



Introduction to Microsoft Excel

Using Microsoft Excel

Introduction

Microsoft Excel is a spreadsheet application used to create lists, perform calculations, and analyze numbers. It can be used in business, economics, or accounting, etc.

While the default features of Microsoft Excel should be enough in most scenarios, in some cases you will want more complex functionality to perform advanced operations. To make this possible, Microsoft Excel is accompanied by Microsoft Visual Basic, a programming environment that allows you to use the Visual Basic language to enhance the usefulness and functionality of a spreadsheet.

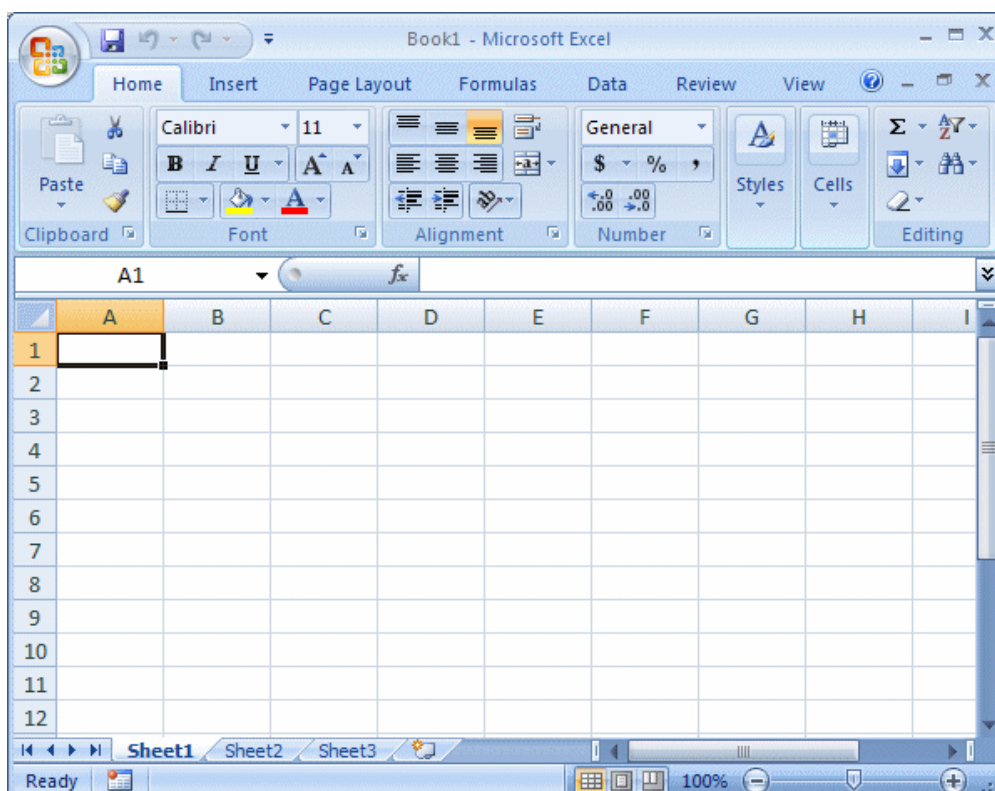
Introduction to Microsoft Excel

To use Microsoft Excel, you can launch like any regular Microsoft Windows application. You can click Start -> (All) Programs -> Microsoft Office -> Microsoft Office Excel 2007. If you have a Microsoft Excel document in Windows Explorer, in My Documents, or in an email, etc, you can double-click it. This would also start Microsoft Excel and would open the document.

The classic way users launch Microsoft Excel is from the Start menu on the task bar. You can also start the application from a shortcut on the desktop. There are many ways you can create a shortcut on your desktop. To create a Microsoft Excel shortcut on the desktop, do one of the following:


❖ Practical Learning: Starting Microsoft Excel

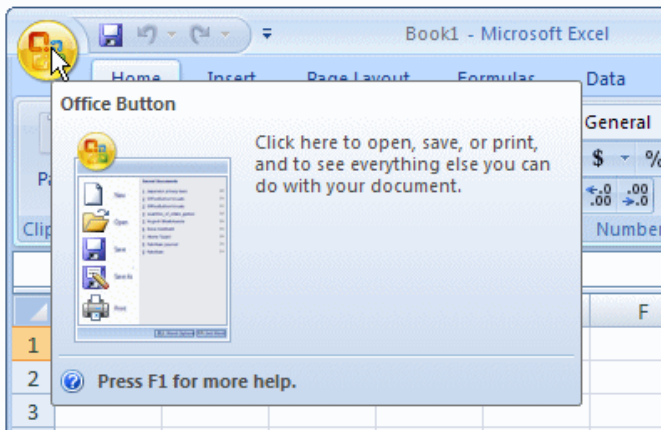
- To start Microsoft Excel, from the Taskbar, click Start -> (All) Programs -> Microsoft Office -> Microsoft Office Excel



The Office Button

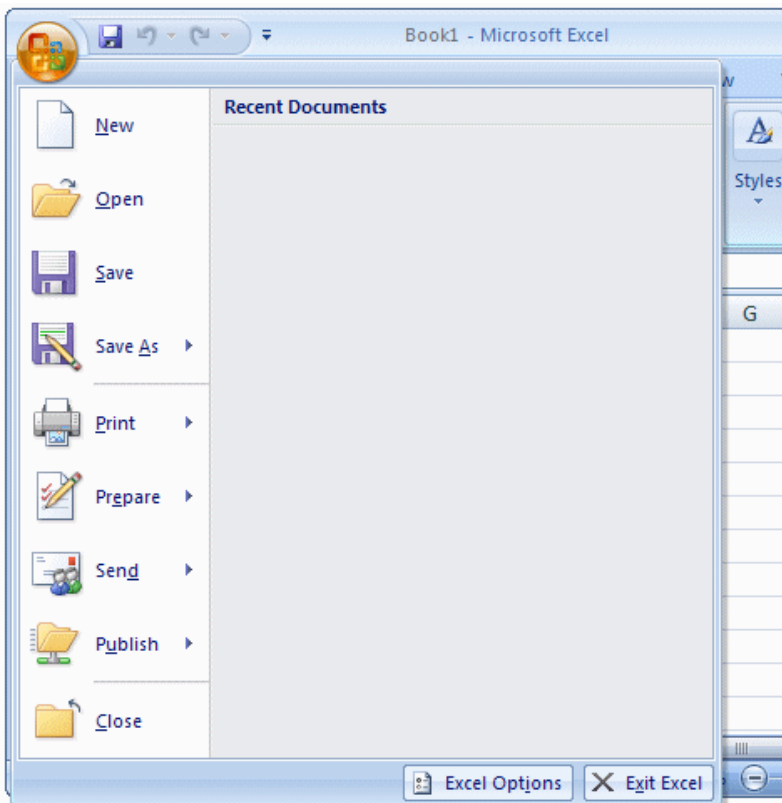
When Microsoft Excel opens, it displays an interface divided in various sections. The top section

displays the title bar which starts on the left side with the Office Button . If you position the mouse on it, a tool tip would appear:

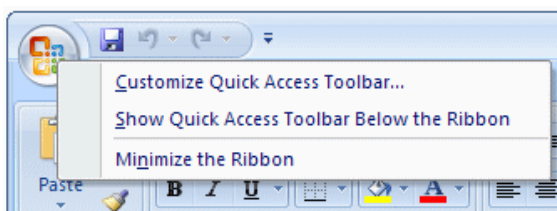


The Options of the Office Button

When clicked, the Office Button displays a menu:




As you can see, the menu of the Office Button allows you to perform the routine Windows operations of a regular application, including creating a new document, opening an existing file, or saving a document, etc. If you right-click the office button, you would get a short menu:

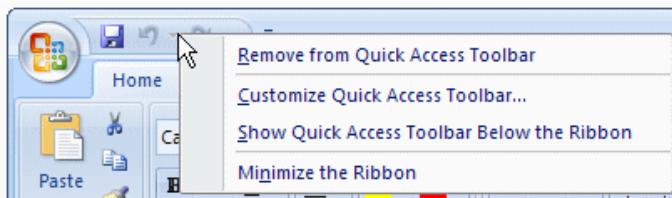


We will come back to the options on this menu.

The Quick Access Toolbar

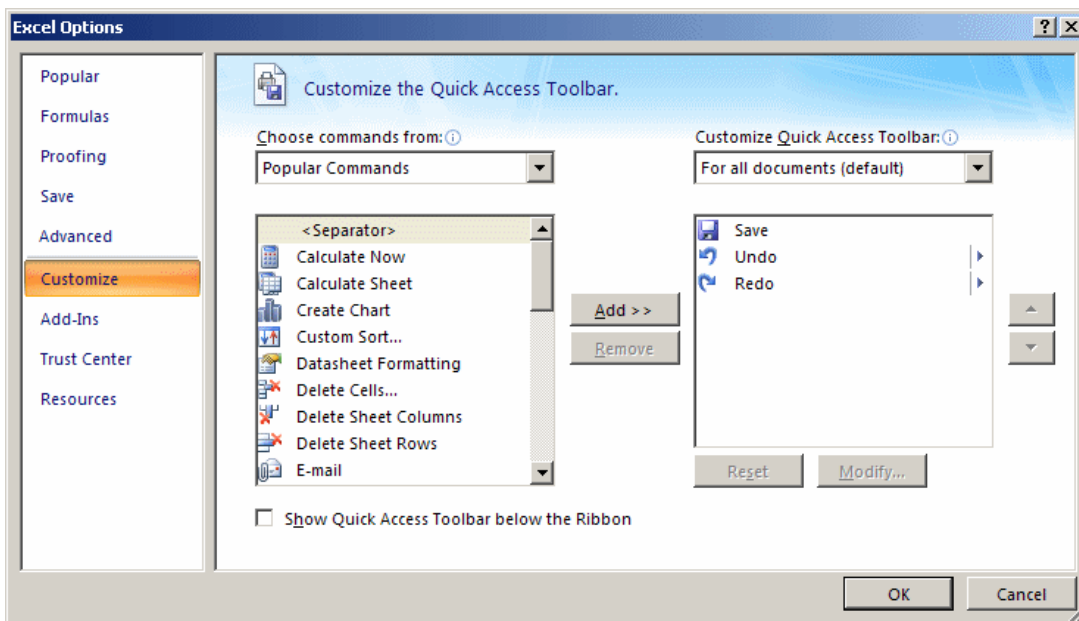
Introduction

The Quick Access Toolbar  is on the right side of the Office Button. It displays a



To hide the Quick Access toolbar, you can right-click it and click Remove Quick Access Toolbar. If you position the mouse on a button, a tool tip would appear.

In the beginning, the Quick Access toolbar displays only three buttons: Save, Undo, and Redo. If you want more buttons than that, you can right-click the Quick Access toolbar and click Customize Quick Access Toolbar... This would display the Excel Options dialog box:

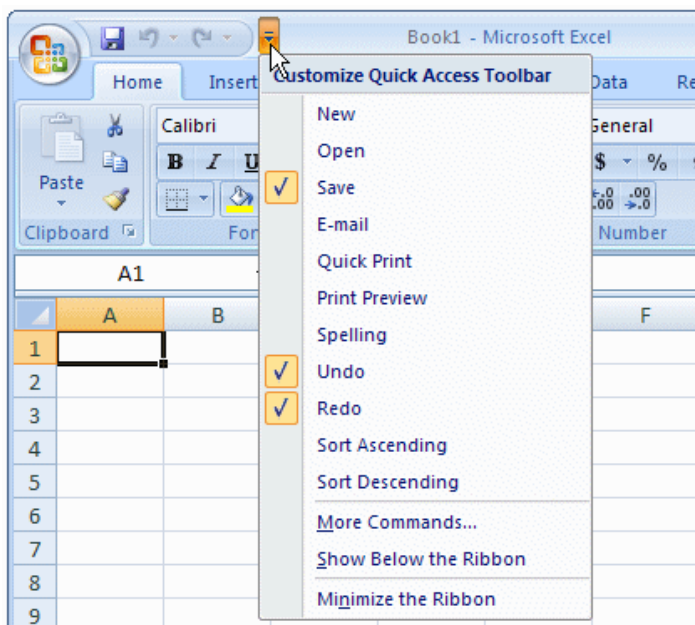


To add a button to the Quick Access toolbar, on the left list of Add, click an option and click Add. After making the selections, click OK.

