()
Dim RowKenLoop As Integer
RowKenLoop = 1
While Cells(RowKenLoop, 1) <> ""
Cells(RowKenLoop, 1).Interior.ColorIndex = 33
RowKenLoop = RowKenLoop + 1
Wend
End Sub
```

Sample Object 1T: This image shows the result of Code 4, after it has been run.

CODE 5: object is to change the cell's background color in all the columns in row 1 that contains a numeric value, as long as they're not empty, i.e., from cells A1:C1, the background color of the cell will be changed to correspond to the numeric value 44 on the built-in color index pallet. Before running this code, you must type something in cells A1:C1.

Sub KenColLoop()

```
Dim ColKenLoop As Integer
ColKenLoop = 1
While Cells(1, ColKenLoop) <> ””
Cells(1, ColKenLoop).Interior.ColorIndex = 44
ColKenLoop = ColKenLoop + 1
Wend
End Sub
```

This first line of code instructions the application (Microsoft Excel) to create a main container to hold all sub-containers, instructions and commands to be executed by the application. The name of this main container is KenColLoop(), as shown below:

```
Sub KenColLoop()
```

This second line of code instructs the application to create a container to hold a numeric character, e.g., 1, 3, 546, etc…; this number will be used to loop through columns, e.g., column A, column B, etc….

```
Dim ColKenLoop As Integer
```

This third line of code instructs the application to fill the new container named ColKenLoop with the number 1.

```
ColKenLoop = 1
```

This fourth line of code instructs your application to start the While loop statement with a criteria that must be met, in order for the loop to even begin. In order for this loop to begin, the contents located in row 1, column 1 (because this is what is currently inside the container named ColKenLoop) must not be an empty cell; this means that cell A1 must not be empty, in order for the loop to begin; if the cell is empty, the loop will not begin.

```
While Cells(1, ColKenLoop) <> ””
```

This fifth line of code instructs the application to go to the content in row 1, column 1 and change the cells interior (background) color to match the number 44 on the built-in color index pallet.

```
Cells(1, ColKenLoop).Interior.ColorIndex = 44
```

This sixth line of code instructs the application to pull the current number in the container named ColKenLoop, which is 1, and add that to 1; thus, we have 1 = 1 + 1; this makes the number in the container named ColKenLoop the number 2. This number will continue to increase by 1 each time that a non-empty cell is present in row 1, e.g, A1, B1, C1, etc.… In this case, the next cell to be impacted will be B1; the next time, it will be C1, etc…

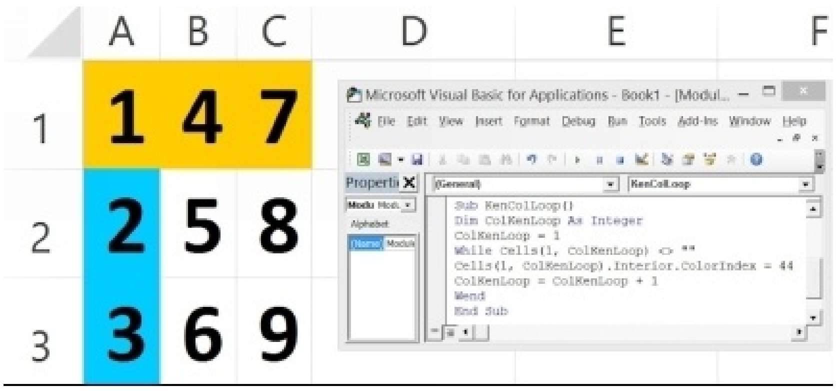
ColKenLoop = ColKenLoop + 1

This seventh line of code instructs your application to *end the While Statement*.

Wend

This eighth line of code instructs the application to close the main container called KenColLoop(), so no more commands can be added. This tells the application that this code can now be executed.

End Sub



	A	B	C
1	1	4	7
2	2	5	8
3	3	6	9

```
Sub KenColLoop()
Dim ColKenLoop As Integer
ColKenLoop = 1
While Cells(1, ColKenLoop) <> ""
Cells(1, ColKenLoop).Interior.ColorIndex = 44
ColKenLoop = ColKenLoop + 1
Wend
End Sub
```

D
E
F
Microsoft Visual Basic for Applications - Book1 - [Modul...
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
Properti
(General)
KenColLoop
Alphabet
(Name) Module

Sample Object 1U: This image shows the result of Code 5, after it has been run.

CODE 6: This example will show you how to only impact diagonal cells, e.g., A1, B2, C3, etc.… This particular code will shade the cells A1, B2 and B3, based on the number in those cells. The number in each cell will be matched to the built-in color index pallet and the cell will be shaded (background color) accordingly. Before running this code, you must type a numeric character in cells A1:C3, as shown in ***Sample Object 1V***.

Sub KenColRowLoop()

Dim ColKenLoop As Integer

Dim RowKKLoop As Integer

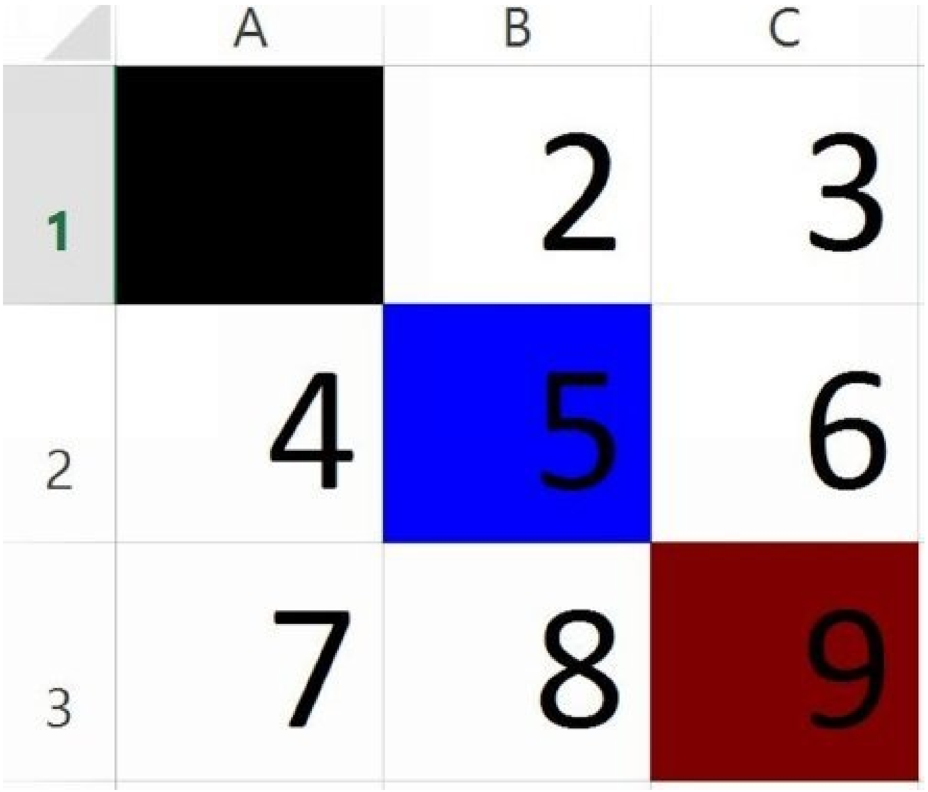
ColKenLoop = 1

RowKKLoop = 1

While Cells(RowKKLoop, ColKenLoop) <> ””

Cells(RowKKLoop, ColKenLoop).Interior.ColorIndex = _

```
Cells(RowKKLoop, ColKenLoop).Value
ColKenLoop = ColKenLoop + 1
RowKKLoop = RowKKLoop + 1
Wend
End Sub
```



	A	B	C
1		2	3
2	4	5	6
3	7	8	9

Sample Object 1V: This image shows the result of Code 6, after it has been run.

CODE 8: the objective of this code is to shade the background color in cells A1:C3 to match the numeric value in each of the cells in the range. Before running this code, you'll need to type unique numbers in each cell within the range A1:C3; the number must start with 1, but it may not exceed 56, as shown in ***Sample Object 1W***.

```
Sub KenColRowLoop()
Dim ColKenLoop As Integer, RowKKLoop As Integer
ColKenLoop = 1
RowKKLoop = 1
While Cells(RowKKLoop, 1) <> ""
If ColKenLoop > 3 Then
End If
For ColKenLoop = 1 To 3
```

```
Cells(RowKKLoop, ColKenLoop).Interior.ColorIndex = _
Cells(RowKKLoop, ColKenLoop).Value
Next
ColKenLoop = ColKenLoop + 1
RowKKLoop = RowKKLoop + 1
Wend
End Sub
```

This code allows the user to shade each cell on line 1, based on the number in that cell, e.g., cell A1 has a number 1, so the application will match 1 to the built-in color index pallet and change the background color of the cell to match the color pallet. It will move on to cell A2, A3, and then down to the next row, e.g., B1, B2, B3, etc…

This first line of code instructs the application (Microsoft Excel) to create a main container to hold all codes to be executed; it will be named KenColRowLoop(), as shown below:

```
Sub KenColRowLoop()
```

This second line of code instruct the application to create two (2) new sub-containers; both will store a number; one sub-container will be named ColKenLoop, and the other sub-container will be named RowKKLoop.

```
Dim ColKenLoop As Integer, RowKKLoop As Integer
```

This third line of code instructs the application to put a number 1 inside the sub-container named ColKenLoop.

```
ColKenLoop = 1
```

This fourth line of code instructs the application to put a number 1 inside the sub-container named RowKKLoop.

```
RowKKLoop = 1
```

This fifth line of code begins the While Statement; it instructs your application to begin the loop, if the contents in row 1, column 1 (e.g., A1) isn’t empty.

```
While Cells(RowKKLoop, 1) <> ””
```

This sixth line of code adds another criteria to the While Statement, on the fifth line of code. It tells the application that if the total number of columns from A1 rightwards is more than 3 columns, e.g., 4, 5, etc…, the loop must be stopped or cancelled. If the columns are 3 or under, the loop can begin.

If ColKenLoop > 3 Then

This seventh line of code is just the ending of If Statement, shown on the sixth line above.

End If

This eight line of code is the start of a For Next Statement nestle inside of a While Wend Statement. This provides even more assurance that the application (Microsoft Excel) will fulfill my request or demand exactly as I've instructed it to do. This adds to the Wend State of asking that Cell A1 not be empty, and that the columns, starting from A1 to C1, have data in them as well. This means 2 separate criteria must be met, before the looping process will start. If, at any time, both criteria cease to be met, the loop will stop.

For ColKenLoop = 1 To 3

This ninth line of code instructs the application to start the looping process in row 1, column 1; this means the process will start in cell A1, and if all requirements are met, cell A1 will have its background color changed to the color from the built-in color index pallet that corresponds to the number in cell A1 which is 1. Notice the underscore that follows the equal (=) sign; it instructs the application to allow the user to do a hard return to continue typing the rest of this line of code on the line below it, but the application still understands that it's one line of code. Without the underscore being put in the right place within the line of code, the user will get an error message, and the code won't work past this line. VBA codes do not wrap, so a code typed in Microsoft Word may wrap if the line of code is very long, but it must not be typed into the VBA Module window as wrapping. If you don't see an underscore, you must continue typing the code on the same line in the VBA module window, unless I instruct you to do otherwise.

Cells(RowKKLoop, ColKenLoop).Interior.ColorIndex = _

Cells(RowKKLoop, ColKenLoop).Value

This tenth line of code instructs the application to go to the next cell. It's already in cell A1, so the next cell will be B1.

Next

This eleventh line of code instructs the application to find the current number stored in the container named ColKenLoop, which is 1, and add 1 to it, e.g., 1 = 1 + 1; this means that, now, the container named ColKenLoop will hold a number 2 inside it. This is necessary to loop to the next column, e.g., B1, and after the loop’s requirements are met again, it will loop to cell C1.

ColKenLoop = ColKenLoop + 1

This twelfth line of code instructs the application to find the current number stored in the container named RowKKLoop, which is 1, and add 1 to it, e.g., 1 = 1 + 1; this means that, now, the container named RowKKLoop will hold a number 2 inside it. This is necessary to loop to the next row, e.g., B1; after the loop’s requirements are met again, it will loop to cell C1, but it can’t go to the next line until it has looped from A1 to B1 to C1; then, it will go to A2 then B2 then C2, etc…; this is the looping order or sequence for this particular loop statement.

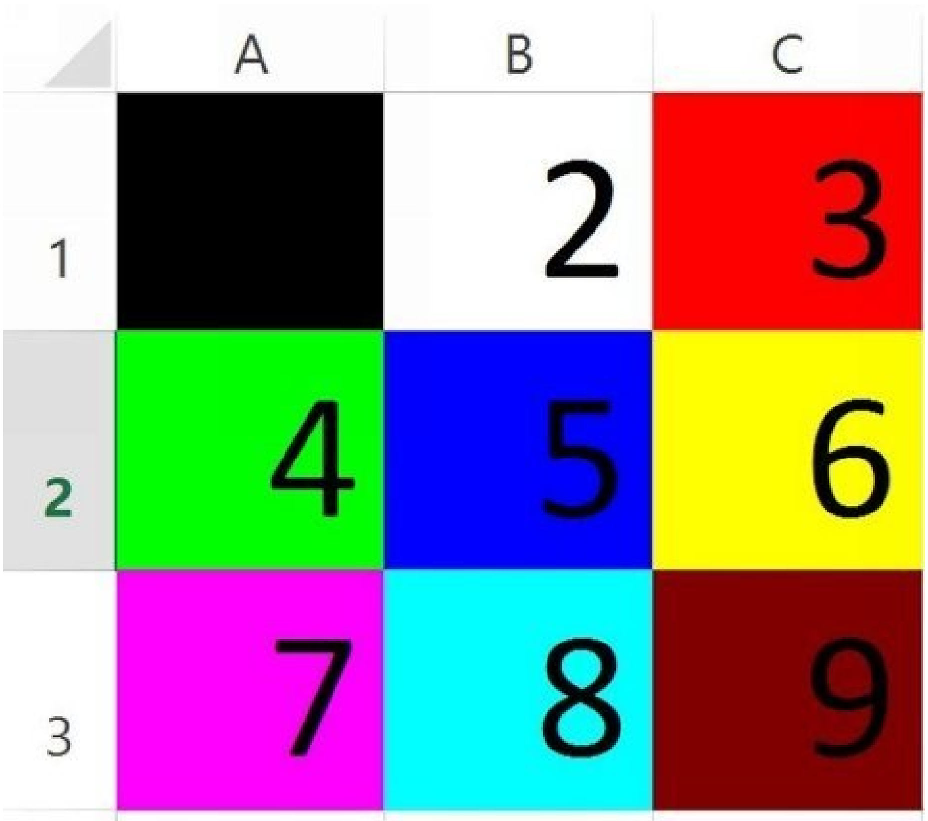
RowKKLoop = RowKKLoop + 1

This thirteenth line of code instructs the application to end the While Statement, so no more requests can be added.

Wend

This fourteenth line of code informs the application that the main container is now complete with its full code, and it’s ready to be executed.

End Sub



	A	B	C
1		2	3
2	4	5	6
3	7	8	9

Sample Object 1W: This image shows the result of Code 8, after it has been ran.

CODE 9: this code requires that you first type numbers in column A, so type 1 in A1, 2 in A2, 3 in A3 and 4 in A4, before you run this code. This code will change cell A1's background color to match number 1 on the built-in color index pallet; it will delay for 1 second, before it moves down to cell A2 where it will change cell A2's background color to match number 2 on the built-in color index pallet; it will delay for 1 second, and this will repeat until it has changed the cell's background color for cell A3 and A4.

```
Sub WaitTest()
Dim kee As Integer
kee = 1
While Cells(kee, 1) <> ””
Cells(kee, 1).Interior.ColorIndex = Cells(kee, 1).Value
Application.Wait (Now + TimeValue(“0:00:01”))
kee = kee + 1
Wend
End Sub
```

This first line of code instructs the application to create a main folder to hold all sub-containers and commands to be executed; the container is to be named WaitTest(), as show below:

```
Sub WaitTest()
```

This second line of code instructs the application to create a new sub-container named kee to hold a numeric character, e.g., 1, 4, 567, etc….

```
Dim kee As Integer
```

This third line of code instructs the application to put a number 1 inside the sub-container named kee, as shown below.

```
kee = 1
```

This fourth line of code instructs the application (Microsoft Excel) to start the loop statement. The requirement for this loop statement to begin and continue is that the cell located in row 1, column 1, which is cells A1, must not be an empty cell. If the cell has something typed in it, the loop code will continue.

```
While Cells(kee, 1) <> ””
```

This fifth line of code instructs your application to take the numeric value that is typed into row 1, column 1, which is cell A1, and compare it to the built-in color index pallet; when it finds that number in the pallet, it uses the color associated with that number to fill the background color in cell A1.

Cells(kee, 1).Interior.ColorIndex = Cells(kee, 1).Value

This sixth line of code is a built-in function within Microsoft Excel and the entire Microsoft Operating System. This delays all functions for the time that is requested by the user (e.g., you or me), based on the time value. In this case, I've ask the application to determine the exact time right now and add one (1) second onto that time. This means that my system will freeze all activity until one (1) second has elapsed. After that one (1) second has elapsed, the system activity will resume. This means that the color background in cell A1 won't change, until one (1) second has passed.

Application.Wait (Now + TimeValue("0:00:01"))

This seventh line of code instructs the application to find the number stored in the sub-container named kee, which is 1, and add one (1) to it; this means 1 = 1 + 1; we now have a number 2 in the container named kee; this is needed, so that the loop continues on the second row in the spreadsheet. We started off in cell A1; after adding one (1) to the value in the container named kee, we know have 2 in that container, so now the application will go to A2 and start the process all over again.

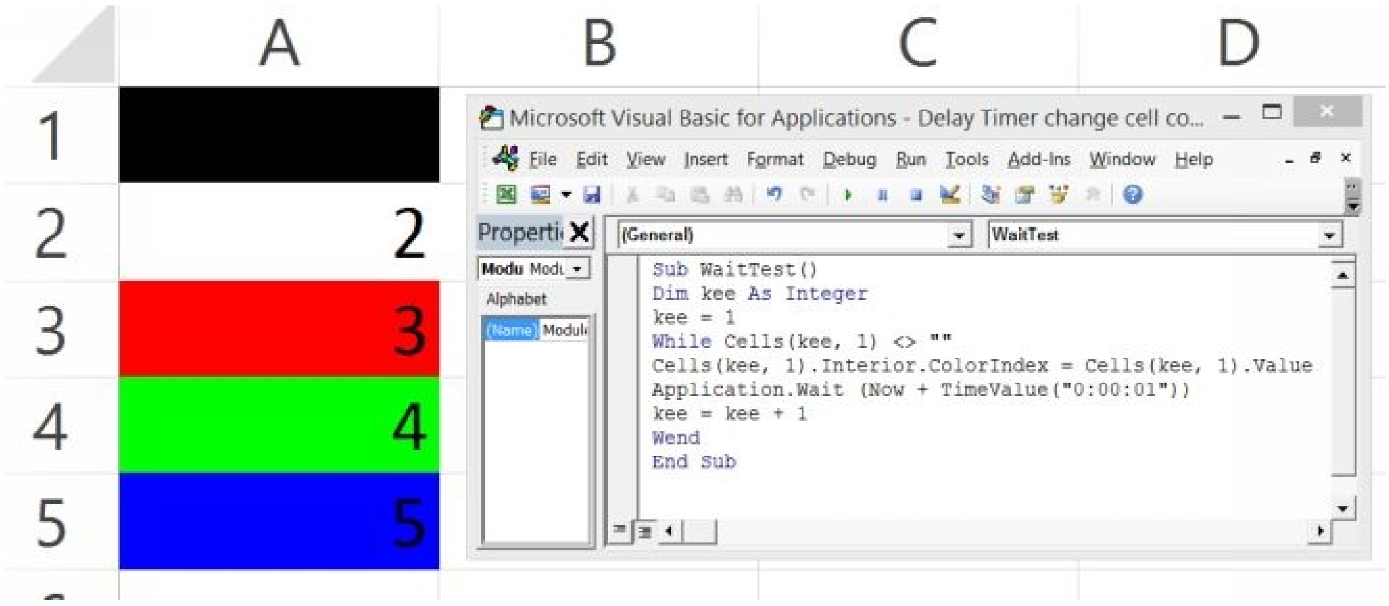
kee = kee + 1

This eight line of code ends the While Statement.

Wend

This ninth line of code closes the main container named WaitTest(); this means that the application can now run the full code.

End Sub



```
Sub WaitTest()
Dim kee As Integer
kee = 1
While Cells(kee, 1) <> ""
Cells(kee, 1).Interior.ColorIndex = Cells(kee, 1).Value
Application.Wait (Now + TimeValue("0:00:01"))
kee = kee + 1
Wend
End Sub
```

Sample Object 1X: This image shows the result of Code 9, after it has been ran.

CODE 10: before running this code, you must type three (3) names in the cell range A1:A3; at least two of the names should contain two (2) consecutive double n's, e.g., Kenny, Penny, etc.… This code finds each name in the range that has a double n, and it places a number 1 in the corresponding column C, as shown in ***Sample Object 1YA***. It then enters a COUNTIF formula in cell B1. This code then counts the number of ones placed in column C by the application; that number will be typed into cell D1 by the application; it will also produce a message box that will indicate how many names contain a double n, before it places that number in cell D1, because that is where your cursor was, prior to you running this code.