To remove a button from the Quick Access toolbar, right-click it on the Quick Access toolbar and click Remove From Quick Access Toolbar.

The Quick Access Button

A button with a down-pointing arrow displays on the right side of the Quick Access toolbar. You can click or right-click that button to display its menu:



The role of this button is to manage some aspects of the top section of the Microsoft Excel interface, including deciding what buttons to display on the Quick Access toolbar. For example, instead of using the Customize Quick Access Toolbar menu item as we saw previously, you can click an option from that menu and its corresponding button would be added to the Quick Access

toolbar. If the options on the menu are not enough, you can click either Customize Quick Access Toolbar or More Commands... This would open the Excel Options dialog box.

The main or middle area of the top section displays the name of the application: Microsoft Excel. You can right-click the title bar to display a menu that is managed by the operating system.

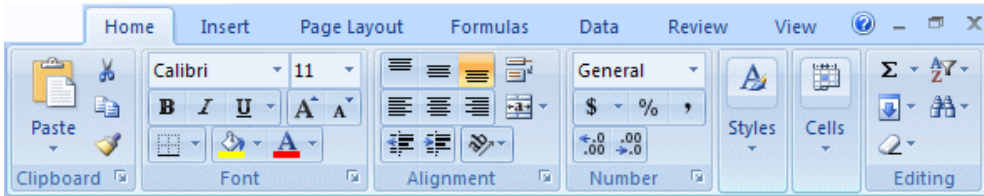
On the right side of the title bar, there are three system buttons that allow you to minimize, maximize, restore, or close Microsoft Access.

Under the title bar, there is another bar with a Help button on the right side.

The Ribbon

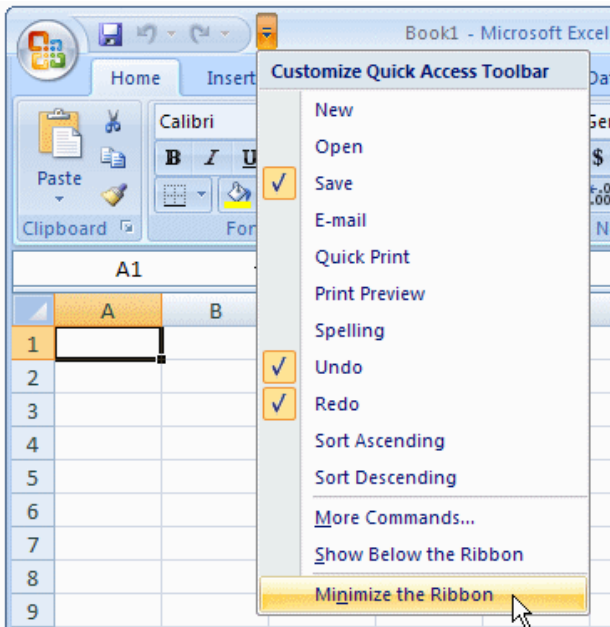
Introduction

Under the title bar, Microsoft Excel displays the Ribbon:

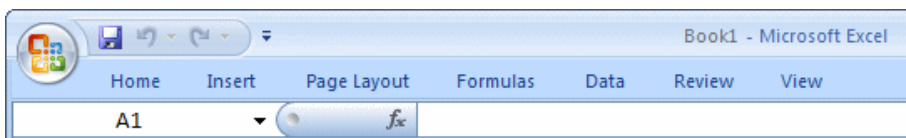


By default, the Ribbon displays completely in the top section of Microsoft Excel under the title bar. One option is to show it the way the main menu appeared in previous versions of Microsoft Excel. To do this:

- Right-click the Office Button, the Quick Access toolbar, or the Ribbon itself, and click Minimize the Ribbon
- Click or right-click the button on the right side of the Quick Access toolbar:



This would display the Ribbon like a main menu:

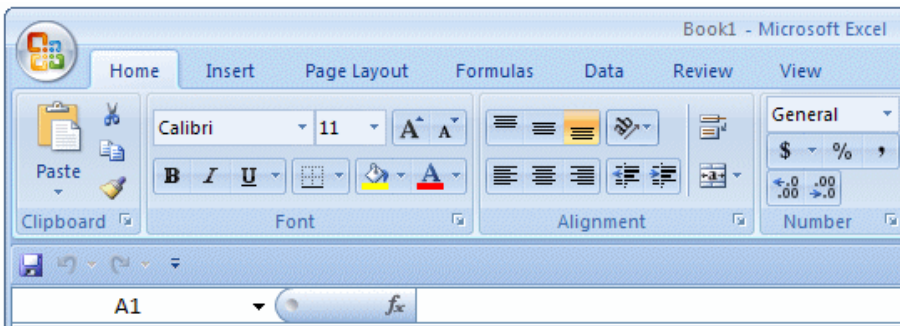


To show the whole Ribbon again:

- Right-click the Office Button, the Quick Access toolbar, or one of the Ribbon menu items, and click Minimize the Ribbon to remove the check mark on it
- Click or right-click the button on the right side of the Quick Access toolbar and click Minimize the Ribbon to remove the check mark on it
- Double-click one of the menu items of the Ribbon

Changing the Location of the Ribbon

By default, the Quick Access toolbar displays on the title bar and the Ribbon displays under it. If you want, you can switch their locations. To do that, right-click the Office Button, the Quick Access toolbar, or the Ribbon, and click Show Quick Access Toolbar Below the Ribbon:

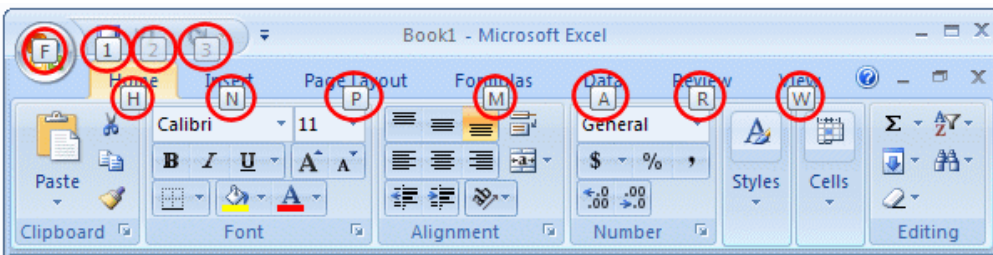


To put them back to the default locations, right-click the Office Button, the Quick Access toolbar, or the Ribbon, and click Show Quick Access Toolbar Above the Ribbon.

The Tabs of the Ribbon

The ribbon is a type of property sheet made of various property pages. Each page is represented with a tab. To access a tab:

- You can click its label or button, such as Home or Create
- You can press ALT or F10. This would display the access key of each tab:




To access a tab, you can press its corresponding letter on the keyboard. For example, when the access keys display, if you press Home, the Home tab would display

- If your mouse has a wheel, you can position the mouse anywhere on the ribbon, and role the wheel (on the mouse). If you role the wheel down, the next tab on the right side would be selected. If you role the wheel up, the previous tab on the left side would be selected. You can keep rolling the wheel until the desired tab is selected

To identify each tab of the Ribbon, we will refer to them by their names.

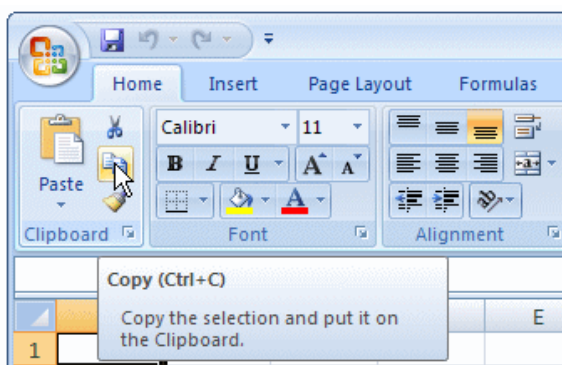
The Sections of a Tab

Each tab of the ribbon is divided in various sections, each delimited by visible borders of vertical lines on the left and right. Each section displays a title in its bottom side. In our lessons, we will refer to each section by that title. For example, if the title displays Font, we will call that section, "The Font Section".

Some sections of the Ribbon display a button . If you see such a button, you can click it. This would open a dialog box or a window.

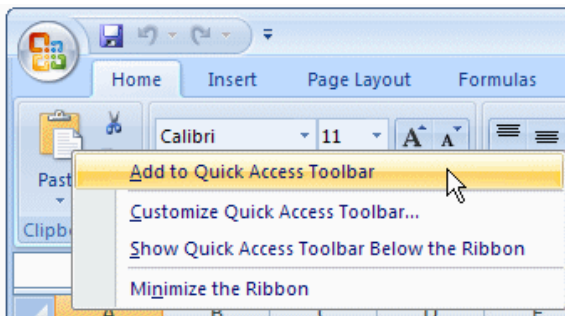
The Buttons of the Ribbon

Since there are various buttons and sometimes they are unpredictable, to know what a particular button is used for, you can position your mouse on it and a tool tip would appear:



You can also use context sensitive help in some cases to get information about an item.

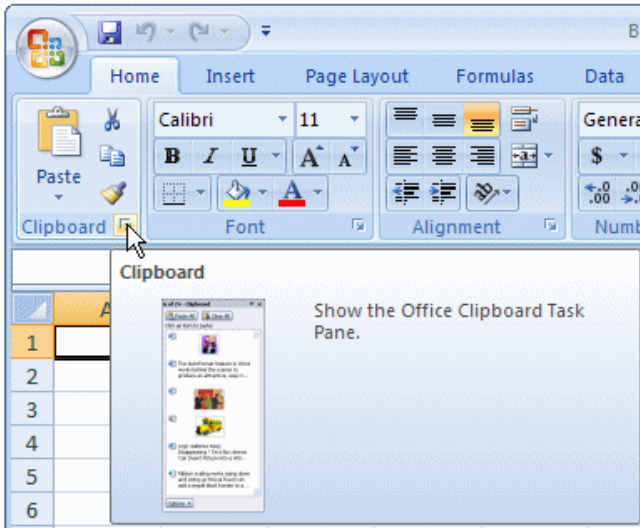
You can add a button from a section of the Ribbon to the Quick Access toolbar. To do that, right-click the button on the Ribbon and click Add to Quick Access Toolbar:



Remember that, to remove a button from the Quick Access toolbar, you can right-click it on the Quick Access toolbar and click Remove From Quick Access Toolbar.

The More Buttons of the Ribbon

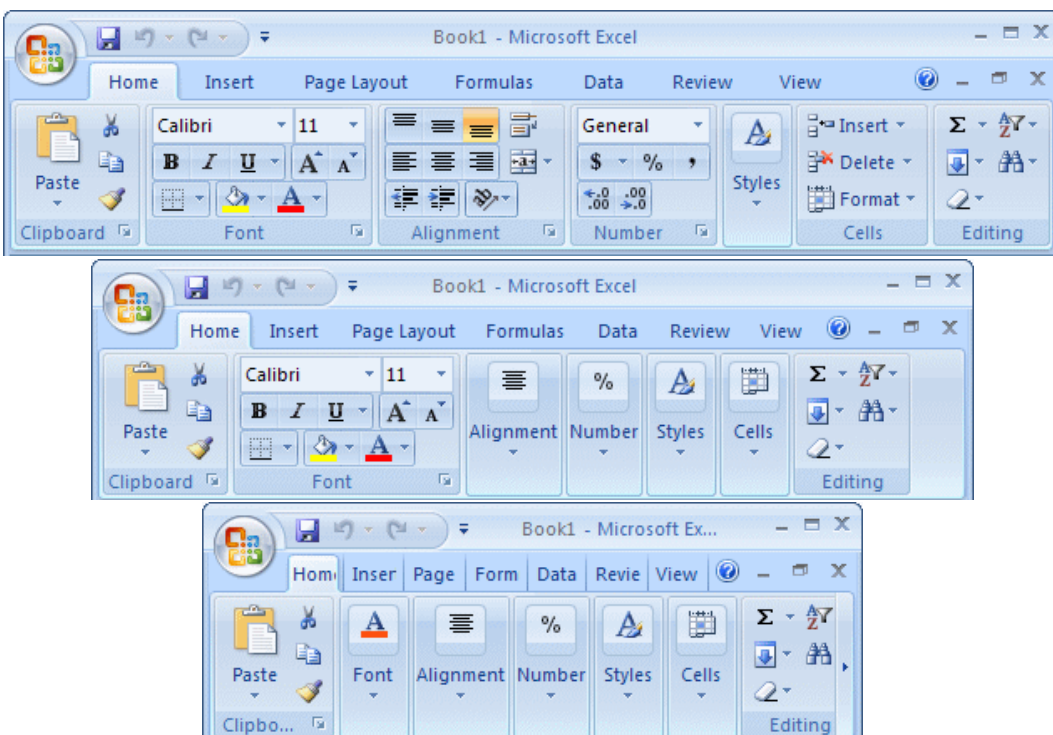
In some sections of the Ribbon, on the lower-right corner, there is a button:



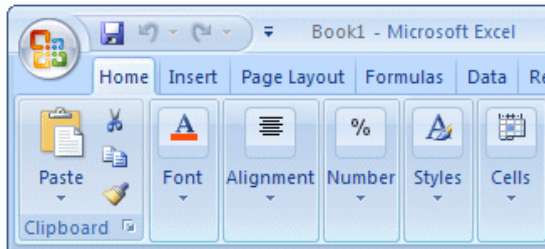
That button is used to display an intermediary dialog box for some actions.

The Size of the Ribbon

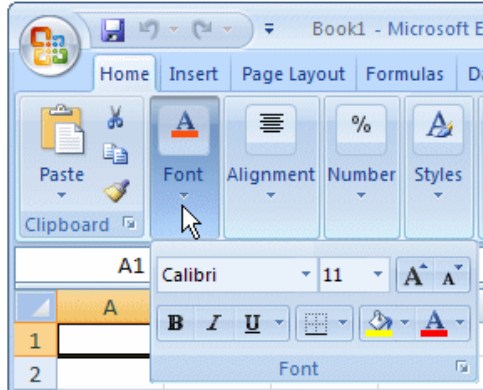
When Microsoft Excel is occupying a big area or the whole area of the monitor, most buttons of the Ribbon appear with text. Sometimes you may need to use only part of the screen. That is, you may need to narrow the Microsoft Excel interface. If you do, some of the buttons may display part of their appearance and some would display only an icon. Consider the difference in the following three screenshots:



In this case, when you need to access an object, you can still click it or click its arrow. If the item is supposed to have many objects, a new window may appear and display those objects:



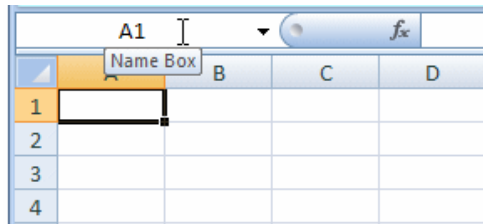
To this:



The Work Area

The Name Box

Under the Ribbon, there is a white box displaying a name like A1 (it may not display A1...), that small box is called the Name Box:

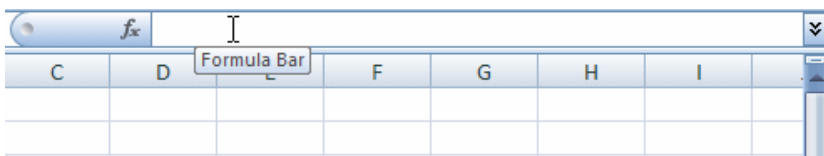


The Insert Function Button

On the right side of the Name box, there is a gray box with an fx button. That fx button is called the Insert Function button.

The Formula Bar

On the right side of the Insert Function button is a long empty white box or section called the Formula Bar:

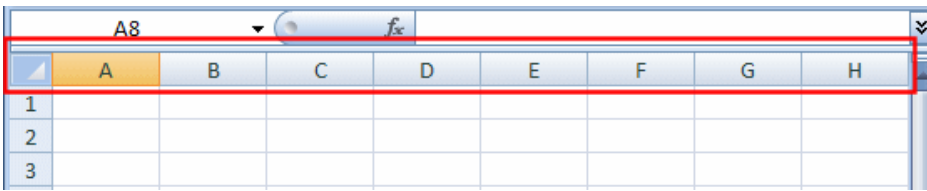


You can hide or show the Formula Bar anytime. To do this, on the Ribbon, click View. In the Show/Hide section:

- To hide the Formula Bar, remove the check mark on the Formula Bar check box
- To show the Formula Bar, check the Formula Bar check box

The Column Headers

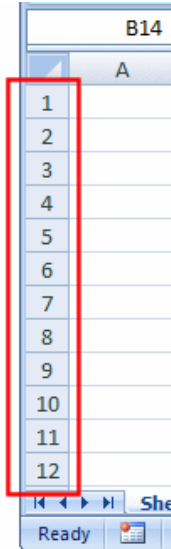
Under the Name Box and the Formula bar, you see the column headers. The columns are labeled A, B, C, etc:



There are 255 of columns.

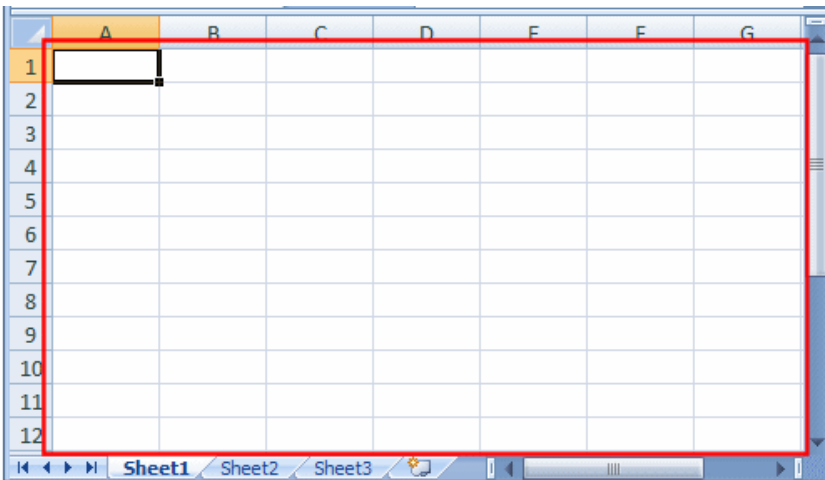
The Row Headers

On the left side of the main window, there are small boxes called row headers. Each row header is labeled with a number, starting at 1 on top, then 2, and so on:



The Cells

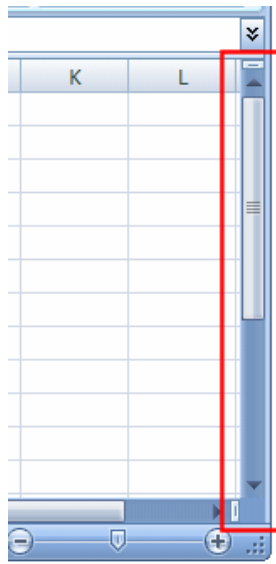
The main area of Microsoft Excel is made of cells. A cell is the intersection of a column and a row:



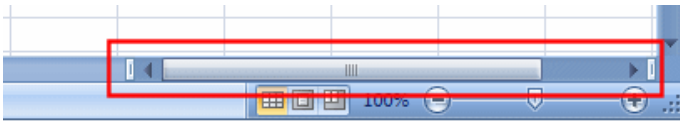
A cell is identified by its name and every cell has a name. By default, Microsoft Excel appends the name of a row to the name of a column to identify a cell. Therefore, the top-left cell is named A1. You can check the name of the cell in the Name Box.

The Scroll Bars

On the right side of the cells area, there is a vertical scroll bar that allows you to scroll up and down in case your document cannot display everything at a time:



In the lower right section of the main window, there is a horizontal scroll bar that allows you to scroll left and right if your worksheet has more items than can be displayed all at once:

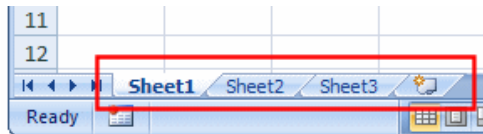


Sometimes the horizontal scroll bar will appear too long or too narrow for you. If you want, you can narrow or enlarge it. To do this, click and drag the button on the left side of the horizontal scroll bar:



The Sheet Tabs

On the left side of the horizontal scrollbar, there are the worksheet tabs:



By default, Microsoft Excel provides three worksheets to start with. You can work with any of them and switch to another at any time by clicking its tab.

The Navigation Buttons

On the left side of the worksheet tabs, there are four navigation buttons:



If you happen to use a lot of worksheets or the worksheet names are using too much space, which would result in some worksheets being hidden under the horizontal scroll bar, you can use the navigation buttons to move from one worksheet to another.

The Status Bar



Under the navigation buttons and the worksheet tabs, the Status Bar provides a lot of information about the job that is going on.

Microsoft Excel File Operations

Saving a File

A Microsoft Excel file gets saved like any traditional Windows file. To save a file:

- You can press Ctrl + S

- On the Quick Access Toolbar, you can click the Save button 
- You can click the Office Button and click Save 

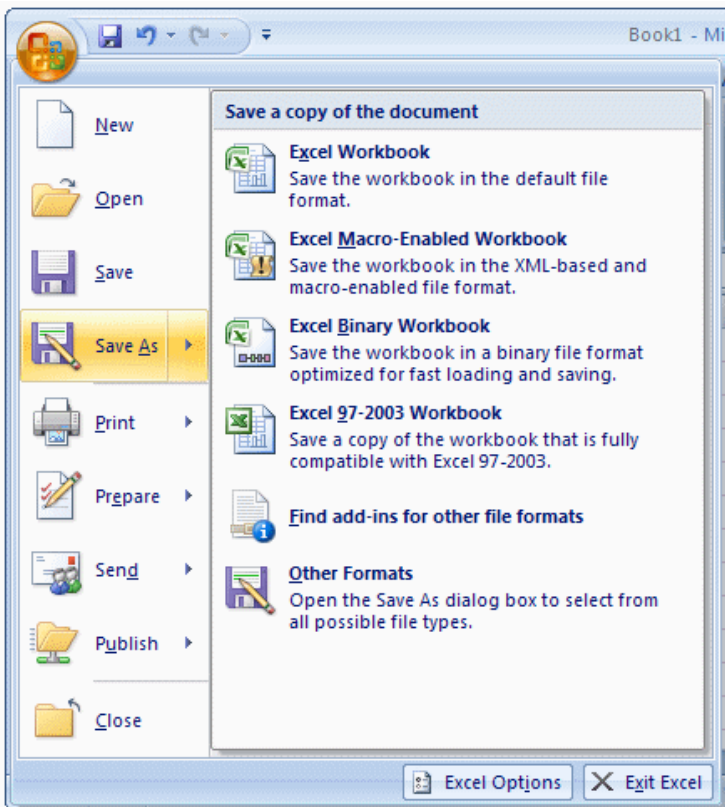
Two issues are important. Whenever you decide to save a file for the first time, you need to provide a file name and a location. The file name helps the computer identify that particular file and register it.