```
Sub kyyyy()
Dim kcnt As Integer, krow As Long
kcnt = 1
krow = Cells(Rows.Count, 1).End(xlUp).Row
While Cells(kcnt, 1) <> ””
If Cells(kcnt, 1).Value Like “*nn*” Then
Cells(kcnt, 3).Value = 1
End If
kcnt = kcnt + 1
Wend
Range(“B1”).Select
ActiveCell.FormulaR1C1 = “=COUNTIF(RC[-1]:R[2]C[-1],”””*nn*””)”
MsgBox “There are ” & krow & ” rows of data.”
```

Range(“D1”).Formula = “=SUM(” & Range(Cells(1, 3), Cells(krow, 3)).Address(False, False) & “)”

End Sub

This first line of code instructs the application to create a main storage container to hold all sub-containers and other commands; once the container is created, it is to be named kyyyy(), as shown below.

Sub kyyyy()

This second line of code instructs the application to create two new sub-containers: one is to be named kcnt to hold a number; the second sub-container is to be named krow; it will also hold a number, but the number may be a long number, e.g., 400, 5000, 70000, etc….

Dim kcnt As Integer, krow As Long

This third line of code instructs the application to fill the sub-container named kcnt with the number 1, as shown below:

kcnt = 1

This fourth line of code instructs the application to fill the sub-container called krow with the number that results from counting the total number of used rows (i.e., rows that have something typed in them) in column 1, which is column A. So, from column A1, if we have something typed in rows A1:A3, but row A4 is empty, the code would result in the number 3, because it noticed that row 3 was the last row to have content typed in it.

krow = Cells(Rows.Count, 1).End(xlUp).Row

This fifth line of code instructs the application to start the While Statement; this loop requires that A1 not be empty, in order for the loop to start or begin. The loop will end in the first empty cell, within column A that it comes to.

While Cells(kcnt, 1) <> ””

This sixth line of code instructs the application to search in each cell located in column 1 or column A; it is looking for any text string, e.g., Kenny Keys, Johnny Cash, a penny in the minny, etc…, that contains double “n” characters; when it finds them, it will do what is instructed in the seventh line of code, below this line of code.

If Cells(kcnt, 1).Value Like “*nn*” Then

This seventh line of code instructs the application to type a number 1 in column 3 or column C that corresponding to the rows in column A where a double "nn" text string is found, i.e., if cell A2 contains "Kenny," and cell A3 contains "penny," the application (Microsoft Excel) will insert or type number one (1) in cells D2 and D3.

Cells(kcnt, 3).Value = 1

This eighth line of code instructs the application to end the If Statement.

End If

This ninth line of code instructs the application to take the number currently in the sub-container named kcnt and add a one (1) to it; currently kcnt has a one (1) in it, so 1 = 1 + 1; this means that kcnt will now have a 2 inside it.

kcnt = kcnt + 1

This tenth line of code instructs the application to conclude the While Statement.

Wend

This eleventh line of code instructs the application to find cell B1, and select it. This is done only after the loops statements have finished looping; that means that this is not to be done except one time, after the loops have concluded.

Range("B1").Select

This twelfth line of code instructs the application to type a COUNTIF formula in the selected cell B1; that formula will count the number of text strings that contains a double "nn" in cells A1: A3; so, if cell A2 had the name "Kenny" typed in it, and cell A3 had the word "penny" typed in it, the formula would result in the number two (2), because it found two (2) strings in the range A1:A3 that contained double "nn's." Therefore, cell B1 would have the number 2 entered into it by the application.

ActiveCell.FormulaR1C1 = "=COUNTIF(RC[-1]:R[2]C[-1],""*nn*"")"

This thirteenth line of code instructs the application to produce a message box that will display two (2) text strings surrounding a number that will be found in the sub-container named krow, i.e., if the sub-container named krow stores a 3, then the message box would display this: There are 3 rows of data.

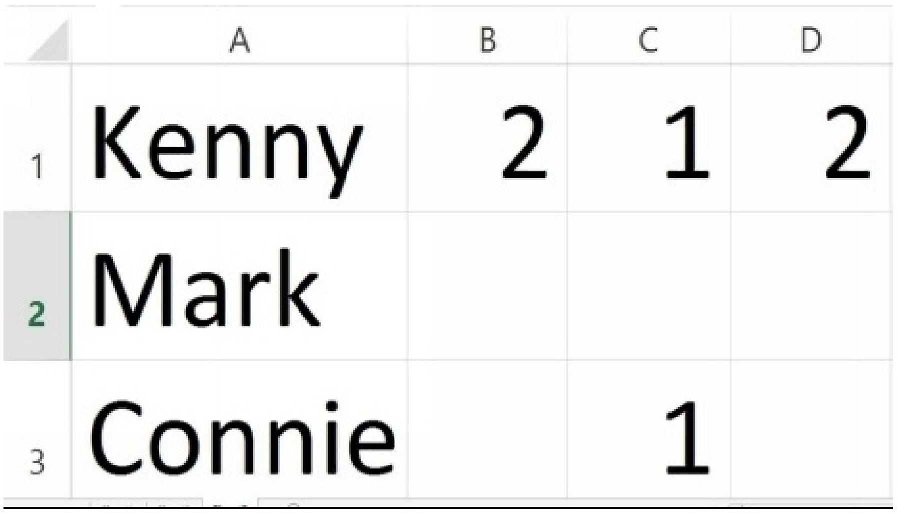
MsgBox "There are " & krow & " rows of data."

This fourteenth line of code instructs the application to sum the numeric characters found in the 3rd column or column C and put the result in cell D1. This formula will be explained in detail, so you understand it. To make it easier to understand, this is what the sum formula would look like after the application executed the functions: Sum(C1,last row address); let's assume that the very last row that had data in it was row 3, so the Sum formula would become Sum(C1,C3). Cells(1,3) is easy to interpret. It means row 1, column 3; this is cell C1. The comma that follows separates the starting cell for the range that needs to be summed, and what follows the comma will be the last cell in the range to be summed. Now, let's break it down. Cells(krow,3) means the 3rd column and whatever row is stored in the krow sub-container. This loop will keep increasing the number in the krow sub-container, until it reaches a cell in column 3 or column C that is an empty cell. Let us assume that C1:C3 has something in them, but C4 has nothing in it, so the loop will stop at C4. The application will store that stopping cell address (C4) in its memory; thus, you have cells(krow, 3).address, as the last cell that isn't empty. Now, the conclusion of this mega formula or function Address(false, False) simply means that both the Address row and column doesn't need to be the built-in address, e.g., D3, D6, F6, etc…. Instead, by indicating False, False, the application uses the Cells(krow = row 1, 3 = 3rd column or column C).Address(use number for row = 3, use number for column = 3); thus, we have Sum(Range(Cells(C1, C3))). This formula is a very complex formula to learn and understand for new users, so I'd encourage you to practice using it in different scenarios, before you can truly feel comfortable memorizing it.

```
Range("D1").Formula = "=SUM(" & Range(Cells(1, 3), Cells(krow,
3)).Address(False, False) & ")"
```

This fifteenth line of code instructs the application to stop allowing code instructions by the user, and it informs the application that this complete code is ready to be executed.

```
End Sub
```



	A	B	C	D
1	Kenny	2	1	2
2	Mark			
3	Connie		1	

Sample Object 1YA: This image shows the result of Code 10, after it has been ran

CODE 11: this code will change the background color of cell A1 to ten (10) different colors; there will be three (3) second intervals between each color change.

```
Sub WaitTest()
Dim kee As Integer
kee = 1
While kee < 10
If kee > 10 Then
End If
Cells(1, 1).Interior.ColorIndex = kee
Application.Wait (Now + TimeValue("0:00:03"))
kee = kee + 1
Wend
End Sub
```

This first line of code instructs the application to create a new main container named WaitTest().

```
Sub WaitTest()
```

This second line of code instructs the application to create a new sub-container named kee; this container will store a numeric character.

```
Dim kee As Integer
```

This third line of code instructs the application to fill the sub-container named kee with the number 1, as shown below:

```
kee = 1
```

This fourth line of code begins the While Statement. This line of code instructs the application to execute instructions, as long as the numeric value stored in the sub-container named kee doesn't exceed 10.

```
While kee < 10
```

This fifth line of code instructs the application to add a second criteria to executing instructions that will follow this line of code. The numeric value stored in the sub-container named kee must not be greater than 10; if it is greater than 10, the loop must cease or stop.

```
If kee > 10 Then
```

This sixth line of code instructs the application to end the If statement, and it also ends the loop statement, once the numeric value in the sub-container exceeds the number 10.

```
End If
```

This seventh line of code instructs the application to remain in the first row, first column, e.g., A1. Cell A1 will change colors nine (9) times, based on the number stored in the sub-container named kee. The container named kee will start as 1, and it will increase by 1, until it reaches the number 9. There will be a three (3) second delay of the background color changing in cell A1; the colors will change by comparing each number to the built-in color index chart.

```
Cells(1, 1).Interior.ColorIndex = kee
```

This eighth line of code instructs the application to delay the color change that will take place in cell A1 in three (3) second intervals. The delay or Wait function actually delays the entire system; this means that you won't be able to do anything until the delay or wait function concludes.

```
Application.Wait (Now + TimeValue("0:00:03"))
```

This ninth line of code instructs the application to pull the current numeric value stored in the container named kee and increase it by 1, e.g., 1 = 1 + 1; this means that (now) there's a number 2 in the container named kee.

```
kee = kee + 1
```

This tenth line of code instructs the application to conclude the While Statement.

```
Wend
```

This eleventh line of code instructs the application to conclude the starting Sub container named WaitTest().

```
End Sub
```

CODE 15: this code reverses the order of names found in Column A of this spreadsheet. Let's assume that we had the name "Xing, Li" in cell A1, and "Xi, Li M" in cell A2. This shows the last names first, followed by a comma and a blank space, then the first name and middle initial, if there is one. We don't want names in this order; instead, we want first names and middle initials, followed by a blank space, then followed by the last names, e.g., Li Xing, Li M Xi. Before you run this code, you'll need to type two names that start last name first followed by first name and middle initials, as shown in ***Sample***

Object 1YB.

```
Sub LeftMidLen()
Dim kinteger As Integer
kinteger = 2
While Cells(kinteger, 1) <> ””
Dim kstring As String
Dim kspace As String
kstring = Cells(kinteger, 1).Value
kspace = ” “
Dim findit As Integer
findit = InStr(kstring, kspace)
MsgBox Mid(kstring, findit + 1, Len(kstring)) & ” ” & Left(kstring, findit - 2)
Cells(kinteger, 4).Value = Mid(kstring, findit + 1, Len(kstring)) & ” ” & Left(kstring,
findit - 2)
kinteger = kinteger + 1
Wend
End Sub
```

This first line of code instructs the application to create a main container that will store all of your smaller sub-containers and commands to the application (Microsoft Excel); the name of this new container will be LeftMidLen(), as shown below:

```
Sub LeftMidLen()
```

This second line of code instructs the application to create a sub-container to store a number; the container will be named kinteger.

```
Dim kinteger As Integer
```

This third line of code instructs the application to fill the container named kinteger with the number 2, as shown below:

```
kinteger = 2
```

This fourth line of code instructs the application to begin the While Statement; this statement will only start and continue as long as the rows in column one (1) or column A has something typed in them, starting with cell A2; as stated, the loop will start at cell A2 and move downwards, e.g., A3, A4, etc… The loop will stop when it reaches an empty cell in column A; in this case, there's nothing typed in cell A4, so the loop will stop there.

While Cells(kinteger, 1) <> ""

This fifth line of code instructs the application to create a sub-container and name it kstring; it will store a text string, as shown below:

Dim kstring As String

This sixth line of code instructs the application to create a sub-container and name it kspace; it will store a text string, as shown below:

Dim kspace As String

This seventh line of code instructs the application to fill the sub-container named kstring with whatever is typed into the rows in column 1 or column A. The rows will change, but the column will be absolute, i.e., the column will not change. The first cell to be impacted will be cell A2, so whatever is in cell A2 will become stored inside the sub-container named kstring.

kstring = Cells(kinteger, 1).Value

This eighth line of code instructs the application to fill the sub-container named kspace with an empty space; this is what the empty set of double quotation marks with a space between them means.

kspace = " "

This ninth line of code instructs the application to create a sub-container named findit, and it will store a number, e.g., 1, 2, 3, etc….

Dim findit As Integer

This tenth line of code instructs the application to fill the sub-container named findit with the result of a formula. The Instr function works just like the FIND formula, but the only difference is the requirements are reversed. In the Find formula, it first requires the user to indicate what it wants to find, and the second requirement requires the user to indicate the cell location of the text string that has the value that needs to be found, e.g., =Find(" ","Kenny Keys"), =Find(" ",A2), etc…. In the case of the Instr function, the requirements are reversed: first the user indicates the text string or cell location that will be searched, e.g., Instr("Kenny Keys"," "), or Instr(A2," ")…. The Instr example that I've provided just gives you and ideal of what I'm talking about; however, it is not how you'd write the formula. You'd usually assign a name to the spreadsheet cell location that stores your text string to be searched, and you'd do the same for the string to find in the main text string, as shown below. Kstring will pull from cell A2, and kspace will be an empty space within the main text string to be found.

findit = InStr(kstring, kspace)

This eleventh line of code instructs the application to use very common Microsoft Excel formulas to extract data. In this case, the application will produce a message box, and it will combine text strings that are stored in containers with a text string that isn't stored in a container. The MID formula instructs the application to pull whatever is stored in the container named kstring, so let us assume that kstring contains Keys, Kenny L.; this is how the Mid() formula will be interpreted by the application: Mid("Keys, Kenny L", " ", Len("Keys, Kenny L")). Next, the application will convert each character in between the quotation marks, for the second and third criteria, into a numeric character, based on the length of the text string; spaces count as characters too, so to break it down, we do this: my full name with the two spaces and the comma is thirteen (13) characters long. The first letter of my first name is located seven (7) characters from the left: K = 1, e=2, y=3, s=4, comma (,)=5, space (" ")=6, K=7, e=8, n=9, n=10, y=11, space (" ")=12, L=13. The second requirement in the Mid() formula demands to know what location, in the text string shown in the first requirement, is the very first space; the first space is the 6th character, and the second space is the 12 character, starting from the leftmost character in the text string.

Now that we know the numeric location of each character in the text string that spells out my full name, we want to reverse my name order; in other words, I want my first name and middle initial to precede my last name, e.g., Kenny L Keys. This is why we use the Mid() formula first. Now, I'll explain how the formula executes. The first requirement remains a text string; the second requirement converts to a numeric character, and the third requirement converts to a numeric character, e.g., Mid("Keys, Kenny L", " " + 1, Len("Keys, Kenny L")), becomes Mid("Keys, Kenny L ",6 +1,13); this becomes Mid("Keys, Kenny L ",7,13), so the seventh (7th) character in my name would be the "K" in my first name, in the text string. There are thirteen characters in my full name, so the application will start counting thirteen (13) characters leftwords, until it completes thirteen (13) characters or it gets to the last character in the text string. The application will see that it doesn't need all thirteen characters to reach the last character in the text string, so it will cancel counting, after it reaches the "L" in my Middle name, because that is the last character in the text string of the first (1st) requirement for this Mid() formula; the final result of the Mid() formula will be **Kenny L**. Now, we need to put an empty space between my middle initial and my last name, so we add the following to the end of the mid formula: **& " " &**. Lastly, we connect my first name on by extracting it from the main text strings **Keys, Kenny L** using the Left() formula. The Left() formula only has two (2) requirements: the first requirement is the text string to extract from; this would be "Keys, Kenny L"; the second requirement is a numeric character that indicates how many characters rightwards to count, in order to get to the first (1^{st}) space; once we get to the space, we subtract two (2) character from it, e.g., the comma (,), because we don't want to see the comma at the end of my name, once it is reversed; the other will be the empty space (" "), because we won't need an empty space at the end of my full name, after it is reversed. So, to get to the first space is six (6) characters. We need to minus two (2) characters to remove the comma (,) and the space (" "), so we are left with 4; thus, the formula converted goes from this: Left("Keys, Kenny L"," ") to this: Left("Keys, Kenny L", 4). The end result of the Left() formula will be **Keys**. When the entire mega formula is converted and connected (also known as concatenated), it will look like this: **Kenny L Keys**. So, the message box will produce the following message: **Kenny L Keys**.

MsgBox Mid(kstring, findit + 1, Len(kstring)) & " " & Left(kstring, findit - 2)

This twelfth line of code will not need to be explained, because it is identical to the eleventh line of code, above; however, instead of a message box displaying the end result of the combined formulas, it will be displayed in column 2 or column B for each row that corresponds to column A.

Cells(kinteger, 2).Value = Mid(kstring, findit + 1, Len(kstring)) & " " & Left(kstring, findit - 2)

This thirteenth line of code instructs the application to increase the number stored in the sub-container named kinteger by 1, so 1 = 1 + 1 will equal 2; thus, the kinteger container will now store the number 2.

kinteger = kinteger + 1

This fourteenth line of code instructs the While Statement to conclude.

Wend

This fifteenth line of code instructs the application to conclude the starting **Sub** code named **LeftMidLen()**; this means it's ready to be executed by the application.

End Sub

Note: these are Microsoft Excel spreadsheet formulas that will end in the same results as this VBA code:

Type this formula in cell B2 and drag it down to change Last Name, First Name and Middle Initials (e.g., Keys, Kenny L) to First Name Middle Initial and Last name order (e.g., Kenny L Keys):

This is the complete formula that converts the order of the full name:

=IF(ISERROR(FIND(" ",A2,FIND(" ",A2)+1)),MID(A2,FIND(" ",A2)+1,LEN(A2)),MID(A2,FIND(" ",A2)+1,FIND(" ",A2,FIND(" ",A2)+1)-FIND(" ",A2)))&" "&LEFT(A2,FIND(" ",A2)-2)

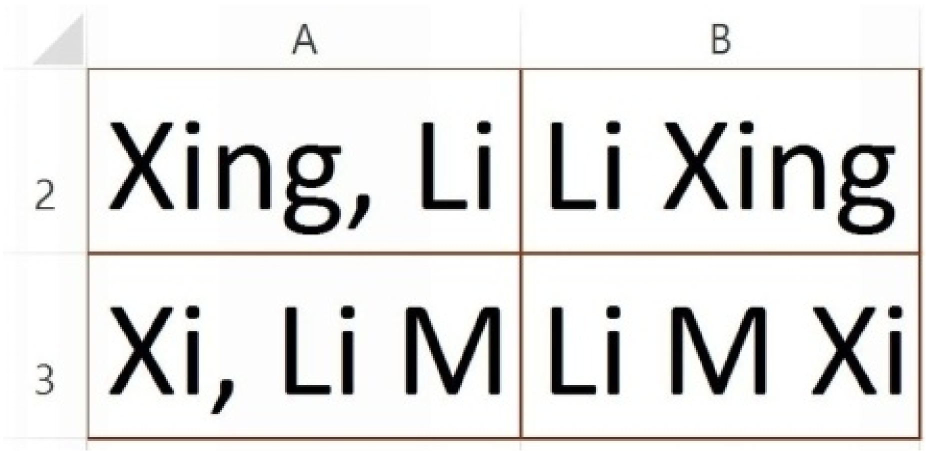
This is the portion that handle extracting the First Name:

=IF(ISERROR(FIND(" ",A2,FIND(" ",A2)+1)),MID(A2,FIND(" ",A2)+1,LEN(A2)),MID(A2,FIND(" ",A2)+1,FIND(" ",A2,FIND(" ",A2)+1)-FIND(" ",A2)))

This is the portion that handle extracting the Last Name:

=LEFT(A2,FIND(" ",A2)-2)

You can learn how to read this mega formula by visiting ***Chapter 5***.



	A	B
2	Xing, Li	Li Xing
3	Xi, Li M	Li M Xi

Sample Object 15: This image shows the result of Code 15, after it has been ran.

For Next Loop Statement: Chapter 2B

This code produces a message box to show numbers 1 – 4 and 6-8.

1st For Next If End Statement

CODE 16: this code produces a message box that counts from 1 to 4; it will skip 5, and it will resume counting from 6-8.

```
Sub ken_FORNEXT()
Dim kennum As Integer
kennum = 1
For kennum = 1 To 4
If kennum > 8 Then
End If
MsgBox kennum
Next
kennum = kennum + 1
For kennum = 6 To 8
If kennum > 8 Then
End If
MsgBox kennum
Next
kennum = kennum + 1
End Sub
```

This first line of code instructs the application to create a new container to hold all sub-containers and commands; the new container is to be named FORNEXT(), as shown below:

```
Sub ken_FORNEXT()
```

This second line of code instructs the application to create a new sub-container and name it kennum; this container will store a number.

Dim kennum As Integer

This third line of code instructs the application to fill the container named kennum with the number 1.

kennum = 1

This fourth line of code begins the For Statement; it demands that the application only allows the container named kennum to store numbers 1 to 4; any number greater or less will not result in the execution of this loop statement.

For kennum = 1 To 4

This fifth line of code adds an additional requirement to the For Statement. This If Statement demands that the application cease or end the loop, if the container named kennum has a number greater than 8, e.g., 9, 10, 11, etc…

If kennum > 8 Then

This sixth line of code concludes the fifth line of code. If the number in the container named kennum is greater than 8, it will end the If Statement, and nothing will be done. The loop will cease to continue.

End If

This seventh line of code instructs the application to produce a message box that will show the number in the container named kennum each time the loop increases its number, e.g., 1, 2, 3, etc….

MsgBox kennum

This eighth line of code instructs the application to move to the next line of code, i.e., move to the ninth line of code.

Next

This ninth line of code instructs the application to increase the number in the container named kennum, e.g., 1 = 1 + 1; this means that now the container named kennum contains a number 2.

kennum = kennum + 1

This tenth line of code is another For Statement that tells the loop to resume when the container reaches the number 6 and loop until it reaches number 8.

For kennum = 6 To 8

This eleventh line of code demands that the container named kennum not store a number greater than 8, e.g., 9, 10, etc…; if it does, the loop will cease.

If kennum > 8 Then

This twelfth line of code instructs the application to cease the loop, if the number stored in the container named kennum is greater than 8.

End If

This thirteenth line of code instructs the application to produce a message box and display the current number stored in the container named kennum.

MsgBox kennum

This fourteenth line of code instructs the application to move to the next line of code, i.e., the fifteenth line of code.

Next

This fifteenth line of code instructs the application to increase the number stored in the container named kennum, e.g., 1 = 1 + 1; this means that now the container named kennum stores the number 2.

kennum = kennum + 1

This end or concludes the full code of instructions stored in the main container named FORNEXT().

End Sub

Do While Loop Statement: Chapter 2C

CODE 18: this loop produces a message box that counts from 1 to 10.

```
Sub kenDowhile()
Dim keddy As Integer
keddy = 1
Do
MsgBox keddy
keddy = keddy + 1
Loop While keddy < 10
End Sub
```

This first line of code instructs the application to create a new main container to store all sub-containers and commands; the new container will be named kenDowhile(), as shown below:

```
Sub kenDowhile()
```

This second line of code instructs the application to create a new sub-container that will store a number, and it will be named keddy, as shown below:

```
Dim keddy As Integer
```

This third line of code instructs the application to fill the new container named keddy with the number 1.

```
keddy = 1
```

This fourth line of code is the beginning of the Do Statement. Nothing else goes on the line with the Do, as shown below:

```
Do
```

This fifth line of code instructs the application to produce a message box and display the current number in the sub-container named keddy.

MsgBox keddy

This sixth line of code instructs the application to retrieve the current number stored in the sub-container named keddy and increase it by 1, e.g., 1 = 1 + 1; this means that the new number in the container named keddy is now 2.

keddy = keddy + 1

This seventh line of code instructs the application to continue to loop, as long as the number in the sub-container named keddy is less than the number 10.

Loop While keddy < 10

This eight line of code instructs the application to conclude or close the main container named kenDowhile(); this means that the complete code it ready to be executed.

End Sub

Do Until Loop Statement: Chapter 2D

CODE 19: this code will require the user to turn the volume on his/her computer up to high, before running the code. This code activates the built-in speech function to speak the numbers from 1 through 10 aloud; after speaking the number, a message box will display the number.

```
Sub kendountil()
Dim sib As Integer
sib = 1
Do
Application.Speech.Speak (sib)
MsgBox sib
sib = sib + 1
Loop Until sib > 10
End Sub
```

This first line of code instructs the application to create a new container to store all sub-containers and commands to be executed; this container is to be named kendountil(), as shown below:

```
Sub kendountil()
```

This second line of code instructs the application to create a new sub-container to store a number; it is to be named sib, as shown below:

```
Dim sib As Integer
```

This third line of code instructs the application to put a number 1 inside the container named sib, as shown below:

```
sib = 1
```

This fourth line of code instructs the application to begin the Do Statement; nothing else goes on the line with the word Do, as shown below:

Do

This fifth line of code instructs the application (Microsoft Excel) to access the built-in speech folder; once the folder is accessed, the application must then access a sub-folder within the **speech** folder; that sub-folder is called **speak**. This path accesses the application's built-in reader that will read aloud what is typed into the Microsoft Excel application. The next step for commanding the application to speak or read aloud is to put the container name or text string in between an opening and closing parenthesis, e.g., Application.Speech.Speak("Kenny L Keys"), Application.Speech.Speak(sib), Application.Speech.Speak (Range("A1")), Application.Speech.Speak (Cells(1, 1)), etc…; in this case, we type the container named **sib** inside parentheses; this instructs the application to read aloud anything found inside the sib container, i.e., if the **sib** sub-container has a number one (1) inside it, the built-in voice of the application will speak "one (1)" aloud.

Application.Speech.Speak (sib)

This sixth line of code instructs the application to produce a message box and to display whatever is stored inside the sub-container named **sib**; in this case, it will display the number **1**.

MsgBox sib

This eighth line of code instructs the application to increase the number inside the sub-container by 1, e.g., 1 = 1 + 1; this means that the new number inside the sub-container named sib is 2.

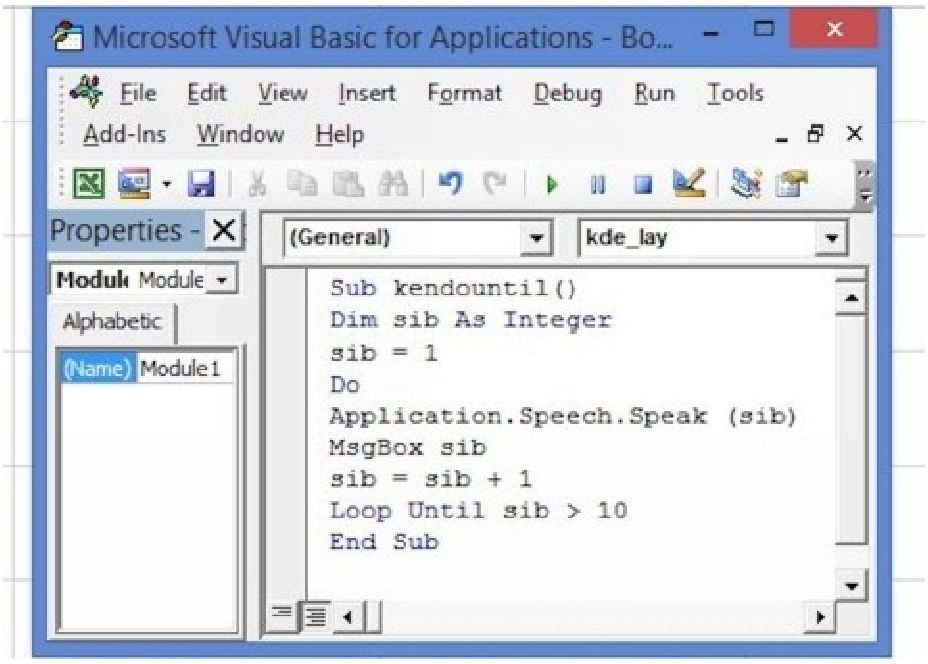
sib = sib + 1

This ninth line of code instructs the application to continue to loop until the number stored in the sub-container named sib reaches 11. When he number becomes 11, all looping will cease or stop.

Loop Until sib > 10

This tenth line of code instructs the application to conclude the main folder named kendountil(); this also instructs the application that the full code is now complete, and it is ready to be executed.

End Sub


Microsoft Visual Basic for Applications - Bo...
File Edit View Insert Format Debug Run Tools
Add-Ins Window Help
Properties -
Module Module
Alphabetic
(Name) Module1
(General)
kde_lay

```
Sub kendountil()
Dim sib As Integer
sib = 1
Do
Application.Speech.Speak (sib)
MsgBox sib
sib = sib + 1
Loop Until sib > 10
End Sub
```

Sample Object 19: this shows how CODE 19 looks typed inside the VBA Module window.

For Each In Loop Statement: Chapter 2E

CODE 20: this code is identical to CODE 19; the only difference is beside speaking a number aloud, it speak aloud all the sheet names within the active workbook, etc., sheet 1, sheet 2, kensheet, etc….

```
Sub KenForEachIn()
For Each sht In Application.Sheets
Application.Speech.Speak (sht.Name)
MsgBox sht.Name
Next sht
End Sub
```

This first line of code instructs the application to create a new main container to hold all sub-containers and commands; it is to be named KenForEachIn(), as shown below:

```
Sub KenForEachIn()
```

The second line of code instructs the application to search the entire, current, active workbook for all the sheets in the workbook; once it determines how many sheets are in the workbook, it also quickly records the names of all the sheets in the workbook. Now that it's done these two things, the line of code below demands that this information be stored inside a folder named **sht** until the third line of code makes the next request.

```
For Each sht In Application.Sheets
```

This third line of code instructs the application to use the information about the names of each spreadsheet that exists in the current, active workbook; next, it instructs the application (Microsoft Excel) to access the voice feature; this is done by accessing the **speech** subfolder located in the parent **application** folder. Next, the applications is instructed to look inside the **speech** subfolder and find another subfolder inside of it; the subfolder will be named **speak**; once this subfolder is found, the application demands that the speech function speaks the sheet names in this workbook aloud for all to hear. You must be sure to turn your computer's speaker volume up, so you can hear. **Sht** is a new container name that stores the path **Application.Sheets**; this is similar to the ***Set*** function that we discuss earlier in ***Chapter 2***…. Since the **sht** container basically shortens the full path **Application.Sheets**, we treat it as the full path; we do this by continuing to enter the next subfolder within the **Sheets** built-in subfolder; the next subfolder will is **Name**. We put **sht.Name** inside the parentheses; it's the same as putting the full path **Application.Sheets.Name** inside the parentheses.

Application.Speech.Speak (sht.Name)

This fourth line of code instructs your application to produce a message box; that message box must display each sheet name, after each is spoken aloud by the application.

MsgBox sht.Name

This fifth line of code instructs your application to proceed to the next sheet, so its name can be spoken aloud and its name can be displayed in a message box.

Next sht

This sixth line of code instructs your application to conclude all instructions or demands for the application; this is also a sign that the code is ready to be executed.

End Sub

CODE 21: this code produces a message box that displays the content and address info found in the range A1:A3, e.g. Kenny Keys $A$1, etc…. Before running this code, you must type something in cells A1:A3, and then run the code.

```
Sub ForEachIn2()
Dim krange As String
For Each kennyfind In ActiveSheet.Range("A1:A3")
krange = krange & vbNewLine & kennyfind.Value & " " & kennyfind.Address
Next kennyfind
MsgBox krange
End Sub
```

This first line of code instructs the application to create a main storage container to store all sub-containers and all command and instructions to be executed; the container will be named ForEachIn2(), as shown below:

Sub ForEachIn2()

This second line of code instructs the application to create a new sub-container to store a text string; it is to be named kranges, as shown below:

Dim krange As String

The third line of code instructs the application to start the For Each Statement; in this case, it will create a new container named **kennyfind** and fill it with this built-in folder path: **ActiveSheet.Range("A1:A3")**. This means that the application now pulls all the data from the spreadsheet range **A1:A3**, and it is stored inside the new container named **kennyfind**.

For Each kennyfind In ActiveSheet.Range("A1:A3")

This fourth line of code instructs the application to create a new sub-container and name it **krange**; once this is done, the application is to convert all the data currently located in the container named **kennyfind** into text strings; once that is done, the application is instructed to change the contents inside the **krange** folder into the ***value (e.g., Kenny, okflksd, Moki, etc…)*** and cell ***location (e.g., $A$1, $G$45, etc…)*** for each cell's data content.

krange = krange & vbNewLine & kennyfind.Value & " " & kennyfind.Address

This fifth line of code instructs the application to move to the next cell location within the range A1:A3.

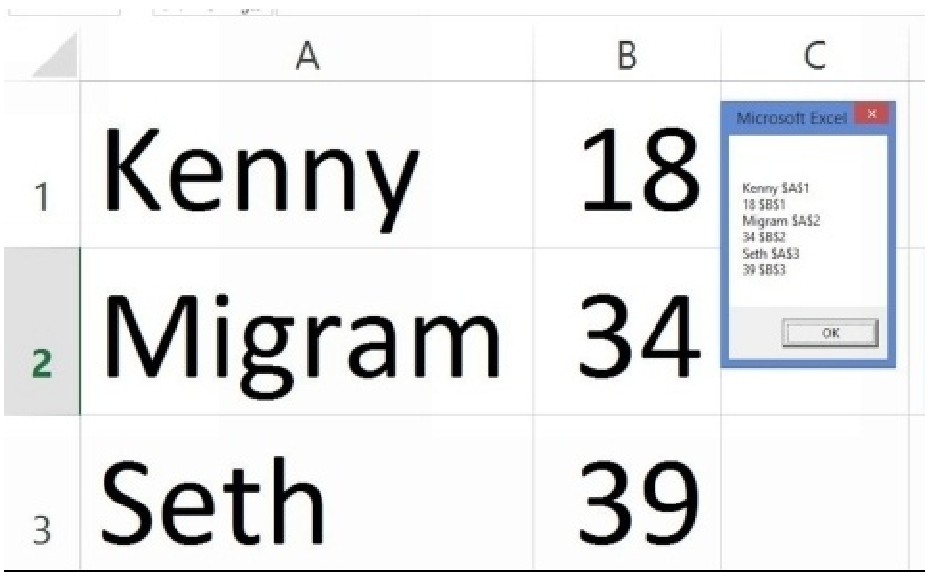
Next kennyfind

This sixth line of code instructs the application to produce a message box and display each cell's content within the range A1:A3 along with its cell location in the spreadsheet.

MsgBox krange

This seventh line of code instructs the application to conclude the main storage container; this means it's time to execute the complete code.

End Sub



	A	B	C
1	Kenny	18	
2	Migram	34	
3	Seth	39	

```
Microsoft Excel

Kenny $A$1
18 $B$1
Migram $A$2
34 $B$2
Seth $A$3
39 $B$3
```

Sample Object 21: this shows the result of CODE 21, after using the range A1:B3, but you will only use the range A1:A3, before running the code; therefore, your results will be slightly different.

CODE 22: this code is identical to ***CODE 21***, but this code includes the **SPEECH** function; it will read aloud the contents in each cell in the range **A1:B3** that contains double "**nn**" characters, e.g., Kenny, Minny, henney, Connard, etc.… Before running this code, you must first type something (anything) in cell range **A1:B3**, but you must make sure to include the double **nn**'s in at least 2 or three cells, within the range, e.g., A1 = Kenny, A2 = John, A3 = Mary, B1 = 45, B2 = 23nn, B3 = 477.

```
Sub ForEachIghn3()
Dim krange1 As String
For Each stacy In ActiveSheet.Range("A1:b3")
If InStr(stacy.Value, "nn") > 0 Then
stacy.Interior.ColorIndex = 1
stacy.Font.ColorIndex = 2
Application.Speech.Speak (stacy.Value)
MsgBox stacy.Value
End If
Next stacy
End Sub
```

This first line of code instructs the application to create a new main storage container; it is to be named ForEachIghn3(), as shown below:

```
Sub ForEachIghn3()
```

This second line of code instructs your application to create a sub-container and name it krange; this container will store a text string.

Dim krange1 As String

This third line of code instructs the application to create a folder named **stacy** and store the built-in file path **ActiveSheet.Range("A1:B3")** inside it.

For Each stacy In ActiveSheet.Range("A1:B3")

This fourth line of code begins the If Statement; this instructs the application to search all cell content, within the range **A1:B3**; this range is stored in the container named **stacy**; once it finds a cell that has a double **nn** text string inside it, it will executed the instructions found in codes on lines five, six, seven and eight.

If InStr(stacy.Value, "nn") > 0 Then

This fifth line of code instruct the application to turn the cell background color to match number 1 on the built-in color index pallet, for any cell that contains a double **nn** in the cell's text string, e.g., Kenny, Penny, Donny, etc…

stacy.Interior.ColorIndex = 1

This sixth line of code instruct the application to turn the font color to match number 2 on the built-in color index pallet, for any cell that contains a double **nn** in the cell's text string, e.g., Kenny, Penny, Donny, etc…

stacy.Font.ColorIndex = 2

This seventh line of code instructs the application to **speak** the text string aloud for any cell that contains a double **nn** in the cell's text string, e.g., Kenny, Penny, Donny, etc…

Application.Speech.Speak (stacy.Value)

This eighth line of code instructs the application to produce a message box and display the text string for any cell that contains a double **nn** in the cell's text string, e.g., Kenny, Penny, Donny, etc…

MsgBox stacy.Value

This ninth line of code instructs the application to conclude or end the If Statement.

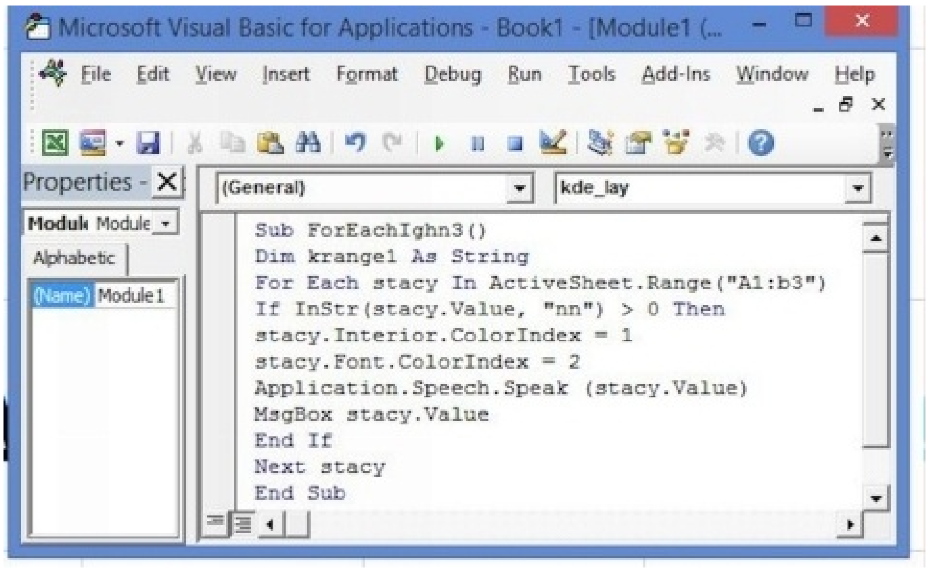
End If

This tenth line of code instructs the application to move to the next cell within the range A1:B3 (e.g, A1, B1, A2, B2, A3, B3), until all cells have been searched for strings containing a double **nn** combination.

Next stacy

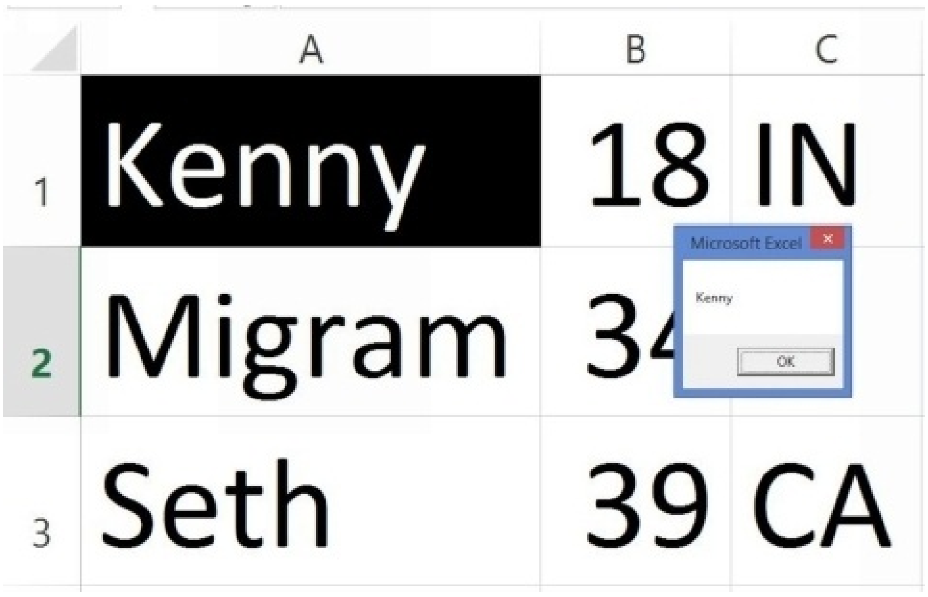
This eleventh line of code concludes the full code stored in the container named ForEachIghn3(); this means that the application is ready to execute the complete code.

End Sub