A file name can consist of up to 255 characters, you can include spaces and dashes in a name. Although there are many characters you can use in a name (such as exclamation points, etc), try to avoid fancy names. Give your file a name that is easily recognizable, a little explicit. For example such names as Time Sheets, Employee's Time Sheets, GlobalEX First Invoice are explicit enough. Like any file of the Microsoft Windows operating systems, a Microsoft Excel file has an extension, which is .xls but you don't have to type it in the name.

The second important piece of information you should pay attention to when saving your file is the location. The location is the drive and/or the folder where the file will be saved. By default, Microsoft Excel saves its files in the My Documents folder. You can change that in the Save As dialog box. Just click the arrow of the Save In combo box and select the folder you want.

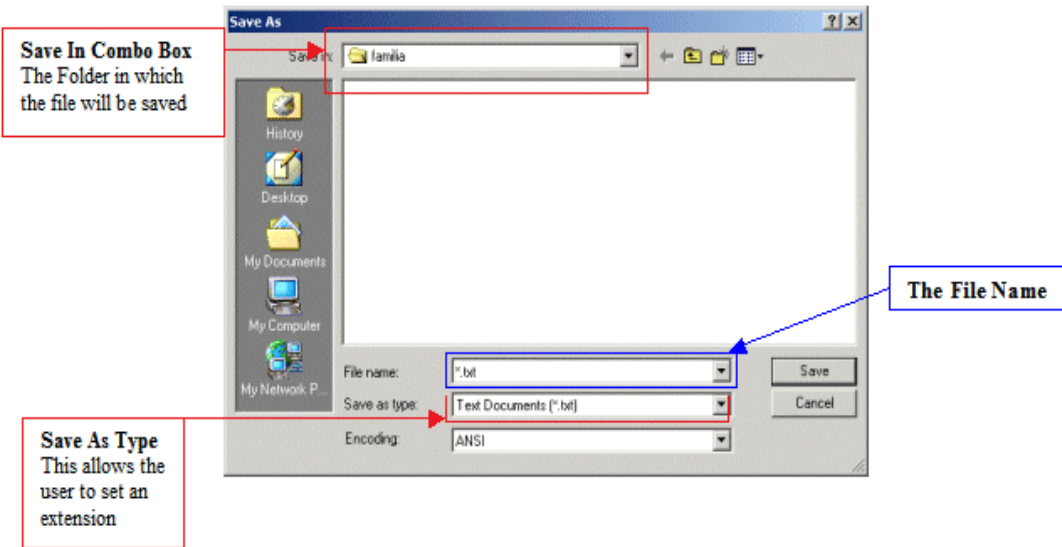
Microsoft Excel allows you to save its files in a type of your choice. To save a file in another format:

- Press F12 or Shift + F12
- You can click the Office Button and position the mouse on Save As and select the desired option:



- On the Quick Access Toolbar, you can click the Save button . Then, in the Save As dialog box, click the arrow of the Save As Type combo box and select a format of your choice

There are other things you can do in the Save As dialog box:



Saving under a Different Name and New Folder

You can save a file under a different name or in another location, this gives you the ability to work on a copy of the file while the original is intact.

There are two primary techniques you can use to get a file in two names or the same file in two locations. When the file is not being used by any application, in Windows Explorer (or in My Computer, or in My Network Places, locate the file, right-click it and choose Copy. To save the file in a different name, right-click in the same folder and choose Paste. The new file will be named Copy Of... You can keep that name or rename the new file with a different name (recommended). To save the file in a different location, right-click in the appropriate folder and click Paste; in this case, the file will keep its name.

In Microsoft Excel, you can use the Save As dialog box to save a file in a different name or save the file with the same name (or a different name) in another folder. The Save As dialog box also allows you to create a new folder while you are saving your file (you can even use this technique to create a folder from the application even if you are not saving it; all you have to do is create the folder, click OK to register the folder, and click Cancel on the Save As dialog box).

Opening a File

The files you use could be created by you or someone else. They could be residing on your computer, on another medium, or on a network. Once one of them is accessible, you can open it in your application.

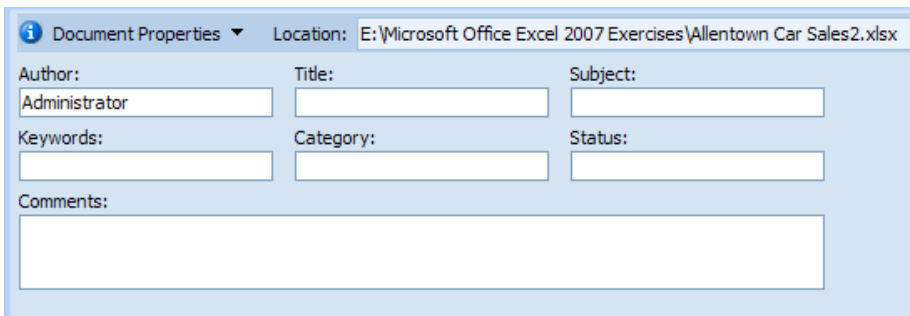
You can open a document either by double-clicking its icon in Windows Explorer, in My Computer, from the Find Files Or Folders window, in My Network Places, or by locating it in the Open dialog box. To access the open dialog box, on the main menu, click File -> Open... You can also click the Open button on the Standard toolbar.

A shortcut to call the Open dialog box is Ctrl + O.

Files Properties

Every file has some characteristics, attributes, and features that make it unique; these are its properties. You can access a file's properties from three main areas on the computer:

- If the file is saved on the desktop and/or it has a shortcut on the desktop, if you open My Computer, Windows Explorer, or the folder (as a window) where the file is stored, right-click the file and click Properties. If the file were saved on the desktop, you would see only some of its properties, the most you can do there is to assign a Read-Only attribute. In My Computer and Windows Explorer, you will be able to change the file's properties. Before opening a file or while in the Open dialog box, you can view some of the file's properties although you won't be able to change them.
- When the file is opened in Microsoft Excel, you can click the Office Button, position the mouse on Prepare, and click Properties. This would display some of the most common attributes of the file:



Document Properties Location: E:\Microsoft Office Excel 2007 Exercises\Allentown Car Sales2.xlsx

Author: Administrator Title: Subject:

Keywords: Category: Status:

Comments:

To change an item, you can click its text box and edit or replace the content. To get more options, you can click the Document Properties button and click Advanced Properties...

A file's properties are used for various reasons. For example, you can find out how much size the file is using, where it is located (the hosting drive and/or folder), who created the file, or who was the last person to access or modify it. The Properties dialog box is also a good place to leave messages to other users of the same file, about anything, whether you work as a team or you simply want to make yourself and other people aware of a particular issue regarding the file.

❖ Practical Learning: Closing Microsoft Excel

- To close Microsoft Excel, click the Office Button and click Exit Excel.
If you are asked whether you want to save the file, click No



Introduction to VBA

Microsoft Visual Basic Fundamentals

Introduction

Microsoft Excel is a spreadsheet application that provides simple to advanced means of creating and managing any type of list. To enhance it beyond its default function, it ships with a language called Microsoft Visual Basic or simply Visual Basic.

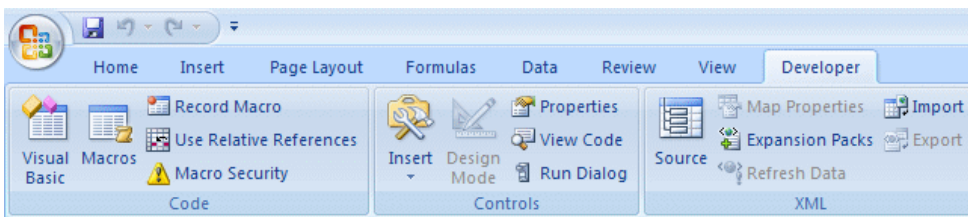
Microsoft Visual Basic for Applications (VBA) is a computer language based on Microsoft Visual Basic. It allows you to write code that can automatically perform actions on a document and/or its content. When using that language, you write pieces of code, using an external environment.

Microsoft Visual Basic is a programming environment that gets automatically installed when you setup Microsoft Excel. It stays apart because most people would not need or use it. This means that, if you want to use the Microsoft Visual Basic programming environment that ships with Microsoft Excel, you must ask for it, which can be easily done.

Launching Microsoft Visual Basic

In our lessons, we will learn how to use both Microsoft Excel and Microsoft Visual Basic to create and manage spreadsheets. The Microsoft Visual Basic programming environment we will use depends on Microsoft Excel. As a result, to use Microsoft Visual Basic, you must first open Microsoft Excel. Then, to write code, you must open Microsoft Visual Basic. There are various ways you can do this, depending on your intention.

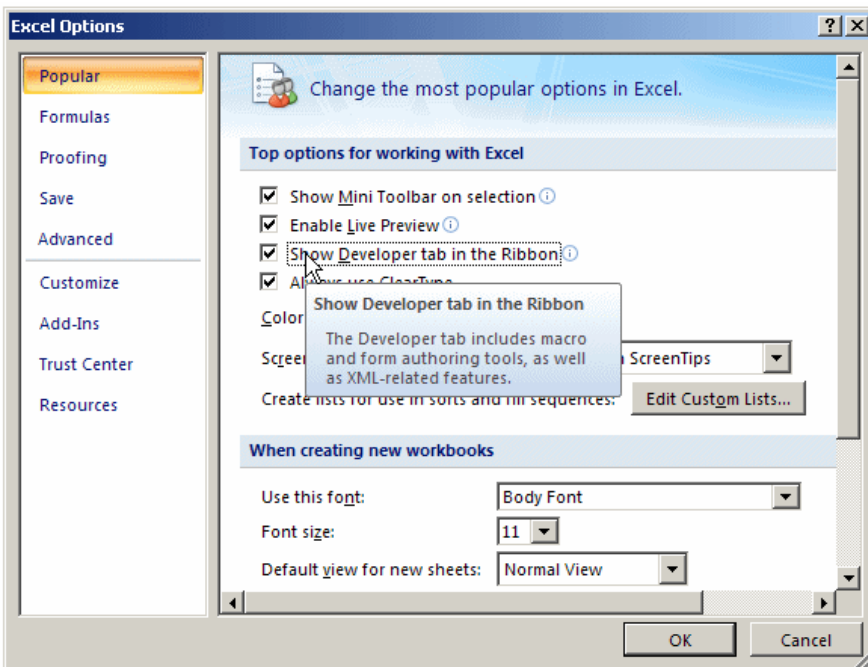
Before using code, you should add a new tab, the Developer tab, to the Ribbon. To do this, you can click the Office Button and click Excel Options. In the Excel Options dialog box, click the Show Developer tab in the Ribbon check box and click OK. The Ribbon would become equipped with a new tab:



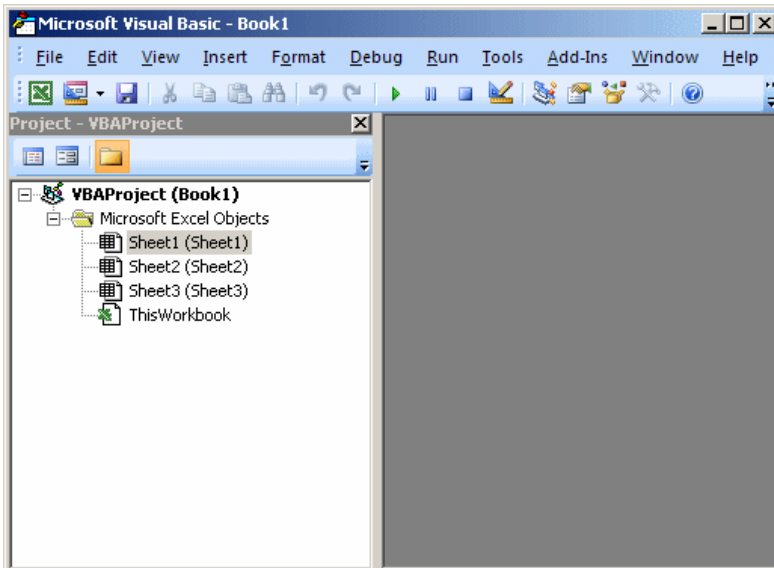
From the Developer tab of the Ribbon, to launch Microsoft Visual Basic, you can click the Visual Basic button.

❖ Practical Learning: Starting Microsoft Visual Basic

1. Start Microsoft Excel
2. Click the Office Button and click Excel Options
3. In the Excel Options dialog box, click the Show Developer tab in the Ribbon check box:



4. Click OK
5. In the Code section of the Developer tab of the Ribbon, to launch Microsoft Visual Basic, click Visual Basic:




The Microsoft Visual Basic Interface

Introduction

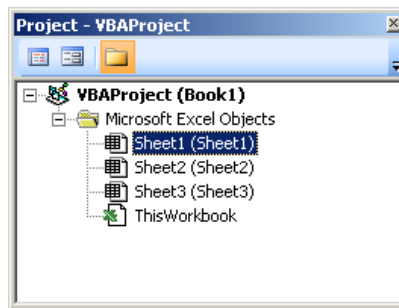
When it opens, like any regular Windows application, Microsoft Visual Basic displays a title bar in the top section. Under the title bar, the application displays a menu, followed by a Standard toolbar.

To assist you with your development, Microsoft Visual Basic can display various windows.

The Project Explorer

The Project Explorer window shows a list of the code segments that are available to your worksheet. It is usually available whenever you open Microsoft Visual Basic. It is usually positioned in the top-left section. If it is not present, to display it, on the main menu of Microsoft Visual Basic, you can click View -> Project Explorer. To close it, you can click its Close button .

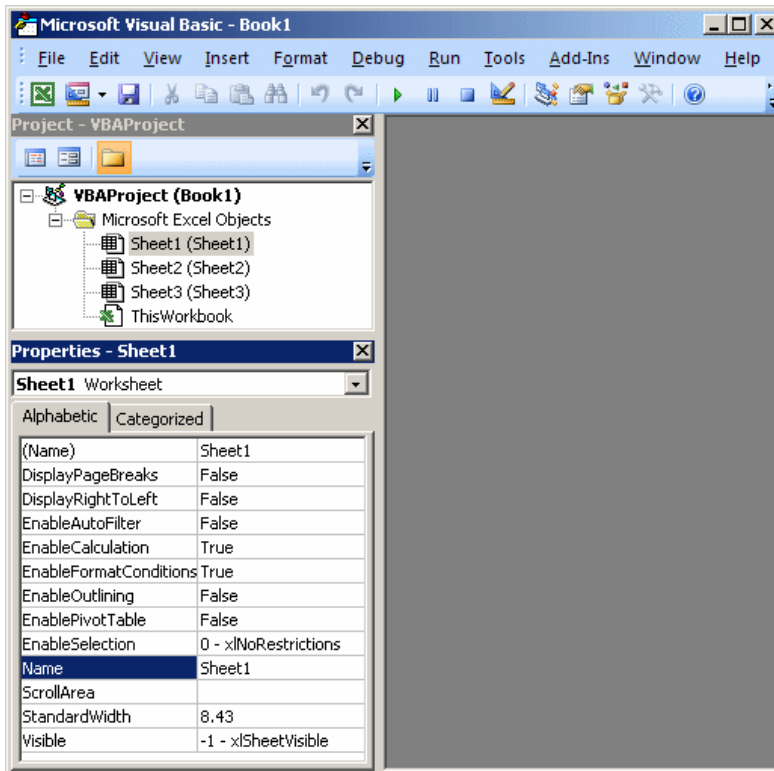
You can move the Project Explorer to another section of the interface. To do this, click its title bar and drag it away from there:



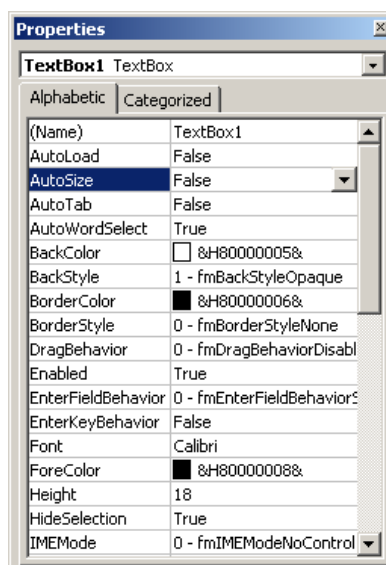
To put the window back where it was previously, you can double-click its title bar.

The Project Explorer

The Properties window is usually positioned in the bottom-left section of the screen. When it does not appear, to display it, on the main menu, click View -> Properties Window:



The Properties Window shows the characteristics of an object that is selected. Like any other window, to move the Properties window from its position, drag its title bar:



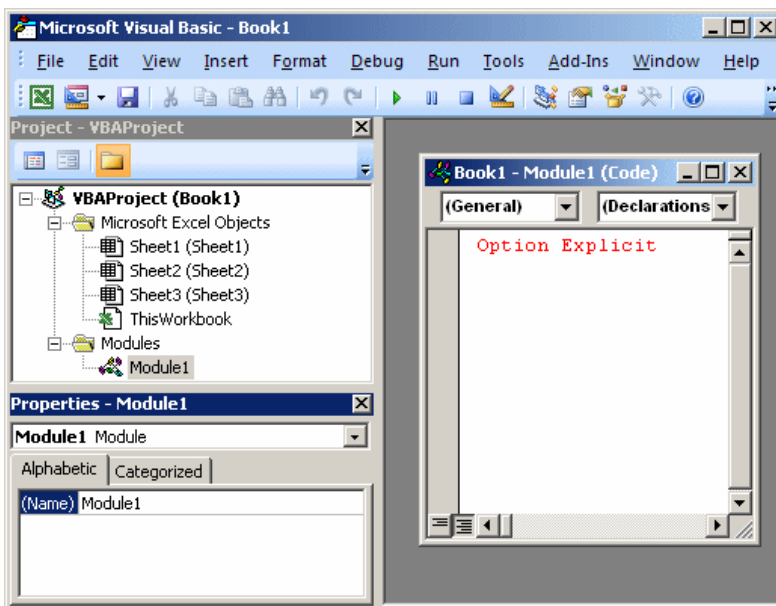
The main area of Microsoft Visual Basic uses a gray background. This area is gray because, in reality, Microsoft Visual Basic is a multiple document interface (MDI) that can be used to display various windows at the same time. At times, this gray area will be occupied with other windows.

Modules

A module is a blank window that resembles a piece of paper on which you write code. When you use Microsoft Excel and work on a document, a default module is automatically allocated for it, whether you use it or not. You can also create a module that is independent of any worksheet.

❖ Practical Learning: Creating a Module

1. On the main menu of Microsoft Visual Basic, click Insert -> Module
2. Notice that a blank window with a blinking caret appears

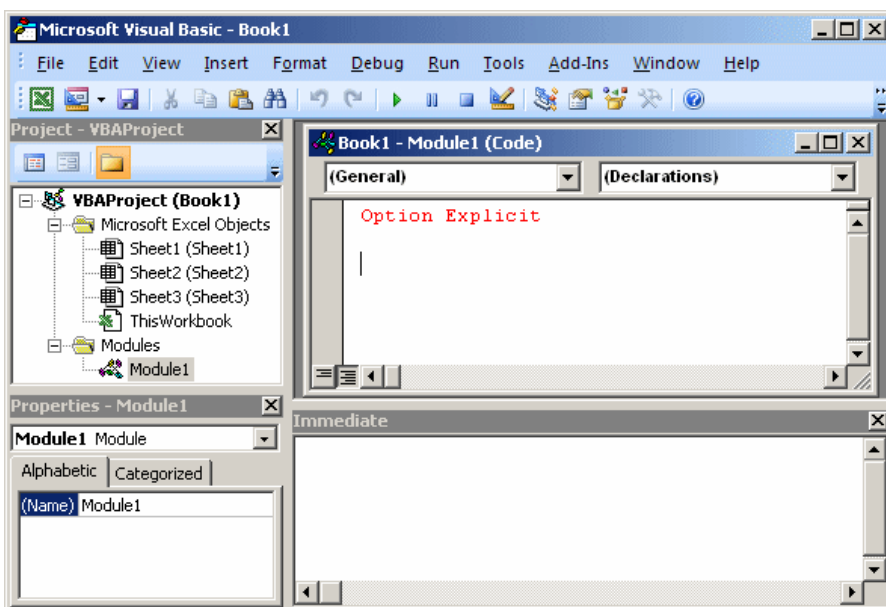


The Immediate Window

To help you test code, Microsoft Visual Basic provides a special window called the Immediate Window. To display it, on the main menu of Microsoft Visual Basic, you can click View -> Immediate Window.

❖ Practical Learning: Displaying the Immediate Window

1. To display the Immediate Window, on the main menu of Visual Basic, click View -> Immediate Window
2. Notice that a window with an Immediate title bar appears at the bottom with a blinking caret



3. To return to Microsoft Excel, on the Standard toolbar of Visual Basic, click the View Microsoft Excel button
4. To close Microsoft Visual Basic, on the main menu, click File -> Close and Return to Microsoft Excel

VBA in Visual Basic

Introduction

In the spreadsheet you will create, you use Microsoft Excel to create normal documents using the default settings of the application. To apply some advanced features to a spreadsheet, you can use Microsoft Visual Basic that is automatically installed with Microsoft Excel.

To create a spreadsheet with functionality beyond the defaults, you write code. Microsoft Visual

Basic is a programming environment that uses a computer language. That language is called Visual Basic for Applications (VBA). Although VBA is a language of its own, it is in reality derived from the big Visual Basic computer language developed by Microsoft. In our lessons, we will learn how to use VBA in Microsoft Excel.

To take advantage of the functionalities of the Microsoft Visual Basic environment, there are many suggestions you can use or should follow. Because VBA is normal computer language, there are various rules you must follow for the language to work.