```
Sub ForEachIghn3()
Dim krange1 As String
For Each stacy In ActiveSheet.Range("A1:b3")
If InStr(stacy.Value, "nn") > 0 Then
stacy.Interior.ColorIndex = 1
stacy.Font.ColorIndex = 2
Application.Speech.Speak (stacy.Value)
MsgBox stacy.Value
End If
Next stacy
End Sub
```

Sample Object 22A: this shows how CODE 22 looks, when it's correctly typed inside the VBA Module window.



	A	B	C
1	Kenny	18	IN
2	Migram	34	
3	Seth	39	CA

Sample Object 22B: this shows the final result of CODE 22, after the code has been ran.

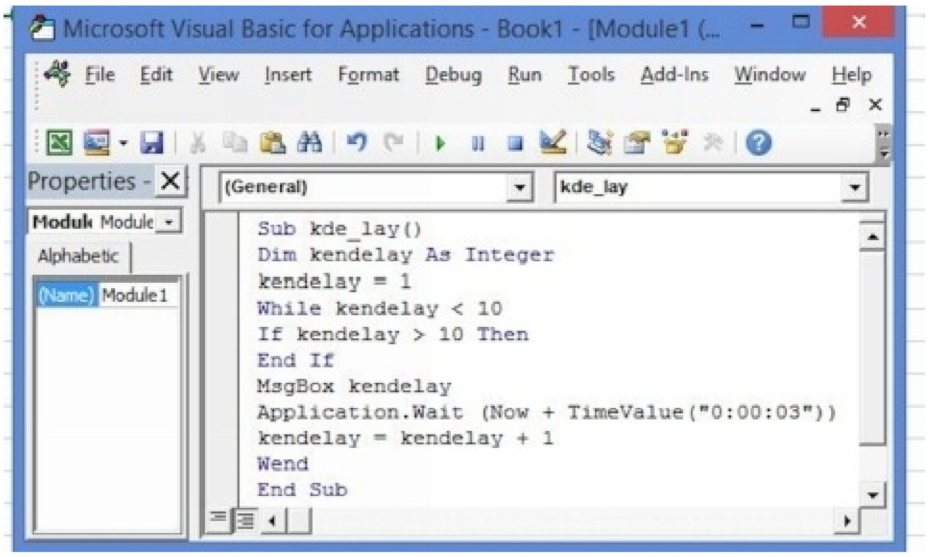
Exercises: Chapter 3

EXERCISE 1: the objective is to produce message boxes that counts from 1 to 9 in three (3) second intervals.

I need you to type this code into your VBA Module window and run it; after you've ran the code, I need you to change the sub-container named **kendelay** to **uui** everywhere in this code for Exercise 1. Be sure to change them all, or the code won't work the next time you run it.

After you've successfully ran the code, after changing **kendelay** to **uui**, I want you to change **While uui < 10** to **While uui < 3**, and I want you to change **If uui > 10** to **If uui > 2**; after this is done, I want you to run the code.

```
Sub kde_lay()
Dim kendelay As Integer
kendelay = 1
While kendelay < 10
If kendelay > 10 Then
End If
MsgBox kendelay
Application.Wait (Now + TimeValue(“0:00:03”))
kendelay = kendelay + 1
Wend
End Sub
```


Microsoft Visual Basic for Applications - Book1 - [Module1 (...
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
Properties -
Module Module
Alphabetic
(Name) Module1
(General)
kde_lay

```
Sub kde_lay()
Dim kendelay As Integer
kendelay = 1
While kendelay < 10
If kendelay > 10 Then
End If
MsgBox kendelay
Application.Wait (Now + TimeValue("0:00:03"))
kendelay = kendelay + 1
Wend
End Sub
```

OBJECT EXERCISE 1: this shows what the code looks like, if it has been correctly typed inside the VBA Module window.

EXERCISE 2: the objective is to delay the execution of nine (9) message boxes that will display numbers 1 through 9; there will be a one (1) second interval between each message box.

I want you to, first, type the code as it appears below and run it. After you've successfully ran the code, I want you to make the following changes to the code: change the seventh line of code from **MsgBox kendelay** to **Range("B3").Value = kendelay**; after you've made the changes, I want you to run the code again; if you did everything right, the spreadsheet's cell B3 will produce a different number every three (3) seconds.

Next, I want you to add a line of code directly under the seventh line of code that you just changed. The new line of code should be typed as follows:

cells(kendelay, 3).Interior.ColorIndex = kendelay

Sub kde_lay()

Dim kendelay As Integer

kendelay = 1

While kendelay < 10

If kendelay > 10 Then

End If

MsgBox kendelay

Application.Wait (Now + TimeValue("0:00:03"))

kendelay = kendelay + 1

Wend

End Sub

EXERCISE 3: the objective is to delay the background color change in cells located in column A, starting with cell A2. The delay between each cell's color change will be every three (3) seconds. Before running this code, you must type a unique number in cells A1:A5; the number must be between 1 and 56, e.g., 1, 45, 3, 16, etc.… After you've successfully ran this code, I want you to go back and change the numbers in cells A1:A5 to different numbers from 1 to 56 and run the code again. After you've successfully ran the changed code, I want you to change the **0:00:01** to **0:00:04** and run the code again; this will change the intervals of delay from 1 second to 4 seconds per color change.

Sub WaitTest()

Dim kee As Integer

kee = 1

While Cells(kee, 1) <> ""

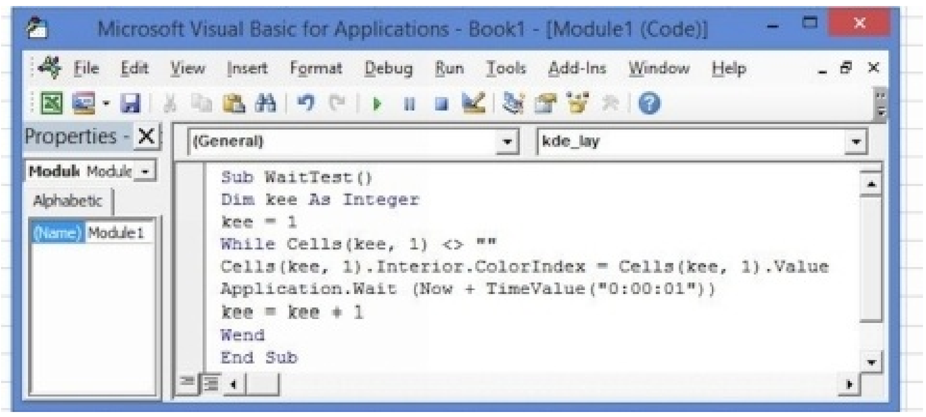
Cells(kee, 1).Interior.ColorIndex = Cells(kee, 1).Value

Application.Wait (Now + TimeValue("0:00:01"))

kee = kee + 1

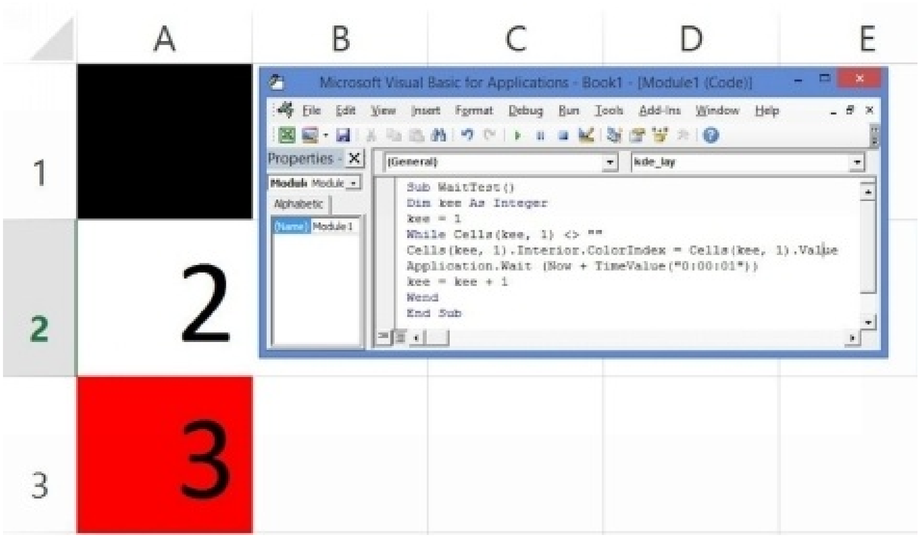
Wend

End Sub


Microsoft Visual Basic for Applications - Book1 - [Module1 (Code)]
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
Properties
Alphabetic
(Name) Module1
(General)
kde_lay

```
Sub WaitTest()
Dim kee As Integer
kee = 1
While Cells(kee, 1) <> ""
Cells(kee, 1).Interior.ColorIndex = Cells(kee, 1).Value
Application.Wait (Now + TimeValue("0:00:01"))
kee = kee + 1
Wend
End Sub
```

OBJECT EXERCISE 3: this shows what a correctly typed (original) code looks like in the VBA Module window, if it has been typed correctly.


A
B
C
D
E
1
2
3
2
3
Microsoft Visual Basic for Applications - Book1 - [Module1 (Code)]
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
Properties
Module Module
Alphabetic
(Name) Module1
(General)
kde_lay

```
Sub WaitTest()
Dim kee As Integer
kee = 1
While Cells(kee, 1) <> ""
Cells(kee, 1).Interior.ColorIndex = Cells(kee, 1).Value
Application.Wait (Now + TimeValue("0:00:01"))
kee = kee + 1
Wend
End Sub
```

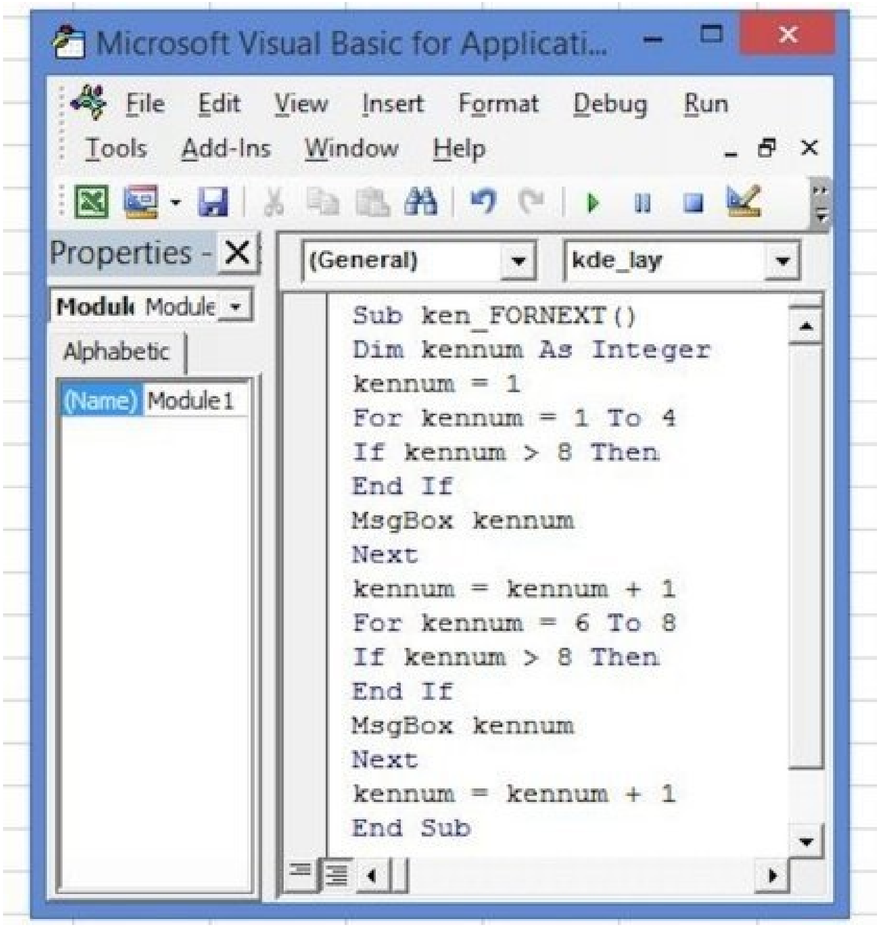
OBJECT EXERCISE 3A: this shows the result of a correctly typed code, after running it.

EXERCISE 4: the objective is to produce seven (7) message boxes that will count from 1 to 4 and 6 to 8; the number 5 will not be displayed in a message box.

After you've successfully ran the code, I want you to change the sub-container named **kennum** to **OUCH**. Be sure to change it everywhere that **kennum** appears in this code; after you've made the changes, I want you to run the code again.

```
Sub ken_FORNEXT()
Dim kennum As Integer
kennum = 1
For kennum = 1 To 4
If kennum > 8 Then
End If
MsgBox kennum
Next
kennum = kennum + 1
For kennum = 6 To 8
If kennum > 8 Then
End If
MsgBox kennum
Next
kennum = kennum + 1
```

End Sub

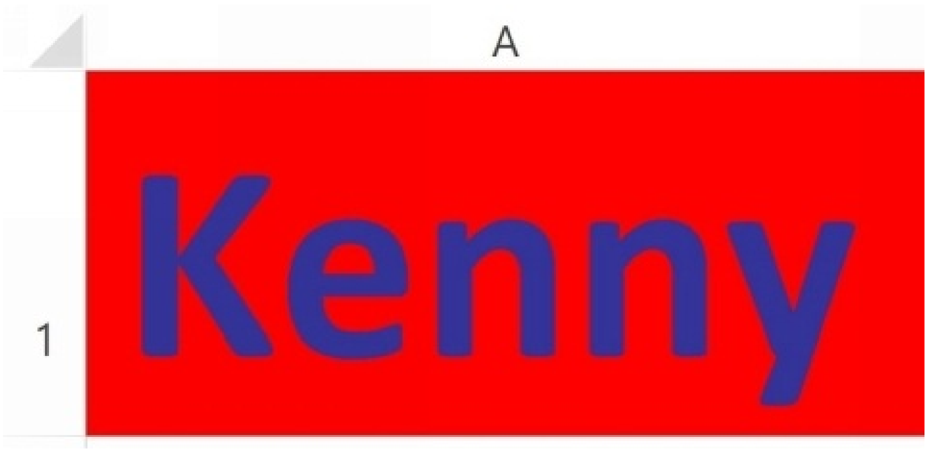

Microsoft Visual Basic for Applicati...
File Edit View Insert Format Debug Run
Tools Add-Ins Window Help
Properties
Module Module
Alphabetic
(Name) Module1
(General)
kde_lay

```
Sub ken_FORNEXT()
Dim kennum As Integer
kennum = 1
For kennum = 1 To 4
If kennum > 8 Then
End If
MsgBox kennum
Next
kennum = kennum + 1
For kennum = 6 To 8
If kennum > 8 Then
End If
MsgBox kennum
Next
kennum = kennum + 1
End Sub
```

OBJECT EXERCISE 4: this shows the code correctly typed in the VBA Module window.

EXERCISE 5: this code is indented to give the user a feel for formatting cell A1's font, cell color background and borders. Please be sure to type each line without breaking.

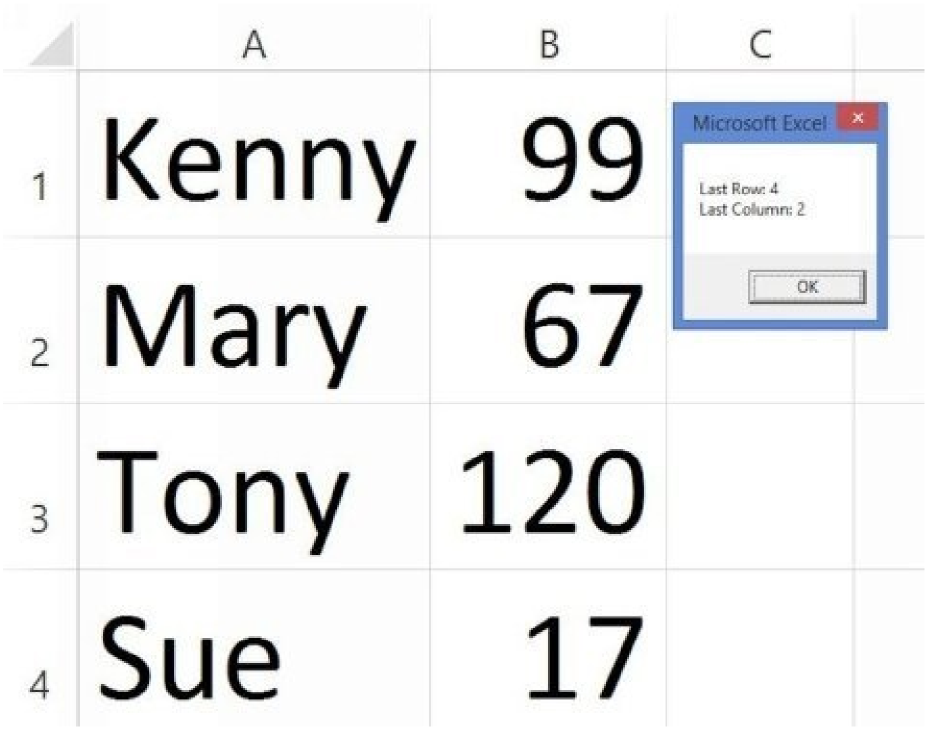
```
Sub gbds()
Dim KennyColor As Integer, KennyFontSize As Integer, _
KennyBoldText As Variant, Kennycellshading As Integer, KennyString As String
KennyColor = InputBox("Type numberic color for font, e.g., 1, 3, 33, etc…", "")
KennyFontSize = InputBox("Type font size, e.g., 12, 22, 33, etc…", "")
KennyBoldText = InputBox("Bold text or no? Type TRUE or FALSE", "")
Kennycellshading = InputBox("Type cell shading number, e.g., 1,3,5,etc…", "")
KennyString = InputBox("Type something", "")
Range("A1").Font.ColorIndex = KennyColor
Range("A1").Font.Size = KennyFontSize
Range("A1").Font.Bold = KennyBoldText
Range("A1").Interior.ColorIndex = Kennycellshading
Range("A1").Value = KennyString
Range("A1").Columns.AutoFit
End Sub
```



OBJECT EXERCISE 5: this shows the result of the code, if it was correctly typed into the VBA Module window, after running the code and doing as instructed by the input boxes.

EXERCISE 6: the purpose of this code is to determine the last column with data typed in it and the last row with data typed in it; both will be displayed in one message box. Before running this code, please typed something in each cell in the range A1:B2 and then run the code.