Using VBA

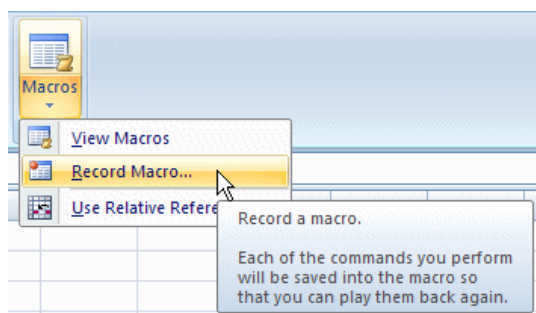
In our lessons, we will use the word VBA sometimes but most of the time, we use the expression "Visual Basic Language". When we use "Visual Basic language", we refer to a concept that is recognized by all child languages of Visual Basic, including VBScript and VBA. When we will use the word VBA, we refer to a concept that either is proper to VBA as a language and is not necessarily applied to some other flavors of Visual Basic, or to the way the Visual Basic language is used in Microsoft Excel. For example, the word **String** is used in all Visual Basic languages but the word **VARIANT** is not used in the 2008 version of the Visual Basic language.

Macros

Creating a Macro

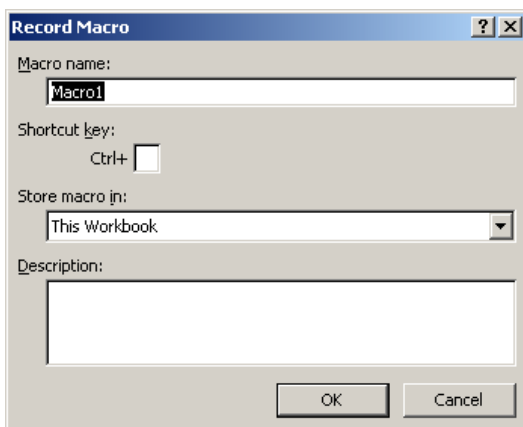
To launch Microsoft Visual Basic using the default installation of Microsoft Excel and launching from a macro:

- On the [Ribbon](#), you can click View. In the Macros section, click the arrow under the Macros button and click Record a Macro:



- Click Developer. In the Code section, click the Record Macro button 

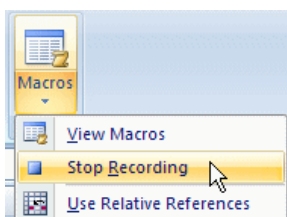
In each case, the Record Macro dialog box would come up:



On the Record Macro dialog box, accept or enter a name for the macro. As an option, you can type a description of the macro in the bottom text box. Once you are ready, click OK. This would bring you to the document in Microsoft Excel where you can do what you want.

After doing what is necessary, to end the creation of the macro, on the Ribbon:

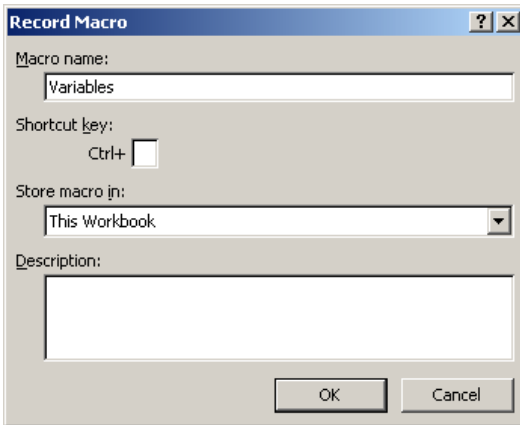
- Click View. In the Macros section, click the the arrow of the Macros button and click Stop Recording:




- Click Developer. In the Code section, click the Stop Recording button 

❖ Practical Learning: Creating a Macro

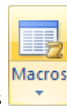
1. Start Microsoft Excel
2. On the Ribbon, click Developer.
In the Code section, click Record Macro
3. Set the Name of the macro as **Variables**




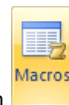
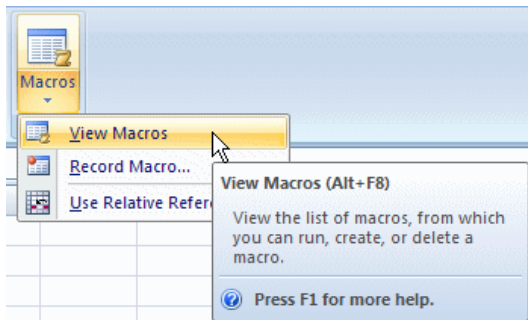
4. Click OK
5. In the document, whatever box is selected (don't click any), type =2
6. On the Formula Bar, click the Enter button 
7. In the Code section of the Ribbon, click Stop Recording

The Skeleton Code of a Macro

When you create a macro, skeleton code is generated for you. To access the code generated for a macro, on the Ribbon:

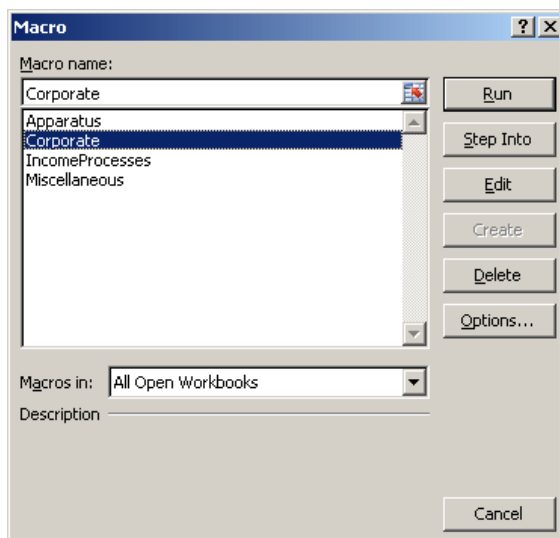


- Click View. In the Macros section, click Macros  or click the arrow of the Macros button and click View Macros



- Click Developer. In the Code section, click the Macros button

Any of these actions would open the Macros dialog box that would display the list of macros in the current document:



To see the code of a macro, click its name and click Edit.

1. To open Microsoft Visual Basic, in the Code section of the Ribbon, click Macros
2. In the Macros dialog box, make sure Exercise1 is selected and click Edit

VBA in a Macro

We will try to reduce as much as possible the code that will be written for you. Still, there are some lines and words we will keep or use but will ignore them for now. As we move on in our lessons, you will understand what everyone of those words means. The code generated in the above Practical Learning section was:

```
Sub Exercise()  
    ActiveCell.FormulaR1C1 = "=2"  
End Sub
```

The first line of code has the word **Sub**. We will introduce it later on. Exercise1 is the name of the macro we created. We will come back to names in a few sections in this lesson. We will also come back to the role of parentheses. The section of code ends with the **End Sub** line. We will come back to it when we study the procedures. For now, consider the **Sub** Exercise1() and **End Sub** lines as the minimum requirements we need as this time, that we don't need to be concerned with, but whose roles we can simply ignore at this time.

The most important line of our code, and the only line we are concerned with, is:

```
ActiveCell.FormulaR1C1 = "=2"
```

This line has three main sections: **ActiveCell.FormulaR1C1**, **=**, and **"=2"**. For now, understand that the **ActiveCell.FormulaR1C1** expression means "whatever box is selected in the document".

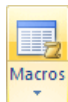
The **=** sign is called the assignment operator. As its name indicates, the assignment operator is used to assign something to another, to give a value to something, or more precisely to store something somewhere.

The thing on the right side of **=** is called a value. Therefore, **"=2"** is a value. Based on this, the expression **ActiveCell.FormulaR1C1 = "=2"** means "Assign the thing on the right side of **=** to the thing on the left side of **=**." Another way to put it is, "Store the value on the right side of the assignment operator to the selected box on the left side of the assignment operator." For now, until indicated otherwise, consider that that's what that line of code means.

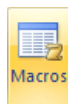
Using a Macro

After creating a macro, you can use it to see its result. This is also referred to as executing a macro or running a macro.

To execute a macro, on the Ribbon:



- Click View. In the Macros section, click Macros or click the the arrow of the Macros button and click View Macros



- Click Developer. In the Code section, click the Macros button

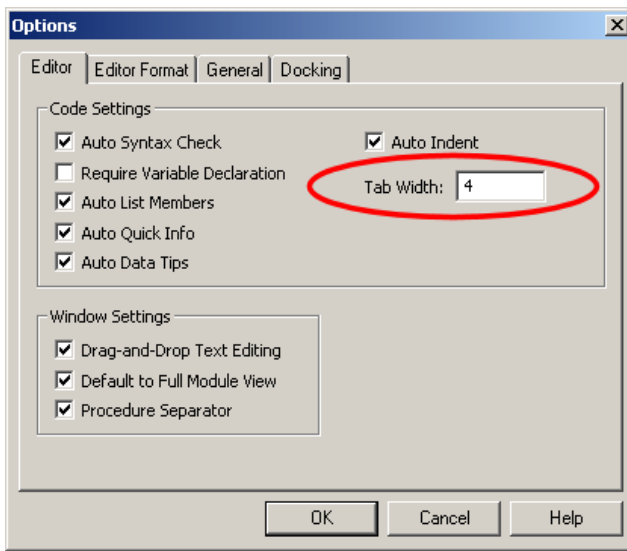
In the Macro dialog box, click the name of the macro and click Run.

Writing Code

Code Indentation

Indentation is a technique that allows you to write easily readable code. It consists of visually showing the beginning and end of a section of code. Indentation consists of moving code to the right side.

The easiest and most common way to apply indentation consists of pressing Tab before typing your code. By default, one indentation, done when pressing Tab, corresponds to 4 characters. This can be automatically set using the Tab Width text box of the Editor property page in the Options dialog box. To change it, on the main menu of Microsoft Visual Basic, you can click Tools -> Options and click the Editor tab:



If you don't want the pressing of Tab to be equivalent to 4 characters, change the value of the Tab Width text box to a reasonable value and click OK. Otherwise, it is (strongly) suggested that you keep to its default of 4 characters.

Comments

A comment is a piece of text in code that would not be considered when reading your code. As such, a comment can be written any way you want.

In the Visual Basic language, the line that contains a comment can start with a single quote. Here is an example:

This line will not be considered as part of the code

Alternatively, you can start a comment with the **Rem** keyword. Anything on the right side of **rem**, **Rem**, or **REM** would not be read. Here is an example:

```
' This line will not be considered as part of the code
Rem I can write anything I want on this line
```

Comments are very useful and you are strongly suggested to use them regularly.

The code that was generated in our Practical Learning section contains a few lines of comment:

```
Sub Exercisel()
'
' Exercisel Macro
'
'
ActiveCell.FormulaR1C1 = "=2"
End Sub
```

❖ Practical Learning: Closing Microsoft Excel

1. To close Microsoft Visual Basic, on the main menu, click File -> Close and Return to Microsoft Excel
2. To close Microsoft Excel, click the Office Button and click Exit Excel.
If you are asked whether you want to save the file, click No



Variables and Data Types

Variables

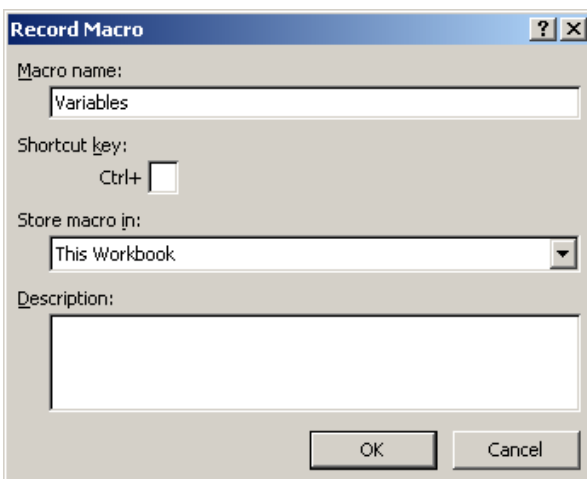
Introduction

To use some values in code, you must first create them. The computer memory is made of small storage areas used to hold the values of your application. When you use a value in your code, the computer puts it in a storage area. When you need it, you let the computer know. The machine "picks it up", brings it to you, and then you can use it as you see fit.