```
Sub Range_End_Method()
Dim lRow As Long
Dim lCol As Long
lRow = Cells(Rows.Count, 1).End(xlUp).Row
lCol = Cells(1, Columns.Count).End(xlToLeft).Column
MsgBox "Last Row: " & lRow & vbNewLine & "Last Column: " & lCol
End Sub
```



	A	B
1	Kenny	99
2	Mary	67
3	Tony	120
4	Sue	17

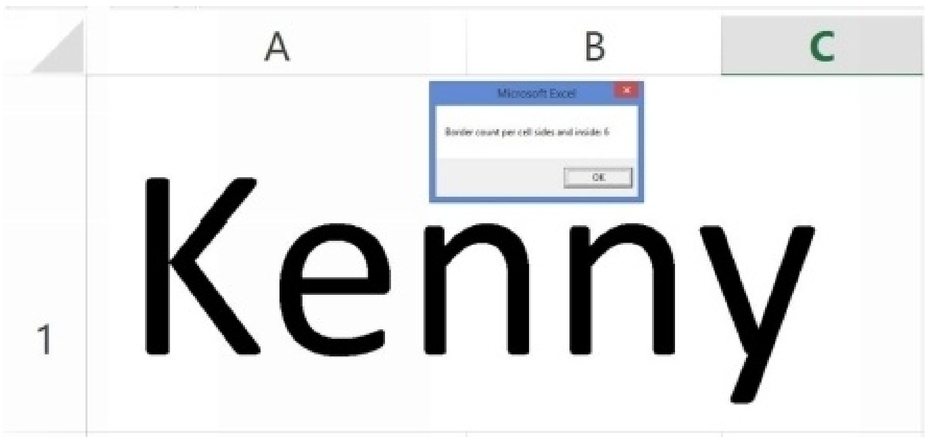
```
Last Row: 4
Last Column: 2
```

OBJECT EXERCISE 6: this displays the result of the code for Exercise 6, if it was correctly typed inside the VBA Module window.

EXERCISE 7: the objective of this code is to encourage users to experiment (on his or her own) with trying different built-in pathways that are responsible for impacting different things, e.g., font, borders, images, text boxes, etc… Each line begins with either Sub, Dim, kfill, kfont, kbold, kital, kfontsize, kfontname, kborder, MsgBox, Range or End Sub. If a line begins with anything else, it is wrapping in this book. But, you must not type it in the VBA Module window as wrapping text. You must continue to type wrapping lines in this book on a single line in the VBA Module window in Microsoft Excel.

```
Sub tellme()
Dim kfill As Variant
Dim kfont As Variant
Dim kbold As Variant
Dim kital As Variant
Dim kfontsize As Variant
Dim kfontname As Variant
Dim kborder As Variant
kfill = Range("A1").Interior.ColorIndex
kfont = Range("A1").Font.ColorIndex
kbold = Range("A1").Font.Bold
kital = Range("A1").Font.Italic
```

```
kfontsize = Range("A1").Font.Size
kfontname = Range("A1").Font.Name
kborder = Range("A1").BorderAround
MsgBox "Border count per cell sides and inside: " & Range("A1").Borders.Count
MsgBox "Border Line Syle Is: " & Range("A1").Borders(xlEdgeLeft).LineStyle
MsgBox "Bottom Border: " & Range("A1").Borders(xlEdgeBottom).LineStyle
MsgBox "Right Border: " & Range("A1").Borders(xlEdgeRight).LineStyle
MsgBox "Top Border: " & Range("A1").Borders(xlEdgeTop).LineStyle
MsgBox "Diagonal Down Border: " &
Range("A1").Borders(xlDiagonalDown).LineStyle
MsgBox "Inside Horizontal Border: " &
Range("A1").Borders(xlInsideHorizontal).LineStyle
MsgBox "Border Weight: " & Range("A1").Borders(xlEdgeTop).Weight
Range("A1").Borders(xlEdgeTop).Weight = 2
MsgBox "Font size: " & kfontsize
MsgBox "Font Name is: " & kfontname
End Sub
```



```
Border count per cell sides and inside: 6
```

OBJECT EXERCISE 7: this shows the result of running the code, if it was correctly typed in the VBA Module window.

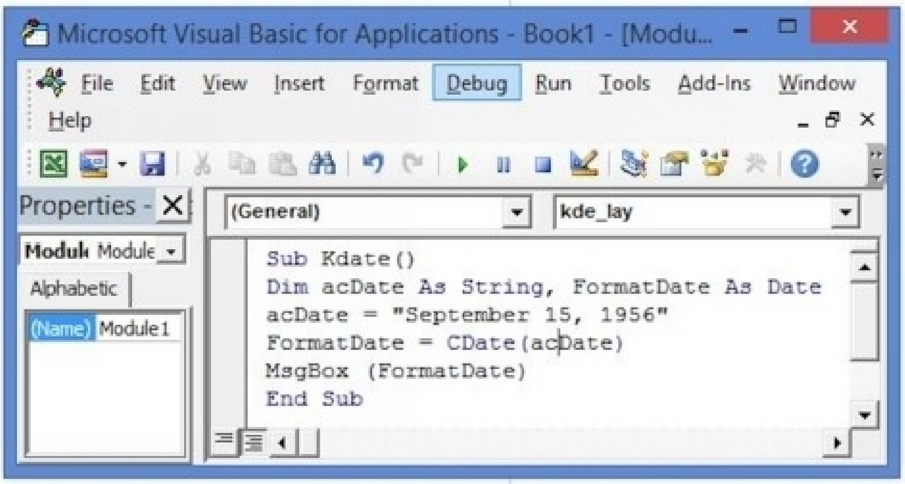
EXERCISE 8: this code shows how to format a date.

```
Sub Kdate()
Dim acDate As String, FormatDate As Date
acDate = "September 15, 1956"
```

FormatDate = CDate(acDate)

MsgBox (FormatDate)

End Sub


Microsoft Visual Basic for Applications - Book1 - [Modu...
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
Properties -
Module Module
Alphabetic
(Name) Module1
(General)
kde_lay

```
Sub Kdate()
Dim acDate As String, FormatDate As Date
acDate = "September 15, 1956"
FormatDate = CDate(acDate)
MsgBox (FormatDate)
End Sub
```

OBJECT EXERCISE 8: this shows how to correctly type the code in the VBA Module window.

EXERISE 9: this code gives users an idea on how to insert a table into a spreadsheet and fill it with text; it shows how to insert a picture from a hard drive; it shows how to insert an oval shape, and it shows how to format a text box that has been entered into the spreadsheet. Before running this code, I want you to go to your **Pictures** folder, and I want you to create a new folder inside the Pictures folder and name it **Samples**. Next I want you to copy one of your pictures and paste it inside the new Sample folder; next, I want you to rename that picture to **A1.JPG**. Once you've done this, I want you to return to your Microsoft Excel application, and I want you to go back into the VBA Module window, and I want you to run this code.

Once you've successfully ran the code, I want you to make some changes (on your own). I only want you to change what is between the double quotation marks (**" "**), e.g, **"Kenny"** changed to **"ooga booie", Range("$A$1:$B$5")** changed to **Range("C3:E8")**. Be sure to download my text file that contains this code, so you understand who the code must be typed inside the VBA Module window versus how it looks in this book. The lines must not wrap in the VBA window, but they'll definitely wrap in this book.

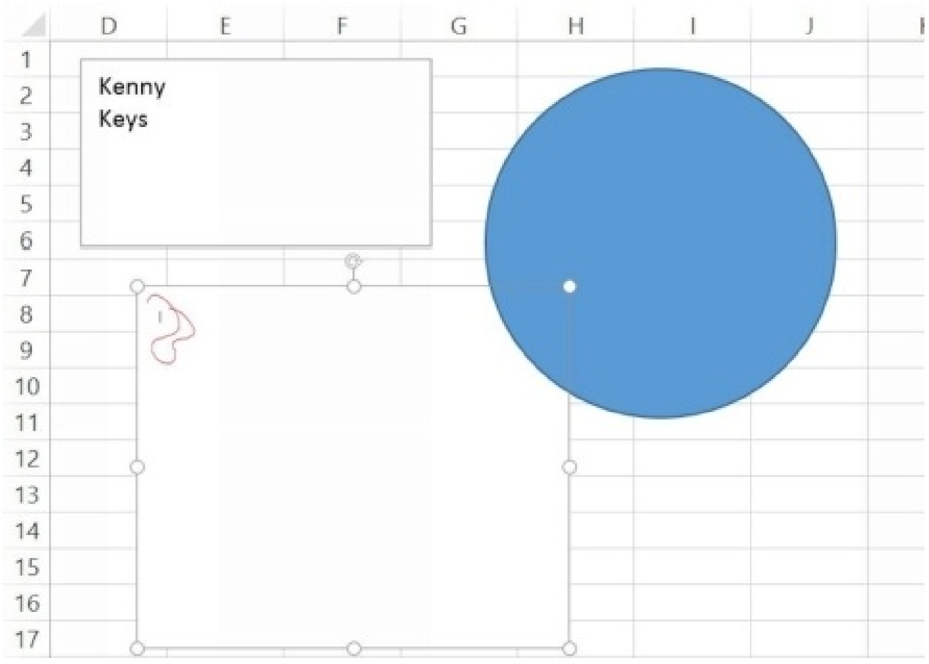
Sub Macro1()

ActiveSheet.ListObjects.Add(xlSrcRange, Range("$A$1:$B$5"), , xlYes).Name = "Table1"

Range("Table1[#All]").Select

ActiveSheet.Pictures.Insert("C:\Users\MedicMan\Pictures\Samples\A1.JPG"). Select

```
ActiveSheet.Shapes.AddShape(msoShapeOval, 330.6, 80.7, 144, 144).Select
ActiveSheet.Shapes.AddTextbox(msoTextOrientationHorizontal, 330.6, 80.7, 144,
144).Select
Selection.ShapeRange(1).TextFrame2.TextRange.Characters.Text = “Kenny” &
Chr(13) & “Keys”
Selection.ShapeRange(1).TextFrame2.TextRange.Characters(1,
16).ParagraphFormat. FirstLineIndent = 0
With Selection.ShapeRange(1).TextFrame2.TextRange.Characters(1, 16).Font
.NameComplexScript = “+mn-cs”
.NameFarEast = “+mn-ea”
.Fill.Visible = msoTrue
.Fill.ForeColor.ObjectThemeColor = msoThemeColorDark1
.Size = 11
.Name = “+mn-lt”
End With
Selection.ShapeRange
End Sub
```



Kenny
Keys

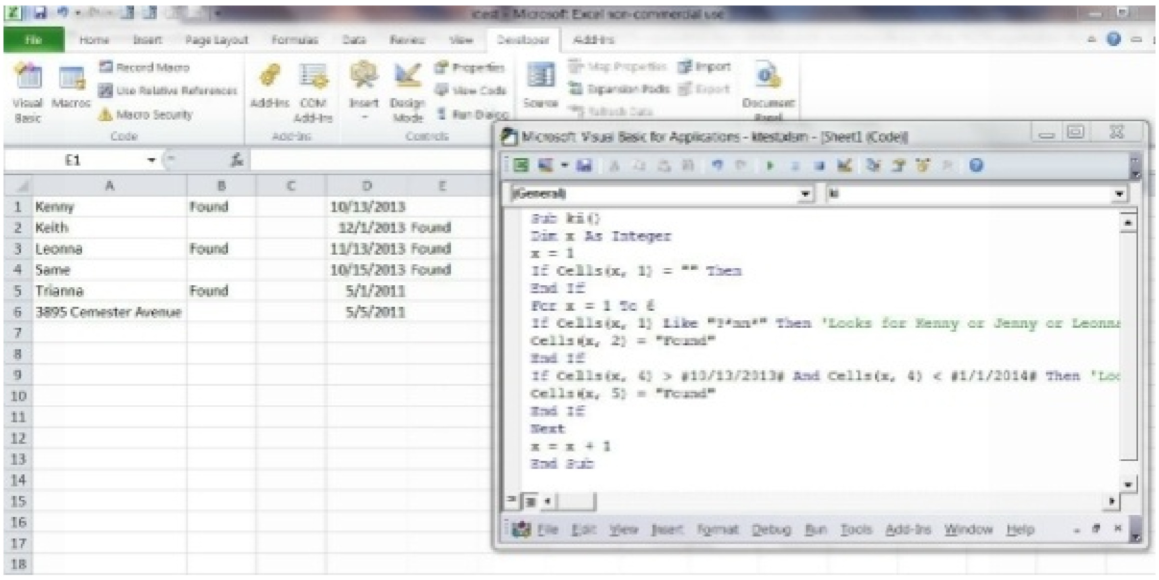
OBJECT EXERCISE 9: this shows the result of correctly typing the VBA code into the VBA Module window and running it.

Combining Microsoft’s Like, And, Less Than & Greater Than Operators with Numeric, Single Character & Any Number of Characters Wildcards in VBA Statements: Chapter 4

In this chapter, you’ll learn how to use VBA operators to filter for data by indicating if a character is before, after or in between a single character or multiple characters in text strings located in Microsoft Excel spreadsheets, and you’ll learn how to filter for specific date ranges.

VBA Code 1:

```
Sub ki()
Dim x As Integer
x = 1
If Cells(x, 1) = ”” Then
End If
For x = 1 To 6
If Cells(x, 1) Like “?*nn*” Then ‘Looks for Kenny or Jenny or Leonna
Cells(x, 2) = “Found”
End If
If Cells(x, 4) > #10/13/2013# And Cells(x, 4) < #1/1/2014# Then ‘Looks for Kenny, Leonna and Trianna
Cells(x, 5) = “Found”
End If
Next
x = x + 1
End Sub
```



	A	B	C	D	E
1	Kenny	Found		10/13/2013	
2	Keith			12/1/2013	Found
3	Leonna	Found		11/13/2013	Found
4	Same			10/15/2013	Found
5	Trianna	Found		5/1/2011	
6	3895 Cemester Avenue			5/5/2011	

```
Sub ki()
Dim x As Integer
x = 1
If Cells(x, 1) = "" Then
End If
For x = 1 To 6
If Cells(x, 1) Like "?*nn*" Then 'Looks for Kenny or Jenny or Leonna
Cells(x, 2) = "Found"
End If
If Cells(x, 4) > #10/13/2013# And Cells(x, 4) < #1/1/2014# Then 'Loc
Cells(x, 5) = "Found"
End If
Next
x = x + 1
End Sub
```

Image 1: this shows a spreadsheet that contains the VBA *Like* operator code being used with the *For…Next* statement and the *If* statement.

In *VBA Code 1* and *Image 1*, the *Like* operator is being used in this code to search for any and all words in column A, starting from A1 downwards until it reaches a cell in column A that is blank, e.g., cell A7, that contains double n's, e.g., Ke*nn*y, Leo*nn*a, Tria*nn*a; when it finds the word, it places the word *Found* in the corresponding B column next to the word in column A. The next command of the code shown in this image requires that Microsoft Excel search column D for any date range that is *greater than* October 13, 2013, i.e., a date *after* October 13, 2013, and *less than* January 1, 2014, i.e., a date that is *before* January 1, 2014; when it finds these date ranges, it is to place the word *Found* next to them in the corresponding E column, as shown in E2, E3 and E4 in this image.

It's time to break down the code shown in *VBA Code 1* and *Image 1*, Line by Line:

[Line 1:] This is the name of the container that holds the entire VBA code, e.g., ki().

[Line 2:] This creates a new integer or counter named *x* that will store a number to be used in the loop statement in this code.

[Line 3:] This instructs Microsoft Excel to start the new counter named *x* at number *1*; this will be used to start looking in row *1* of the spreadsheet shown in *Image 1*.

[Line 4:] This line of the VBA code instructs Microsoft Excel to use the *Cells* object to reference the cell location in the spreadsheet. When using the *Cells* object, it has two criteria: the first criterion is the *row*; the second criterion is the *column*, so this line means that if row *x* or *1* and column *1* is equal to an empty cell, Microsoft Excel must do what is requested on *line five*.

[Line 5:] This line instructs Microsoft Excel to *do nothing*, i.e., *stop the counter from counting to the next number* and *no longer execute the code below this line*.

[Line 6:] This line means that *if line four is not true*, i.e., *cell A1 is not an empty cell*, Microsoft Excel *must execute the code below this line, starting on line seven, as long as the counter named x is 1 to 6*; if the counter reaches 7, Microsoft Excel must stop the counter and cease executing the code.

[Line 7:] This line tells Microsoft Excel to look for any and all content in row x or 1 and column 1, e.g., A1, and *find any word that contains any character that precedes a set of double n's*, e.g., *?nn*; after the first single character, it could be one character, two characters or many characters that follows, before reaching the set of double n's, e.g., *?*nn*; after Microsoft finds a cell in column A that contains a set of double n's, that word must not end with the double set of n's; instead, it must continue on with one or more characters after the set of double n's, e.g., *?*nn**. The *question mark (?)* is the wildcard symbol for any single character found in a text string, e.g., a, ?, -, P, 4, etc…; the *asterisk (*)* is the wildcard symbol for any single character or multiple characters found in a text string, e.g., a, 8, jop4Um, $(#, etc…, but the *nn* shown in the search criterion must be absolute, i.e., the cell must contain a set of double n's somewhere in that cell, e.g., Ke*nn*y, Leo*nn*a, Pe*nn*y, Filioso*nn*i, etc…

[Line 8:] This line means that if the cell in row x or 1 and column 1 contains a word that has *nn* in it, Microsoft Excel must place the word *Found* on that same row, but in the corresponding column B, i.e., if cell A1 contains the word *Kenny*, Microsoft Excel must type *Found* in column B1; however, if cell A1 is not empty, but it contains the word *Melanie*, Microsoft Excel must leave cell B1 empty.

[Line 9:] This line ends the logical *If* statement for the executable lines of code above it.

[Line 10:] This line tells Microsoft Excel that if the cell in row x or 1 and column 1, e.g., A1, contains a date that is *greater than* or *after* October 13, 2013 and that same date is *less than* or *before* January 1, 2014, it must place *Found* in the corresponding column D.

[Line 11:] If cell A1 contains a date that is *after* October 13, 2013 and *before* January 1, 2014, Microsoft Excel will place the word *Found* in cell E1; however, *Image 1* clearly shows that the date does not meet both criteria

of the *And* statement, so nothing must appear in cell E1.

[Line 12:] This line closes out the *If* statement for the date range criteria.

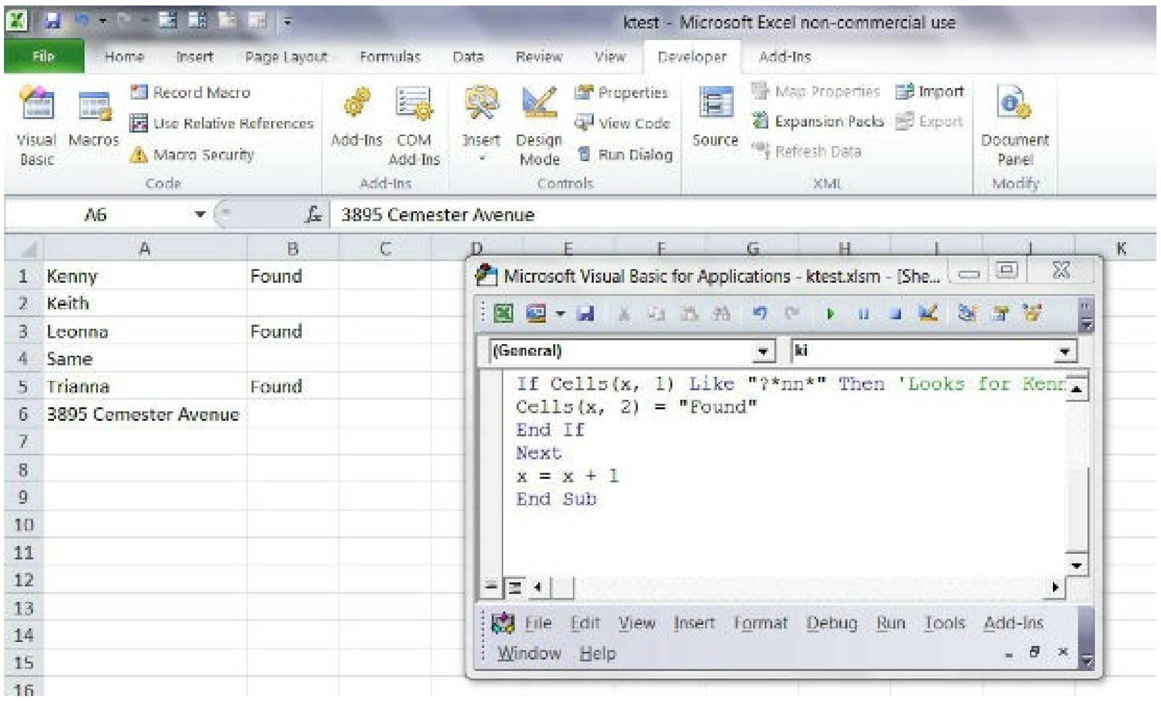
[Line 13:] This line tells Microsoft Excel that it’s time to go to the next line and increase the *x* number container, so it can move to a new row to start the process all over again.

[Line 14:] This instructs Microsoft Excel to see what the number container named *x* currently equals, so it can add *1* to it. Currently, *x* equals *1*, so *x = x + 1* will make *x = 2*; this means Microsoft will now go to row 2 column 1, e.g, A2, and start the process all over again. This process will continue to repeat itself until Microsoft comes to a row in column 1 or A that is empty, e.g., A7.

[Line 15:] This tells Microsoft Excel that the full VBA code is now complete.

VBA Code 2:

```
Sub ki()
Dim x As Integer
x = 1
If Cells(x, 1) = ”” Then
End If
For x = 1 To 6
If Cells(x, 1) Like “?*nn*” Then ‘Looks for Kenny, Leonna & Trianna
Cells(x, 2) = “Found”
End If
Next
x = x + 1
End Sub
```


ktest - Microsoft Excel non-commercial use
File Home Insert Page Layout Formulas Data Review View Developer Add-Ins
Visual Basic
Macros
Record Macro
Use Relative References
Macro Security
Code
Add-Ins
COM Add-Ins
Insert
Design Mode
Properties
View Code
Run Dialog
Controls
Source
Map Properties
Expansion Packs
Refresh Data
Import
Export
XML
Document Panel
Modify
Microsoft Visual Basic for Applications - ktest.xlsm - [She...
(General)
ki
File Edit View Insert Format Debug Run Tools Add-Ins Window Help

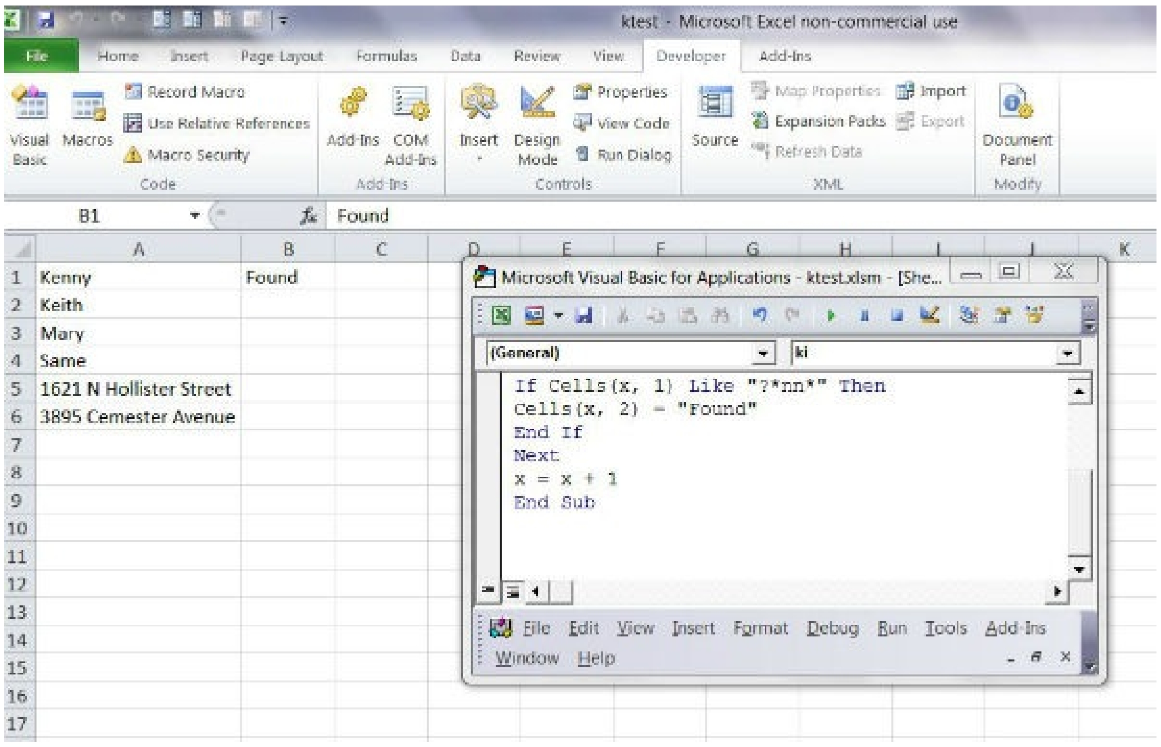
A6 3895 Cemester Avenue

	A	B
1	Kenny	Found
2	Keith	
3	Leonna	Found
4	Same	
5	Trianna	Found
6	3895 Cemester Avenue	

```
If Cells(x, 1) Like "?*nn*" Then 'Looks for Kenn
Cells(x, 2) = "Found"
End If
Next
x = x + 1
End Sub
```

Image 2: this shows a simplified VBA code containing the *If* and *For…Next* statements with the *Cells* object and the *Like* operator.

In *Image 2*, the VBA code instructs Microsoft Excel to look for any word that contains double n’s, e.g., Ke*nn*y in A1, Leo*nn*a in A3, Tria*nn*a in A5; once the words are found, Microsoft Excel must place the word *Found* next to them in the corresponding B column, e.g., B1, B3 and B5.

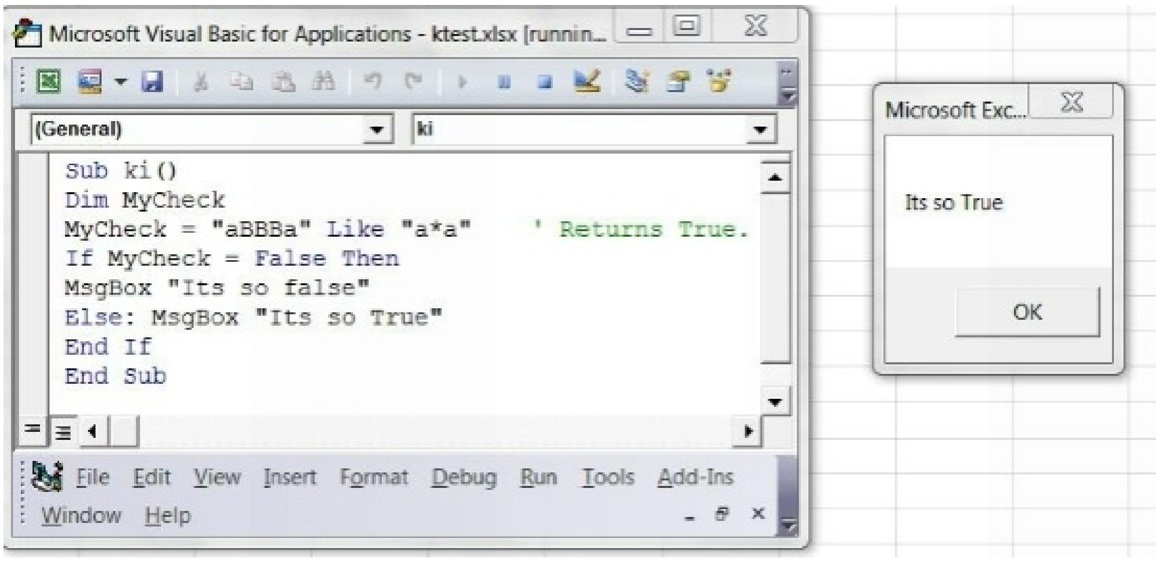

ktest - Microsoft Excel non-commercial use
File Home Insert Page Layout Formulas Data Review View Developer Add-Ins
Visual Basic
Macros
Record Macro
Use Relative References
Macro Security
Code
Add-Ins
COM Add-Ins
Insert
Design Mode
Properties
View Code
Run Dialog
Controls
Source
Map Properties
Expansion Packs
Refresh Data
Import
Export
XML
Document Panel
Modify
Microsoft Visual Basic for Applications - ktest.xlsm - [She...
(General)
ki
File Edit View Insert Format Debug Run Tools Add-Ins Window Help

B1 Found

	A	B
1	Kenny	Found
2	Keith	
3	Mary	
4	Same	
5	1621 N Hollister Street	
6	3895 Cemester Avenue	

```
If Cells(x, 1) Like "?*nn*" Then
Cells(x, 2) = "Found"
End If
Next
x = x + 1
End Sub
```

Image 2a: Just like in *Image 1* and *Image 2*, this image shows a spreadsheet that contains the VBA *Like* operator code being used with the *For…Next* statement and the *If* statement; the *Like* operator is being used in this code to search for any and all words in column A, starting from A1 downwards until it reaches a cell in column A that is blank, e.g., cell A7, that contains double n’s, e.g., Ke*nn*y; when it finds the word, it places the word *Found* in the corresponding B column next to the word in column A.


Microsoft Visual Basic for Applications - ktest.xlsx [runnin...
(General)
ki
File Edit View Insert Format Debug Run Tools Add-Ins
Window Help
Microsoft Exc...
OK

```
Sub ki()
Dim MyCheck
MyCheck = "aBBBa" Like "a*a"     ' Returns True.
If MyCheck = False Then
MsgBox "Its so false"
Else: MsgBox "Its so True"
End If
End Sub
```

```
Its so True
```

Image 3: this example shows a VBA code that contains the *Like* operator in conjunction with the *wildcard character*s for *any single or multiple number of characters (*)*; in this case, the *asterisk (*)* means search for any word that starts and ends with a lowercase *a*, but the characters in between can be uppercase, lowercase, symbols, letters or any known character used in writing. If a word is found beginning and ending with a lowercase *a*, Microsoft Excel must produce a message box that reads: *Its so true*; if a word isn't found matching the description, Microsoft Excel must produce a message box that reads: *Its so false*.

Note: in *Image 3*, the apostrophe is intentionally left out of the word *Its*. Grammatically, the word would read: It's.

In *VBA Code 3* and *Image 3*, Microsoft Excel is being asked to create a container named *MyCheck* and fill it with the two words compared by the *Like* operator (e.g., *"aBBa" Like "a*a");* in this case, the comparison is a match, because both words begins and ends with a lowercase *a*, and the *asterisk (*)* is the *wildcard* symbol for any combination of characters from a single character to multiple characters (e.g., abb*a*, **aBBa**, *a*BBc*a*, *a*Rkifa, *aa*a); however, if the comparison was *"aBBa" Like "a?a"*, it would *not* be a match, because the *question mark (?) wildcard symbol* only represent *any single character*, e.g., a*1*a, a*J*a, a*7*a, a*A*a, a*B*a, but if it was written *"aBBa" Like "a??a"*, the comparison would be a match, because it would equate to both words beginning and ending with a lowercase *a* and containing two characters between them, e.g., *a*<u>7Y</u>*a*, ***a*<u>BB</u>*a***, *a*<u>CD</u>*a*, *a*<u>6t</u>*a*.

Definitions of Operators in VBA

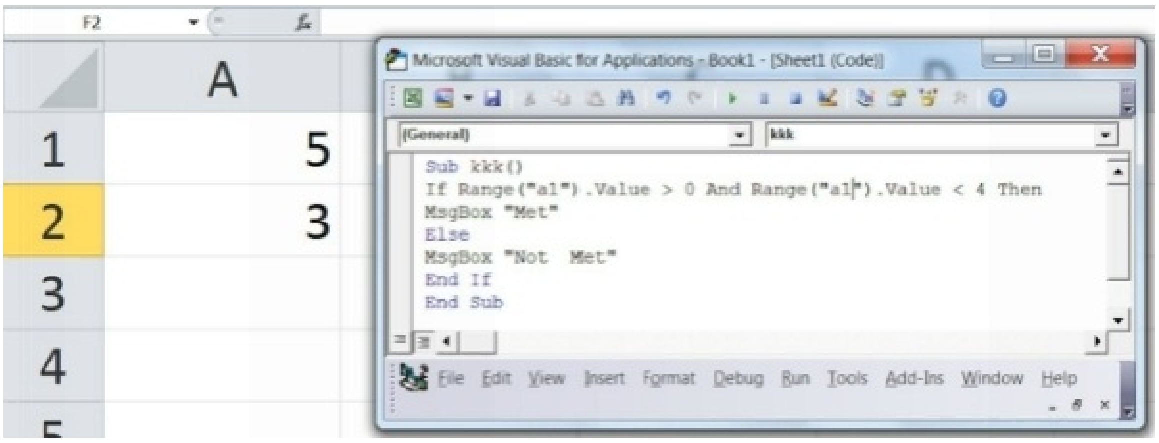
This section explains how to use Microsoft VBA operators and wildcard characters. To ensure that everyone understand these operators, I’ll be describing them in everyday language, instead of using their official descriptions and names, as in the case with these two wildcard characters: asterisk (*) and question mark (?).

And Operator

The *And* operator means that both criteria must be met in order for the statement to be true. Below, you’ll find two examples of using the *And* operator in VBA: one meets both criteria using the *And* operator, and the other one only meets one criterion of the *And* operator.

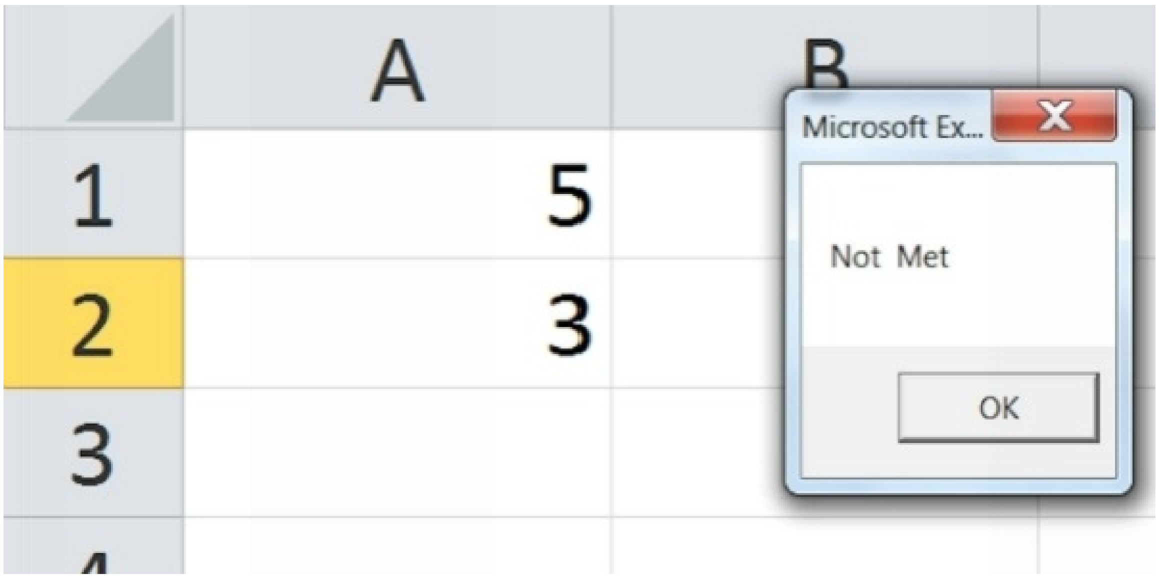
VBA *And* Operator Example Code 1:

```
Sub kkk()
If Range(“a1”).Value > 0 And Range(“a1”).Value < 4 Then
MsgBox “Met”
Else
MsgBox “Not  Met”
End If
End Sub
```



```
Sub kkk()
If Range("a1").Value > 0 And Range("a1").Value < 4 Then
MsgBox "Met"
Else
MsgBox "Not  Met"
End If
End Sub
```

VBA *And* Operator Image 1: this shows *VBA And Operator Example Code 1* in the Excel module window before running the code.



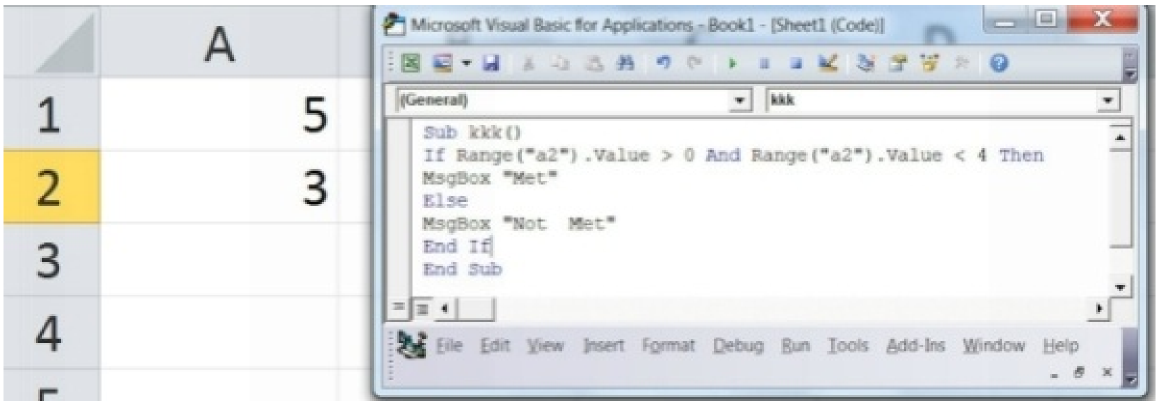
Not Met

VBA *And* Operator Image 1a: this shows *VBA And Operator Example Code 1* result in the Main Microsoft Excel spreadsheet window after running the code.

Using *VBA And Operator Example Code 1*, *VBA And Operator Image 1* and *VBA And Operator Image 1a*, you see that *cell A1 doesn't contain a number that is both greater than zero and less than four*, so *the statement is false*, because both criteria have to be met in order for the statement to equal true; therefore the result is a message box that reads: *Not Met*. However, if I was to change the VBA code range or location cell from A1 (5) to A2 (3), both criteria are met, because *three is greater than zero, and it's also less than four, so it equals true*, and a message box is produced that reads: *Met*. You can see the result in *VBA And Operator Image 2* and *VBA And Operator Image 2a*.

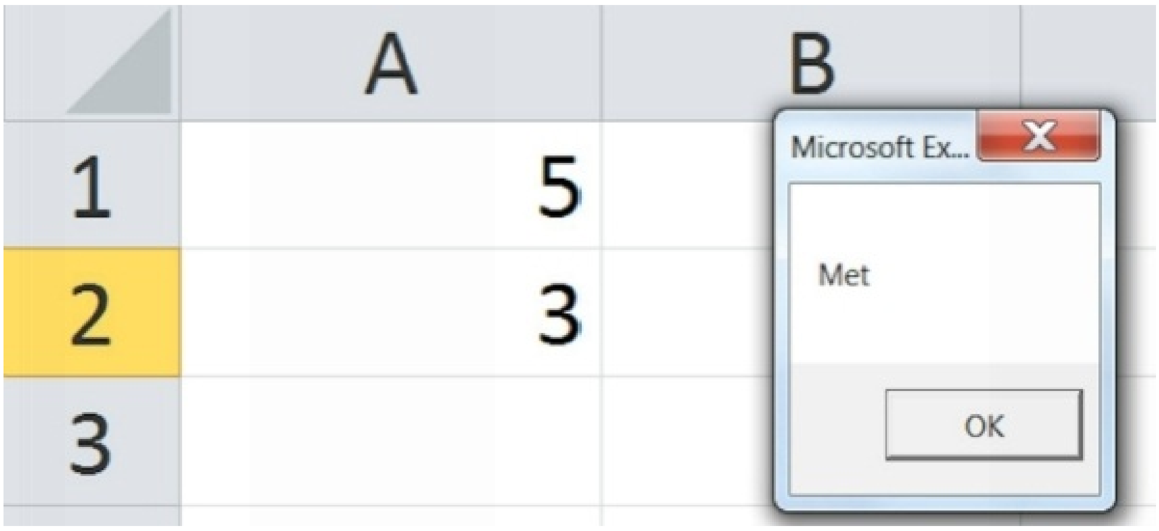
VBA And Operator Example Code 2:

```
Sub kkk()
If Range(“a2”).Value > 0 And Range(“a2”).Value < 4 Then
MsgBox “Met”
Else
MsgBox “Not  Met”
End If
End Sub
```



```
Sub kkk()
If Range("a2").Value > 0 And Range("a2").Value < 4 Then
MsgBox "Met"
Else
MsgBox "Not  Met"
End If
End Sub
```

VBA *And* Operator Image 2: this shows *VBA And Operator Example Code 2* in the Excel module window before running the code.



VBA *And* Operator Image 2a: this shows the result of running *VBA And Operator Example Code 2*.

[Greater Than (>) Operator]

The *greater than (>)* operator requires that the number on the left of the operator be larger than the number to the right of the operator, e.g., 5>3, 100>99 and 87>46.

[Less Than (<) Operator]

The *less than (<)* operator requires that the number to the left of the operator be smaller than the number to the right of the operator, e.g., 3<5, 99<100 and 1<58.

[Any Single Character (?) Wildcard]

The *any single character (?)* wildcard is used to indicate any single character in a text string, as in my name, *Kenny*, e.g., K?nny, Ke??y, ?e?ny, K????.

Any Single Character or Any Multiple Number of Characters (*) Wildcard

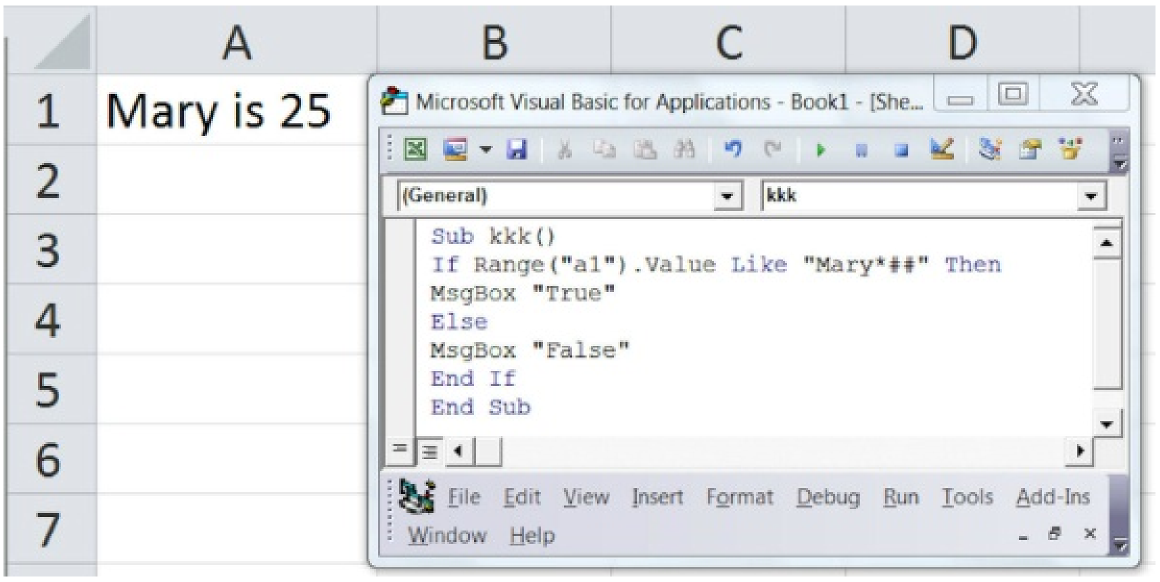
The *any single character or any multiple number of characters (*)* wildcard is used to indicate any number of characters, as in my name, Kenny, e.g., K*y, Ke*n*, *y, K*.

Any Numeric (#)Wildcard Character

The *numeric (#)* wildcard character is placed on both ends of a date, e.g., #10/01/2013#, #01/05/13#, #06/05/2011#. It is place inside of the double quotation marks to indicate that a numeric character must be used at a specific position in the text string, e.g., “Kenny is 2# years old.”

Numeric Wildcard Character VBA Code 1:

```
Sub kkk()
If Range(“a1”).Value Like “Mary*##” Then
MsgBox “True”
Else
MsgBox “False”
End If
End Sub
```

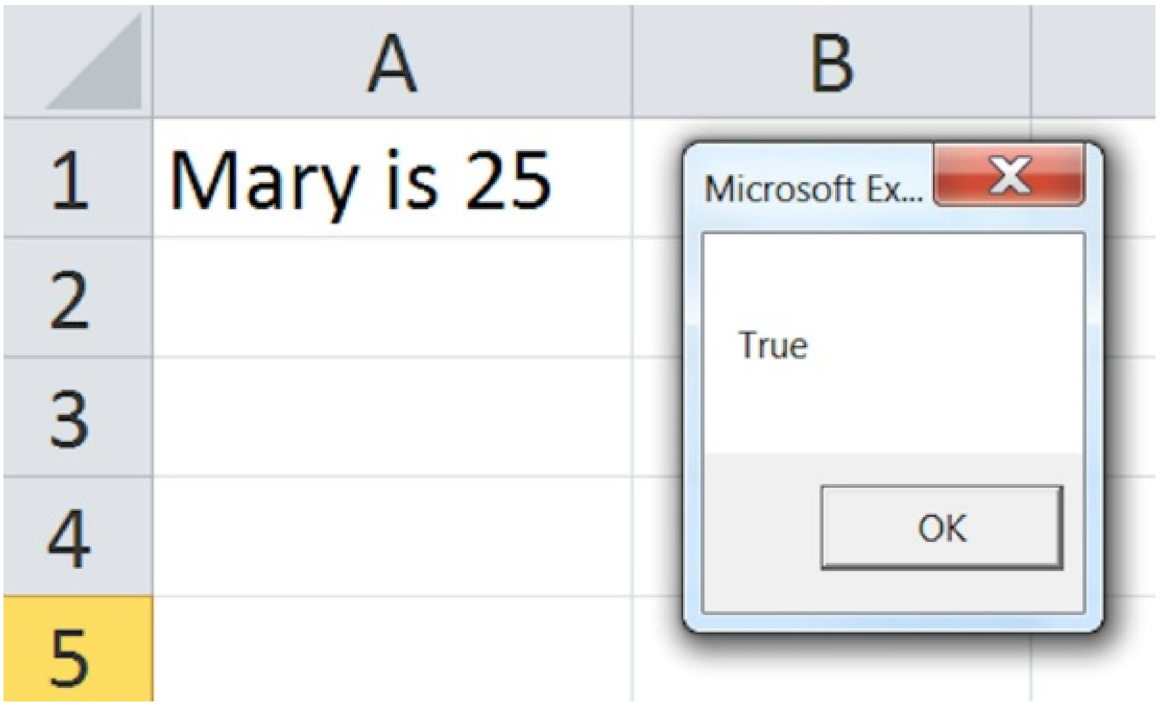


Mary is 25