In the world of computer programming, a variable is a value you ask the computer to temporarily store in its memory while the program is running.

❖ Practical Learning: Creating a Macro

1. Start Microsoft Excel
2. On the Ribbon, click Developer.
In the Code section, click Record Macro
3. Set the Name of the macro as **Variables**



4. Click OK
5. In the document, whatever box is selected (don't click any), type =2
6. On the Formula Bar, click the Enter button
7. In the Code section of the Ribbon, click Stop Recording
8. To open Microsoft Visual Basic, in the Code section of the [Ribbon](#), click Macros
9. In the Macros dialog box, make sure Variables is selected and click Edit

Declaring a Variable

When writing your code, you can use any variable just by specifying its name. When you provide this name, the computer directly reserves an area in memory for it. Microsoft Visual Basic allows you to directly use any name for a variable as you see fit. Fortunately, to eliminate the possibility of confusion, you can first let Visual Basic know that you will be using a variable.

In order to reserve that storage area, you have to let the computer know. Letting the computer know is referred to as *declaring* the variable. To declare a variable, you start with the **Dim** word, like this:

Dim

A variable must have a name. The name is written on the right side of the Dim word. There are rules you should follow when naming your variables:

- The name of a variable must begin with a letter or an underscore
- After starting with a letter or an underscore, the name can be made of letters, underscores, and digits in any order
- The name of a variable cannot have a period
- The name of a variable can have up to 255 characters.
- The name of a variable must be unique in the area where it is used

There are some words you should (must) not use to name your variables. Those words are reserved for the VBA internal use. Therefore, those words are called keywords. Some of them are:

And (Bitwise)	And (Condition)	As	Boolean	ByRef	Byte
ByVal	Call	Case	CBool	CByte	CDate
CDbl	CInt	CLng	Const	CSng	CStr
Date	Dim	Do	Double	Each	Else
Elseif	End	EndIf	Error	False	For
Function	Get	GoTo	If	Integer	Let
Lib	Long	Loop	Me	Mid	Mod
New	Next	Not	Nothing	Option	Or (Bitwise)
Or (Condition)	Private	Public	ReDim	REM	Resume
Select	Set	Single	Static	Step	String
Sub	Then	To	True	Until	vbCrLf
vbTab	With	While	Xor		

As mentioned already, to declare a variable, type Dim followed by a name. Here is an example:

```
Sub Exercise()  
    Dim Something  
End Sub
```

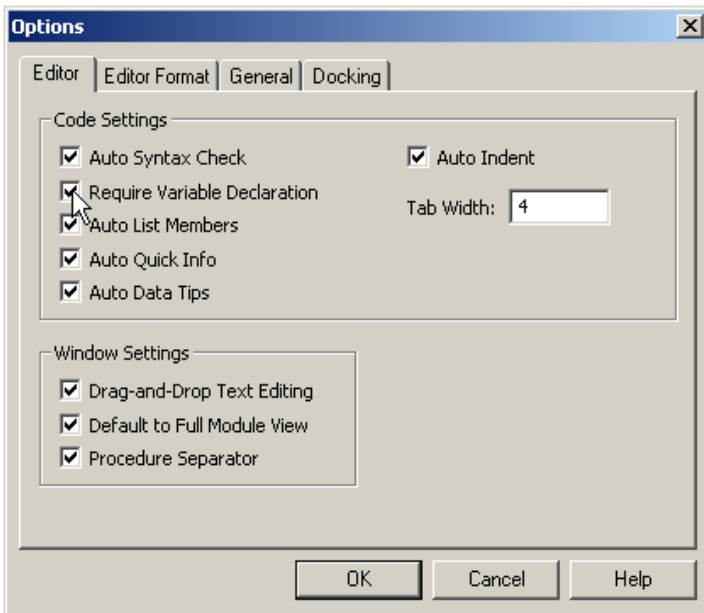
Declaring a variable simply communicates to Visual Basic the name of that variable. You can still use a mix of declared and not-declared variable. If you declare one variable and then start using another variable with a similar but somewhat different name, Visual Basic would still consider that you are using two variables. This can create a great [deal](#) of confusion because you may be trying to use the same variable referred to twice. The solution to this possible confusion is to tell Visual Basic that a variable cannot be used if it has not been primarily declared. To communicate this, on top of each file you use in the Code Editor, type:

```
Option Explicit
```

This can also be done automatically for each file by checking the **Require Variable Declaration** in the Options dialog box.

Practical Learning: Using a Variable

1. On the main menu of Microsoft Visual Basic, click Tools -> Options...
2. Click the Editor property page if necessary. In the Code Settings section, put a check mark in the Require Variable Declaration check box



3. Click OK and return to Microsoft Excel
4. To close Microsoft Excel, click the Office button and click Exit Excel
5. When asked whether you want to save, click No
6. Re-start Microsoft Excel

Declaring Many Variables

In a regular application, it is not unusual to want to use many variables. Once again, you should make it a habit to always declare a variable before using it. To declare a new variable after declaring a first one, you can simply go to the next line and use the Dim keyword to declare the new variable. Here is an example:

```
Sub Exercise()
    Dim Something
    Dim Whatever
End Sub
```

In the same way, you can declare as many variables as you want. Instead of declaring each variable on its own line, you can declare more than one variable on the same line. To do this, use one **Dim** keyword and separate the names of variables with commas. Here are examples:

```
Sub Exercise()
    Dim Father, Mother
    Dim Son, Daughter, Nephew, Niece
    Dim GrandMa
End Sub
```

Notice that each line uses its own **Dim** keyword and every new line of declaration(s) must have a **Dim** keyword.

Value Assignment

We saw that when you declare a variable, the computer reserves a memory space for it but the space is kept empty. After declaring the value, you can store a value you want in the memory that was reserved for it.

To store a value in the memory reserved for a variable, you can assign a value to the variable. To do this, type the name of the variable, followed by the assignment operator which is =, followed by the value you want to store. Here is an example:

```
Sub Exercise()
    Dim Value

    Value = 9374
End Sub
```

As we will learn in the next few lessons, there are different types of values you will use in your document. Also as we will see, the value you (decide to) store must be in accordance with the type of memory that the computer had reserved for the variable.

After assigning a value to a variable, you can use that variable knowing the value it is currently holding. At any time and when necessary, you can change the value held by a variable. That's why it is called a variable (because its value can vary or change). To change the value held by a variable, access the variable again and assign it the new desired value.

A Variable As

A data type tells the computer what kind of variable you are going to use. Before using a variable, you should know how much space it will occupy in memory. Different variables use different amount of space in memory. The information that specifies how much space a variable needs is called a data type. A data type is measured in bytes.

To specify the data type that will be used for a variable, after typing **Dim** followed by the name of the variable, type the **As** keyword, followed by one of the data types we will review next. The formula used is:

```
Dim VariableName As DataType
```

We mentioned earlier that you could use various variables if you judge them necessary. When declaring such variables, we saw that you could declare each on its own line. To specify the data type of a variable, use the same formula as above. Here is an example:

```
Sub Exercise()  
    Dim FirstName As DataType  
    Dim LastName As DataType  
End Sub
```

We also saw that you could declare many variables on the same line as long as you separate the names of the variables with commas. If you are specifying the data type of each, type the comma after each variable. Here are examples:

```
Sub Exercise()  
    Dim FirstName As DataType, LastName As DataType  
    Dim Address As DataType, City As DataType, State As DataType  
    Dim Gender As DataType  
End Sub
```

This code appears as if there is only one type of data. In the next few sections, we will review various types of values that are available. To declare variables of different data types, you declare each on its own line as we saw earlier:

```
Sub Exercise()  
    Dim FullName As DataType1  
    Dim DateHired As DataType2  
    Dim EmploymentStatus As DataType3  
End Sub
```

You can also declare variables of different data types on the same line. To do this, use one Dim keyword and separate the declarations with commas. Here are examples:

```
Sub Exercise()  
    Dim FullName As DataType1, DateHired As DataType2  
    Dim EmploymentStatus As DataType3  
End Sub
```

Type Characters

To make variable declaration a little faster and even convenient, you can replace the **As DataType** expression with a special character that represents the intended data type. Such a character is called a type character and it depends on the data type you intend to apply to a variable. When used, the type character must be the last character of the name of the variable. We will see what characters are available and when it can be applied.

Value Conversion

Every time the user enters a value in an application. That value is primarily considered as text. This means that, if you want to use such a value in an expression or a calculation that expects a specific value other than text, you must convert it from that text. Fortunately, Microsoft Visual Basic provides an effective mechanism to convert a text value to one of the other values we will see next.

To convert text to another value, there is a keyword adapted for the purpose and that depends on the type of value you want to convert it to. We will mention each when necessary.

Integral Numeric Variables

Introduction

If you are planning to use a number in your program, you have a choice from different kinds of numbers that the Visual Basic language can recognize. The Visual Basic language recognizes as a natural number any number that doesn't include a fractional part. In the Visual Basic language,

the number is made of digits only as a combination of 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. No other character is allowed. In future lessons, we will learn that in Microsoft Excel, you can use a comma to separate the thousands, which would make the number easy to read. Microsoft Excel recognizes the comma separator, the Visual Basic language doesn't.

By default, when we refer to a natural number, we expect it in decimal format as a combination of digits. The Visual Basic language also supports the hexadecimal format. A hexadecimal number starts with &H followed by a combination of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, A, B, C, D, E, and F. An example would be &H28E4AABF.

Byte

To declare a variable that would hold natural numbers that range from 0 to 255, use the **Byte** data type. Here is an example:

```
Sub Exercise()
    Dim StudentAge As Byte
End Sub
```

There is no type character for the **Byte** data type.

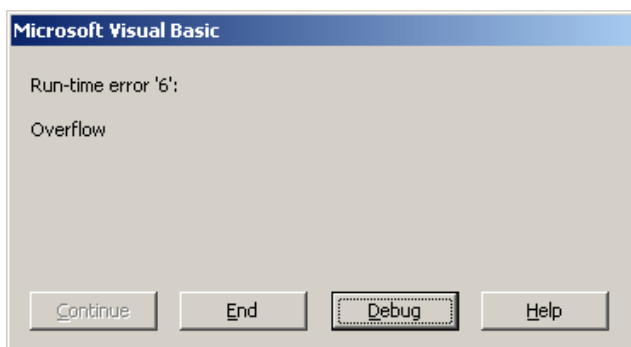
After declaring the variable, you can assign it a small positive number. Here is an example:

```
Sub Exercise()
    Dim Value As Byte

    Value = 246
End Sub
```

You can also use the number in hexadecimal format as long as the number is less than 255.

If you give either a negative value or a value higher to 255, when you attempt to access it, you would receive an error:



To convert a value to a small number, you can use **CByte()**. The formula to use would be:

```
Number = CByte(Value to Convert to Byte)
```

When using **CByte()**, enter the value to convert in the parentheses.