```
Sub kkk()
If Range("a1").Value Like "Mary*##" Then
MsgBox "True"
Else
MsgBox "False"
End If
End Sub
```

A
B
C
D
1
2
3
4
5
6
7
Microsoft Visual Basic for Applications - Book1 - [She...
(General)
kkk
File Edit View Insert Format Debug Run Tools Add-Ins
Window Help

Numeric Wildcard Character Image 1: this shows how the *numeric (#)* wildcard character filters for a text string that contains a 2-digit number (e.g., “Mary is ##”); this is before running the code.



Mary is 25

True

Numeric Wildcard Character Image 1a: this shows how the *numeric (#)* wildcard character filters for a text string that contains a 2-digit number (e.g., “Mary is ##”); this is after running the code.

Using *Numeric Wildcard Character VBA Code 1*, *Numeric Wildcard Character Image 1* and *Numeric Wildcard Character Image 1a*, the VBA *If* statement results in *true*, because the content in cell A1 starts with the word *Mary*, and there are other characters after *Mary*, but the text string in cell A1 ends with a 2-digit numeric number; thus, you have the *##* in the VBA *If* statement shown in *Using Numeric Wildcard Character VBA Code 1* and *Numeric Wildcard Character Image 1* that leads to the result shown in *Numeric Wildcard Character Image 1a*.

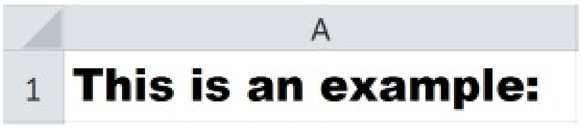
Like Operator

The *Like* operator is used to show a likeness between two text strings using other wildcard operators (e.g., *, ?, #); these are a few examples: “Kenny” Like “K*”, “Kenny” Like “K?nn?”, “Kenny” Like “*ny”, Range(“a1”).Value Like #10/1/2013#.

Microsoft Excel Formulas for Data Extractions: Chapter 5

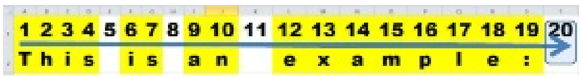
It's important for users to begin each sample formula with the required equal sign; I won't be including them in this book.

The **LEN()** [**length** of characters] formula requires a reference [location] cell, e.g., **LEN(**A1**)**, or a string [characters between quotation marks], e.g., **LEN(**"Kenny Keys"**)**.



	A
1	This is an example:

Figure 1-1: Contents of cell **A1**.

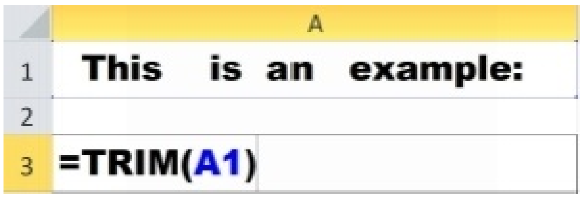


1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	h	i	s		i	s		a	n		e	x	a	m	p	l	e	:	

Figure 1-2: Microsoft Excel counting characters in cells; this table applies to the string shown in **Figure 1-1.**

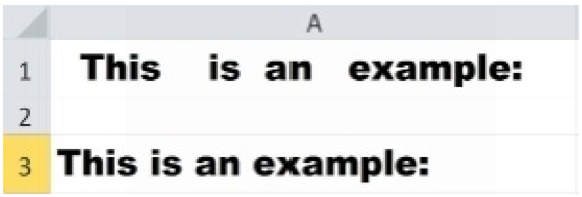
Using **Figure 1-1** and **Figure 1-2, if the formula LEN(*A1*)** is typed in cell **B1**, the result will be **20**. There are **four** spaces shown in **Figure 1-2, e.g., 5, 8, 11, and 20.** All characters are counted; spaces are characters.

The **TRIM()** [**removes** excess (unnecessary) spaces] formula requires a reference cell or characters, e.g., **TRIM(**A1**)** or **TRIM(**"Kenny Keys"**)**.



	A
1	This is an example:
2	
3	=TRIM(A1)

Figure 1-2.9: Cell A3 contains the **TRIM() formula; the reference cell is A1.**

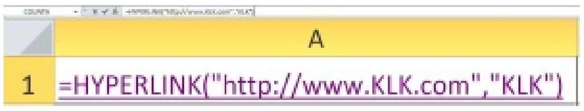


	A
1	This is an example:
2	
3	This is an example:

Figure 1-3: Cell A3 contains the **TRIM()** formula; the reference cell is **A1**.

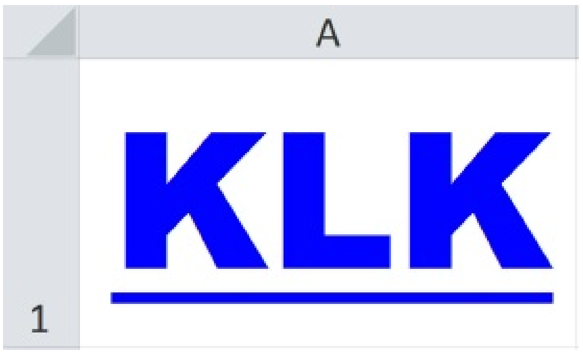
Formulas with Two Requirements

The **HYPERLINK()** formula has two requirements; quotation marks are placed around its requirements, e.g., **HYPERLINK("*http://www.KLK.com*","KLK")**.



	A
1	=HYPERLINK("http://www.KLK.com","KLK")

Figure 1-3.9: This is the **HYPERLINK()** formula that is shown above.



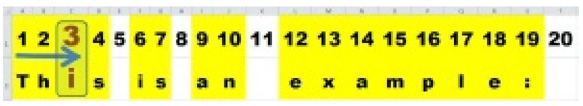
	A
1	KLK

Figure 1-4: This is the result of the **HYPERLINK()** formula that is shown above **Figure 1-3.9**.

There are two other ways to use the **HYPERLINK()** formula: HYPERLINK(“\specialdriver\1st\test\”,“Sample of shared drive location”), i.e., this opens a shared drive.

HYPERLINK(“[c:\temp\sample.xls]sample”,“Sample Workbook”), i.e., this opens another workbook.

The **LEFT()** formula has two requirements, e.g., **LEFT(**A3,3**)** or **LEFT(**“Kenny”,3**)**.

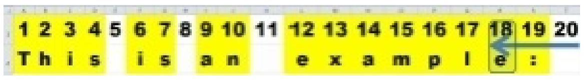


1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	h	i	s		i	s		a	n		e	x	a	m	p	l	e	:	

Figure 1-5: Microsoft Excel counting characters rightward in a cell. This table applies to **Figure 1-3**, cell **A3**.

Using **Figure 1-3** and **Figure 1-5**, if the formula LEFT(A**3,3**) is typed in cell **B3**, the result will be **Thi**.

The **RIGHT()** formula has two requirements, e.g., **RIGHT(**A3,3**)** or **RIGHT** **(**“Kenny”,3**)**.

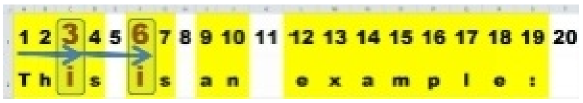


1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	h	i	s		i	s		a	n		e	x	a	m	p	l	e	:	

Figure 1-6: Microsoft Excel counting characters leftward in a cell. This table applies to **Figure 1-3**, cell **A3**.

Using **Figure 1-3** and **Figure 1-6**, if the formula RIGHT(A3**,3**) is typed in cell **B3**, the result will be **e: .**

The **FIND()** formula has two requirements, e.g., **FIND(**“i”,A3**)** or **FIND(**C1,A3**)**.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	h	i	s		i	s		a	n		e	x	a	m	p	l	e	:	

Figure 1-7: Microsoft Excel counting characters rightward in a cell. This table applies to **Figure 1-3**, cell **A3**.

Using **Figure 1-3** and **Figure 1-7**, if the formula FIND(“i”***,A3***) is typed in cell **B3**, the result will be the number **3**. This formula falls under the **three** requirements formula category when searching for multiple locations of the same character in a cell, i.e., in the case of the letter “i” in **Figure 1-3** and **Figure 1-7**. To find the location of the **second** (2**nd**) “i” that is located in cell **A3**, a third requirement is added to the formula, e.g., **FIND(**“i”,A3, **FIND(“i”,A3)+1)**.

Using **Figure 1-3** and **Figure 1-7**, the formula FIND(“**i”*,A3*,FIND(“i”,A3)+1**) is typed in cell **B3**, the result will be the number **6**. Microsoft Excel treats the formula as basic math, e.g., **SUM(10*5)/2** first solves 10*5, which is 50, and then it solves 50/2, which is **25**, i.e., anything between the inmost [farthest inward] parentheses is solved first. The **FIND()** and other formulas are handled the same way. Microsoft Excel solves nestled [inserted, i.e., one inside another] formulas inside out.

FIND(“i”,A3,FIND(“i”,A3)+1)

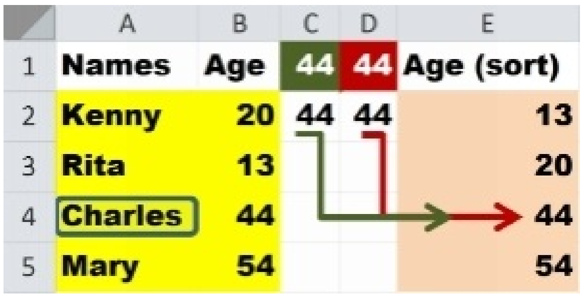
FIND(“i”,A3,3+1)

FIND(“i”,A3,4) final formula to execute

Using **Figure 1-3** and **Figure 1-7**, Microsoft Excel locates the **fourth (4th)** character in cell **A3**, and it moves rightward until it finds the next “i,” as shown in **Figure 1-3** and **Figure 1-7**. The result is **6**.

The **SMALL()** formula has **two** requirements. Using **Figure 1-8**, cell **C1**, the formula SMALL**(*B2:B5*,3**) instructs Microsoft Excel to look for the **third (3rd)** smallest number in cell range [group of cells, e.g., **A1:J9**, **B3:E3**, **D2:D10**, **NAMES**, **CITIES**] **B2:B5**, which is **44**.

The **LARGE()** formula has **two** requirements. Using **Figure 1-8**, cell **D1**, the formula LARGE**(*B2:B5*,2**) instructs Microsoft Excel to look for the **second (2nd)** largest number in cell range **B2:B5**, which is **44**.



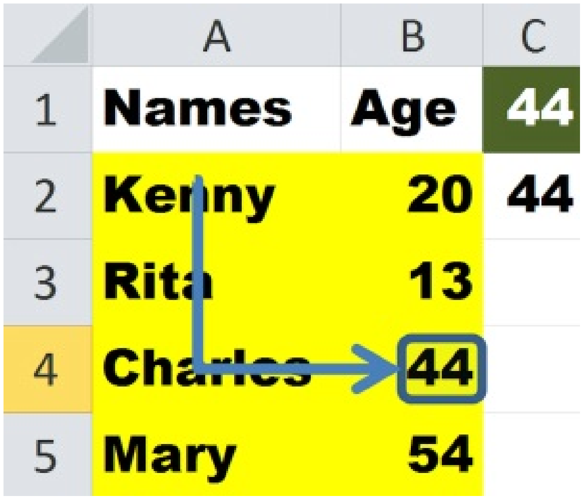
	A	B	C	D	E
1	Names	Age	44	44	Age (sort)
2	Kenny	20	44	44	13
3	Rita	13			20
4	Charles	44			44
5	Mary	54			54

Figure 1-8: Cell **C1** contains the **SMALL()** formula, and cell **D1** contains the **LARGE()** formula.

Formulas with Three Requirements

The **MID()** formula has **three** requirements, e.g., **MID(**A3,1,3**)** or **MID(**A3,12,3**)** or **MID(**"Kenny",3,3**)**. Using **Figure 1-3** and **Figure 1-5**, if the formula MID(A3,**1,3**) is typed in cell **B3**, the result will be **Thi**. The first and the last requirements of the **MID()** formula does exactly what the **LEFT()** formula does; however the **second (2nd)** requirement tells Microsoft Excel where to begin its count, e.g. MID(A3**,12,7**), i.e., in cell **A3**, go to **twelfth** (**12th**) character–this will be character number one (1)—from this point, extract **seven** (7) characters, which is **example**.

The **INDEX()** formula has **three** requirements, e.g., **INDEX(**A2:b5,3,2**)**, i.e., **INDEX(**range, row, column**)**.



	A	B	C
1	Names	Age	44
2	Kenny	20	44
3	Rita	13	
4	Charles	44	
5	Mary	54	

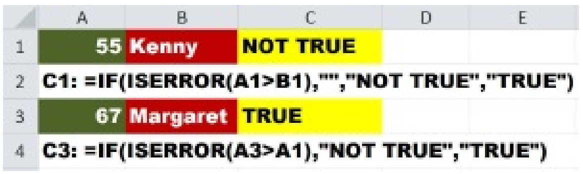
Figure 1-9: Cell **C1** contains an **INDEX()** formula.

Using **Figure 1-9**, the formula INDEX(***A2:b5*,**3**,2**) is typed in cell **C1**, the result will be **44**. Microsoft Excel starts at cell **A2**, counting it as row one, it counts down two more rows, which would be cell **A4**; next, it counts columns; cell **A4** is the first column; one column to the right is the second column; thus, the cell location is **B4**; therefore, the result is **44**.

The **IF()** formula has **three** requirements, e.g., IF(***3>1***,"**This is TRUE**", **"This is**

FALSE”). We know that **3** is greater than **1**, so the statement is TRUE; therefore, the result will be **This is TRUE**. With a false equation, e.g., IF(***3>55,*”This is TRUE**“, **“This is FALSE”),** we know **3** is **not** greater than **55**; therefore, the result will be **This is FALSE**.

The **IF()** formula can sometimes lead to errors (e.g., #DIV/0!, #N/A, #NAME?, #NULL!, #NUM!, #REF!, #VALUE!), so we nestle it with the **ISERROR()** formula. If the formula IF(**ISERROR(**7>kenny**),“Not a proper equation”,“proper equation”**) is typed in any cell, the result will be **Not a proper equation**, i.e., it’s true that this equation is illogical; therefore, the TRUE result is executed, i.e., the one after the equation.

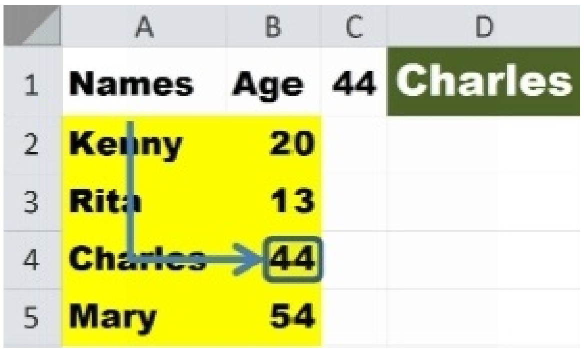


	A	B	C	D	E
1	55	Kenny	NOT TRUE		
2	C1: =IF(ISERROR(A1>B1),"","NOT TRUE","TRUE")				
3	67	Margaret	TRUE		
4	C3: =IF(ISERROR(A3>A1),"NOT TRUE","TRUE")				

Figure 1-10: Cells **C1** and **C3** contains **IF()** and **ISERROR()** formulas.

Formulas with Four Requirements

The **VLOOKUP()** formula has **four** requirements.

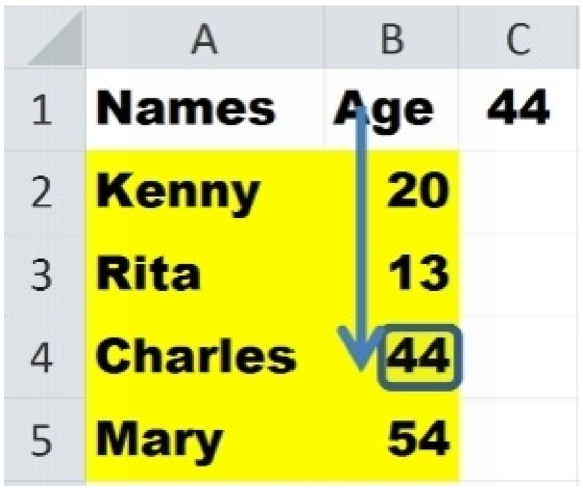


	A	B	C	D
1	Names	Age	44	Charles
2	Kenny	20		
3	Rita	13		
4	Charles	44		
5	Mary	54		

Figure 1-11: Cell **D1** contains a **VLOOKUP()** formula.

Using **Figure 1-11**, if the formula VLOOKUP**(*D1*,A2:B5,2,false**) is typed in cell **C1**, the result will be **44**. The content of **D1** is being matched to names in the first column of the range **A2:B5**; the **first (1st)** column is “A;” the **second (2nd)** column is “B.” The name in cell **D1** matches cell **A4**, so Microsoft Excel moves to the **second (2nd)** column; thus, the result is **44**. **VLOOKUP** means vertical lookup. The formula’s last requirement, **false,** instructs Microsoft Excel to find an exact [precise, as opposed to approximate] match.

HLOOKUP() has **four** requirements.



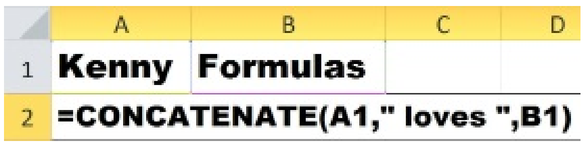
	A	B	C
1	Names	Age	44
2	Kenny	20	
3	Rita	13	
4	Charles	44	
5	Mary	54	

-Figure 1-12: Cell **C1** contains the **HLOOKUP()** formula.

Using **Figure 1-12**, if the formula HLOOKUP**(*"Age"*,A1:D5,**4,**false**) is typed in cell **C1**, the result will be **44**. Within the range **A1:D5**, Microsoft Excel looks for an "**Age**" column heading; that heading is in column B; next, it starts at **B1**, which is the **first (1st)** row in the range; it moves down **three (3)** rows, which makes **four (4)** rows; the contents of cell **B4** is pulled into cell **C1**.

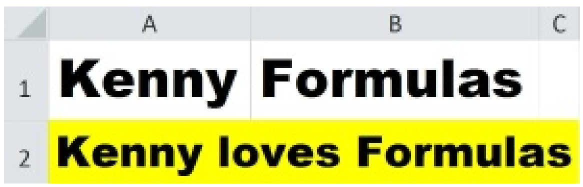
Combining Data

The CONCATENATE() formula is used to combine characters (e.g., i, K, 5, $, !, -, etc…), strings (e.g., "Kenny", "have a nice day", etc…), reference cells, and arrays (e.g., Kenny, Region, i.e., named [defined] cells and ranges).



	A	B	C	D
1	Kenny	Formulas		
2	=CONCATENATE(A1," loves ",B1)			

Figure 1-13: Cell **A2** contains the **CONCATENATE()** formula.



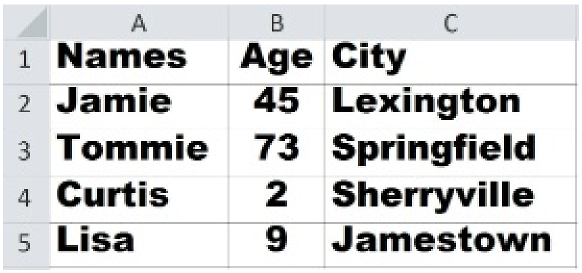
	A	B	C
1	Kenny	Formulas	
2	Kenny loves Formulas		

Figure 1-14: Cell **A2** contains the **CONCATENATE()** formula; this shows the result of the formula.

There are two ways to combine data: you can use the **ampersand** (**&**) symbol, i.e., the "**and**" sign;" the other method is the concatenate formula.

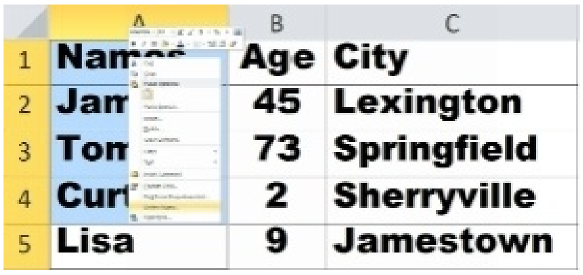
Defining and Naming Cells and Ranges

It's a good idea to name [define] cells and ranges, because it eliminates repetitive typing. **Figures 1-17** through **1-23** provides instructions on naming cell content.



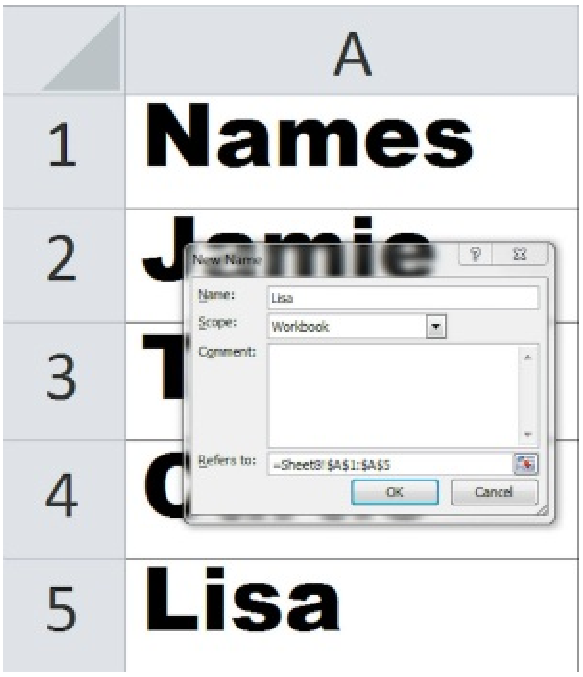
	A	B	C
1	Names	Age	City
2	Jamie	45	Lexington
3	Tommie	73	Springfield
4	Curtis	2	Sherryville
5	Lisa	9	Jamestown

Figure 1-17: Cell range **A1:A5** has been selected.



	A	B	C
1	Nam	Age	City
2	Jam	45	Lexington
3	Tom	73	Springfield
4	Curt	2	Sherryville
5	Lisa	9	Jamestown

Figure 1-18: Under the **Formulas** tab, you'll find the **Define Name** feature.



	A
1	Names
2	Jamie
3	T
4	C
5	Lisa

New Name

Name: Lisa

Scope: Workbook

Comment:

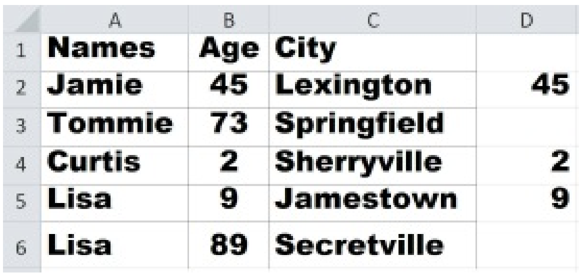
Refers to: =Sheet8!$A$1:$A$5

Figure 1-19: "**Names**" typed in the **Names** field, within the **Define Name** box; the **OK** button is pressed to complete the procedure.

Select the range **A2:B5** and name it "**range,**" because we'll be using it in a formula later.

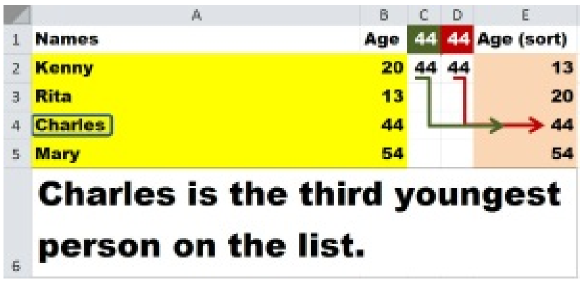
Arrays and Mega-formulas

Using **Figure 1-24**, cell range **F2:F6** is named a**ge**; therefore, the range name will be used in future formulas. After entering the formula IF(age<70,age,””), Cntrl+Shift+Enter buttons are pressed simultaneously to execute the array [a single formula that applies to multiple cells in a range]; it’s the same as dragging a formula from one cell to another.



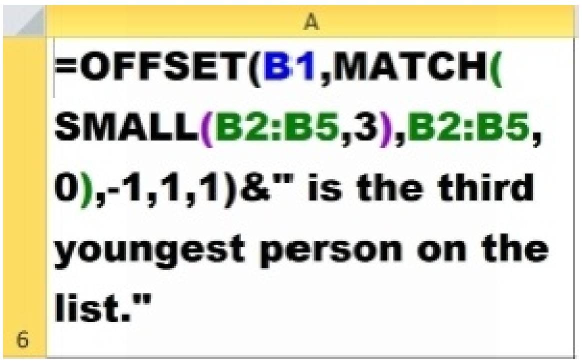
	A	B	C	D
1	Names	Age	City	
2	Jamie	45	Lexington	45
3	Tommie	73	Springfield	
4	Curtis	2	Sherryville	2
5	Lisa	9	Jamestown	9
6	Lisa	89	Secretville	

Figure 1-24: Cell range **B2:B6** is named “**Age**.”



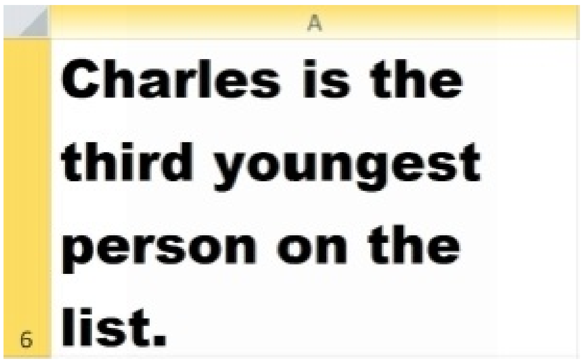
	A	B	C	D	E
1	Names	Age	44	44	Age (sort)
2	Kenny	20	44	44	13
3	Rita	13			20
4	Charles	44			44
5	Mary	54			54
6	Charles is the third youngest person on the list.				

Figure 1-25: A mega formula that consists of three individual formulas, and one text string at the end of it.



```
=OFFSET(B1,MATCH(SMALL(B2:B5,3),B2:B5,0),-1,1,1)&" is the third youngest person on the list."
```

Figure 1-25.1: A mega formula that consists of three individual formulas; this is a closer look at the formula.



Charles is the third youngest person on the list.

Figure 1-25.2: This is the end result of the formula in **Figures 1-25** and **1-25.1**.

Figure 1-25, cell **A6** contains this mega formula [does the work of several intermediate formulas]:

OFFSET(**B1,*MATCH(*SMALL(B2:B5,3),*B2:B5,0)*,**-1,1,1)&**" is the third youngest person on the list."**

Charles is the third youngest person on this list is the result of the mega formula in cell **A6**.

The mega formula is color coded for easy viewing, while it is processed [put through the steps, in order to perform operations on data].

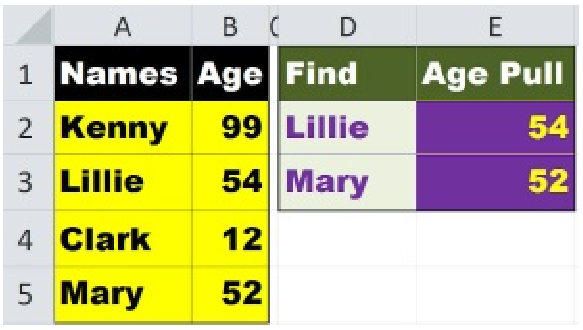
OFFSET(**B1,*MATCH(*SMALL(B2:B5,3)*,B2:B5,0)***,-1,1,1)&**" is the third youngest person on the list."**

OFFSET(**B1,*MATCH(*44)*,B2:B5,0***),-1,1,1)& **" is the third youngest person on the list."**

OFFSET(**B1,*3***,-1,1,1)& **" is the third youngest person on the list."**

Using **Figure 1-25**, Microsoft Excel locates cell **B1**; it counts **three (3)** rows down; the first row is cell **B2**; therefore, cell **B4** is row **3**; next, it moves **one (1)** column leftward; in cell **A4**, the name **Charles** is the result of the mega formula; the last action is combining **Charles** and the **text string**.
IF(**ISERROR(*VLOOKUP(names,range,2,FALSE)*)**,"",***VLOOKUP(names,range,2,FALS***

This is a combined mega formula, and it's an array [requires pressing Cntrl+Shift+Enter simultaneously]:



	A	B	C	D	E
1	Names	Age		Find	Age Pull
2	Kenny	99		Lillie	54
3	Lillie	54		Mary	52
4	Clark	12			
5	Mary	52			

Figure 1-26: Cells **A2:B5** has been named "**range**," and cells **D2:D3** has been named "**names**."

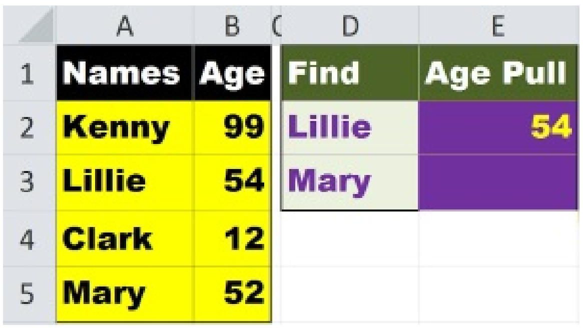
Using **Figure 1-26**, cell range **E2:E3**, this formula is both a mega formula and an array:

IF(**ISERROR(*VLOOKUP(names,range,2,FALSE)*)**,"",***VLOOKUP(names,range,2,FALS***
this formula has two results: **54** for cell **E2**, and **52** for cell **E3**.

The **AND()** formula demands that all requirements are met before the formula can lead to TRUE; otherwise, it will result in FALSE, and the **IF()** formula will have to execute the FALSE result. If all **AND()** formula requirements are met, the **IF()** formula will execute the TRUE result. The **AND()** formula can have anywhere from two or more requirements; it, along with the **OR()** formula, can have more requirements than any other formula.

Using **Figure 1-27**, cell range **F1:F2**, Lillie is 54 years old; that is greater than 53, and it is less than 70; thus, the **VLOOKUP()** formula result is **54**; however, cell F3 is FALSE, because both **AND()** formula requirements are not TRUE; therefore, the FALSE result leaves cell **F3** blank, because two empty quotation marks(””) will result in a blank cell. This is the mega formula:

IF(AND(VLOOKUP(D2,range,2,FALSE)>53,VLOOKUP(D2,range,2,FALSE)<70),VLOOKUP(D2,range,2,FALSE),””)



	A	B	C	D	E
1	Names	Age		Find	Age Pull
2	Kenny	99		Lillie	54
3	Lillie	54		Mary	
4	Clark	12			
5	Mary	52			

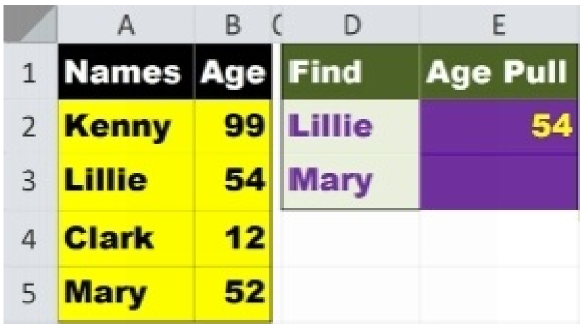
Figure 1-27: Cells **A2:B5** has been named “**range**,” and cells **D2:D3** has been named “**names**.”

Using **Figure 1-27**, cell **F2** and cell **F3**, the formula is typed into cell **F2**; it is dragged down to **F3**. This formula isn’t an array, because it won’t work properly. Once an individual becomes familiar with formulas, he or she will automatically know when to enter a formula as an array, and when it should be entered as a regular formula.

The **OR()** formula demands that one of its requirements are met before the formula can lead to TRUE; if none of its requirements are met, it will result in FALSE, and the **IF()** formula will execute the FALSE result. If one of the **OR()** formula requirements are met, the **IF()** formula will execute the TRUE result. **The OR()** formula can have anywhere from **two** or more requirements; it, along with the **AND()** formula, can have more requirements than any other formula.

Using **Figure 1-28**, cell **D2**, Lillie is 54 years old; that is greater than 53, but it’s not less than 50; however, **one** requirement has been met; therefore, the TRUE result, in cell **F2**, is executed. In cell **F3**, because neither of the **OR()** formula’s requirements are met, the FALSE result will result in nothing [empty **quotation marks** (**””**) equals an empty cell, e.g., ””] being displayed. This formula is typed in cell **F2**, and it is dragged down to cell **F3**:

IF(OR(VLOOKUP(D3,range,2,FALSE)>53,VLOOKUP(D3,range,2,FALSE)<50),VLOOKUP(D3,range,2,FALSE),””)



	A	B	C	D	E
1	Names	Age		Find	Age Pull
2	Kenny	99		Lillie	54
3	Lillie	54		Mary	
4	Clark	12			
5	Mary	52			

Figure 1-28: Cells **A2:B5** has been named “**range**.”

Author