Practical Learning: Using Byte Variables

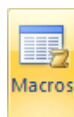
1. In the code, click ActiveCell, press Home, press Enter, and press the up arrow key
2. To use byte variables, change the code as follows:

```
Sub Variables()
    Dim Shirts As Byte
    Dim Pants As Byte
    Dim OtherItems As Byte
    Dim TotalItems As Byte

    Shirts = 6
    Pants = 4
    OtherItems = 2
    TotalItems = Shirts + Pants + OtherItems

    ActiveCell.FormulaR1C1 = TotalItems
End Sub
```

3. To return to Microsoft Excel, on the Standard toolbar, click the View Microsoft Excel button 
4. In Microsoft Excel, click any box



5. In the Code section of the Ribbon, click the Macros button



7. To return to Microsoft Visual Basic, in the Code section of the Ribbon, click Visual Basic

Integer

To declare a variable that would hold a number that ranges from -32768 to 32767, use the **Integer** data type. Here is an example of declaring an integer variable:

```
Sub Exercise()
    Dim Tracks As Integer
End Sub
```

Instead of using `As Integer`, you can use the `%` type character. Therefore, the above declaration could be done as follows:

```
Sub Exercise()
    Dim Tracks%
End Sub
```

After declaring the variable, you can assign the desired value to it. If you assign a value lower than -32768 or higher than 32767, when you decide to use it, you would receive an error.

If you have a value that needs to be converted into a natural number, you can use **CInt()** using the following formula:

```
Number = CInt(Value to Convert)
```

Between the parentheses of **CInt()**, enter the value, text, or expression that needs to be converted.

Long

A long integer is a number that can be used for a variable involving greater numbers than integers. To declare a variable that would hold such a large number, use the **Long** data type. Here is an example:

```
Sub Exercise()
    Dim Population As Long
End Sub
```

The type character for the Long data type is `@`. The above variable could be declared as:

```
Sub Exercise()
    Dim Population@
End Sub
```

A **Long** variable can store a value between $\approx 2,147,483,648$ and $2,147,483,647$ (remember that the commas are used to make the numbers easy to read; do not be used them in your code). Therefore, after declaring a **Long** variable, you can assign it a number in that range.

To convert a value to a long integer, call **CLng()** using the following formula:

```
Number = CLng(Value to Convert)
```

To convert a value to long, enter it in the parentheses of **CLng()**.

Decimal Variables

Single Precision

In computer programming, a decimal number is one that represents a fraction. Examples are 1.85 or 426.88. If you plan to use a variable that would that type of number but precision is not your main concern, declare it using the **Single** data type. Here is an example:

```
Sub Exercise()
    Dim Distance As Single
End Sub
```

The type character for the Single data type is `!`. Based on this, the above declaration could be done as:

```
Sub Exercise()
    Dim Distance!
End Sub
```

A **Single** variable can hold a number between $1.401298e^{45}$ and $3.402823e^{38}$. for negative

values or between $1.401298e^{-45}$ and $3.402823e^{38}$ for positive values.

If you have a value that needs to be converted, use **CSng()** with the following formula:

```
Number = CSng(Value to Convert)
```

In the parentheses of **CSng()**, enter the value to be converted.

Double Precision

If you want to use a decimal number that requires a good deal of precision, declare a variable using the **Double** data type. Here is an example of declaring a **Double** variable:

```
Sub Exercise()
    Dim Distance As Double
End Sub
```

Instead of **As Double**, the type character you can use is #:

```
Sub Exercise()
    Dim Distance#
End Sub
```

A **Double** variable can hold a number between $-1.79769313486231e^{308}$ and $4.94065645841247e^{324}$ for negative values or between $4.94065645841247e^{324}$ and $1.79769313486231e^{308}$ for positive values.

To convert a value to double-precision, use **Cdbl()** with the following formula:

```
Number = Cdbl(Value to Convert)
```

In the parentheses of **Cdbl()**, enter the value that needs to be converted.


Practical Learning: Using Decimal Variables

1. Change the code as follows:

```
Sub Variables()
    Dim Side As Double
    Dim Perimeter As Double

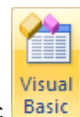
    Side = 32.75
    Perimeter = Side * 4

    ActiveCell.FormulaR1C1 = Perimeter
End Sub
```

2. To return to Microsoft Excel, on the Standard toolbar, click the View Microsoft Excel button 
3. In Microsoft Excel, click any box



4. In the Code section of the Ribbon, click the Macros button
5. In the Macros dialog box, make sure Exercise1 is selected and click Run



6. To return to Microsoft Visual Basic, in the Code section of the Ribbon, click Visual Basic

A String

A string is a character or a combination of characters that constitute text of any kind and almost any length. To declare a string variable, use the **String** data type. Here is an example:

```
Sub Exercise()
    Dim CountryName As String
End Sub
```

The type character for the String data type is \$. Therefore, the above declaration could be written as:

```
Sub Exercise()
    Dim CountryName$
End Sub
```

As mentioned already, after declaring a variable, you can assign a value to it. The value of a string variable must be included inside of double-quotes. Here is an example:

```
Sub Exercise()
    Dim CountryName As String

    CountryName = "BrÃ©sil"
End Sub
```

If you have a value that is not primarily text and you want to convert it to a string, use **CStr()** with the following formula:

```
CStr(Value To Convert to String)
```

In the parentheses of the **CStr()**, enter the value that you want to convert to string.


Practical Learning: Using a String

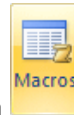
1. Change the code as follows:

```
Sub Variables()
    Dim CustomerName As String

    CustomerName = "Paul Bertrand Yamaguchi"

    ActiveCell.FormulaR1C1 = CustomerName
End Sub
```

2. To return to Microsoft Excel, on the Standard toolbar, click the View Microsoft Excel button 
3. In Microsoft Excel, click any box



4. In the Code section of the Ribbon, click the Macros button
5. In the Macros dialog box, make sure Exercise1 is selected and click Run



6. To return to Microsoft Visual Basic, in the Code section of the Ribbon, click Visual Basic

Currency Values

The **Currency** data type is used to deal with monetary values. Here is an example of declaring it:

```
Sub Exercise()
    Dim StartingSalary As Currency
End Sub
```

Instead of using the As Currency expression, you can use @ as the type character to declare a currency variable. Here is an example of declaring it:

```
Sub Exercise()
    Dim StartingSalary@
End Sub
```

A variable declared with the **Currency** keyword can store a value between "â¬" 922,337,203,685,477.5808 and 922,337,203,685,477.5807. Once again, keep in mind that the commas here are used only to make the number easy to read. Don't use the commas in a number in your code. Also, when assigning a value to a currency-based variable, do not use the currency symbol.

Here is an example of assigning a currency number to a variable:

```
Sub Exercise()
    Dim StartingSalary As Currency

    StartingSalary = 66500
End Sub
```

If you want to convert a value to currency, use **CCur()** with the following formula:

```
Number = CCur(Value to Convert)
```

To perform this conversion, enter the value in the parentheses of **CCur()**.

Practical Learning: Using Currency Values

1. Change the code as follows:

```
Sub Variables()
```


```

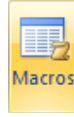
Dim NumberOfShirts As Byte
Dim PriceOneShirt As Currency
Dim TotalPriceShirts As Currency

NumberOfShirts = 5
PriceOneShirt = 1.25
TotalPriceShirts = NumberOfShirts * PriceOneShirt

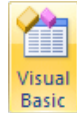
ActiveCell.FormulaR1C1 = TotalPriceShirts
End Sub

```

2. To return to Microsoft Excel, on the Standard toolbar, click the View Microsoft Excel button 
3. In Microsoft Excel, click any box



4. In the Code section of the Ribbon, click the Macros button
5. In the Macros dialog box, make sure Exercise1 is selected and click Run



6. To return to Microsoft Visual Basic, in the Code section of the Ribbon, click Visual Basic

A Date

In Visual Basic, a **Date** data type can be used to store a date value. Therefore, to declare either a date or a time variables, use the **Date** data type. Here is an example:

```

Sub Exercise()
    Dim DateOfBirth As Date
End Sub

```

After declaring the variable, you can assign it a value. A date value must be included between two # signs. Here is an example:

```

Sub Exercise()
    Dim DateOfBirth As Date

    DateOfBirth = #10/8/1988#
End Sub

```

There are various formats you can use for a date. We will deal with them in another lesson.

If you have a string or an expression that you want to convert to a date value, use **CDate()** based on the following formula:

```

Result = CDate(Value to Convert)

```

In the parentheses of **CDate()**, enter the value that needs to be converted.

Practical Learning: Using a Date

1. Change the code as follows:


```

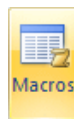
Sub Variables()
    Dim DepositDate As Date

    DepositDate = #2/5/2008#

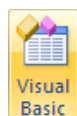
    ActiveCell.FormulaR1C1 = DepositDate
End Sub

```

2. To return to Microsoft Excel, on the Standard toolbar, click the View Microsoft Excel button 
3. In Microsoft Excel, click any box



4. In the Code section of the Ribbon, click the Macros button
5. In the Macros dialog box, make sure Exercise1 is selected and click Run



6. To return to Microsoft Visual Basic, in the Code section of the Ribbon, click Visual Basic

In Visual Basic, the **Date** data type can also be used to store a time value. Here is an example of declaring a variable that can hold a time value:

```
Sub Exercise()
    Dim ShiftTimeIn As Date
End Sub
```

After declaring the variable, to assign a value to it, include the value between two # signs. The value follows different rules from a date.

To convert a value or an expression to time, use **CDate()**.

Any-Type Variables

A Variant

So far, we declared variables knowing the types of values we wanted them to hold. VBA provides a universal (or vague) data type you can use for any type of value. The Variant data type is used to declare a variable whose type is not explicitly specified. This means that a Variant data type can hold any type of value you want.

Here are examples of Variant-declared variables that hold different types of values:

```
Sub Exercise()
    Dim FullName As Variant
    Dim EmploymentStatus As Variant
    Dim HourlySalary As Variant
    Dim DateHired As Variant

    FullName = "Patricia Katts"
    EmploymentStatus = 2
    HourlySalary = 35.65
    DateHired = #6/22/2004#
End Sub
```

A Variable Without a Data Type

In the variables we declared in the last few sections, we specified a data type for each. You can declare a variable without giving its data type. Here are examples:

```
Sub Exercise()
    Dim FullName
    Dim EmploymentStatus
    Dim HourlySalary
    Dim DateHired
End Sub
```

Of course, you can declare more than one variable on the same line.

To indicate how much space is needed for the variable, you must assign it a value. Here are examples:

```
Sub Exercise()
    Dim FullName
    Dim EmploymentStatus
    Dim HourlySalary
    Dim DateHired

    FullName = "Patricia Katts"
    EmploymentStatus = 2
    HourlySalary = 35.65
    DateHired = #6/22/2004#
End Sub
```

Once the variable holds a value, you can use it as you see fit.

The Scope or Lifetime of a Variable

Introduction

So far, we were declaring our variables between the **Sub Name** and the **End Sub** lines. Such a variable is referred to as a local variable. A local variable is confined to the area where it is declared. Here is an example:

```
Option Explicit

Sub Exercise()
    Dim FirstName As String
```

```
    FirstName = "Patricia"
End Sub
```

You cannot use such a variable outside of its **Sub Name** and the **End Sub** lines.

Global Variables

A global variable is a variable declared outside of the **Sub Name** and the **End Sub** lines. Such a variable is usually declared in the top section of the file. Here is an example:

```
Option Explicit

Dim LastName As String

Sub Exercise()

End Sub
```

After declaring a global variable, you can access it in the other areas of the file. Here is an example:

```
Option Explicit

Dim LastName As String

Sub Exercise()
    Dim FirstName As String

    FirstName = "Patricia"
    LastName = "Katts"
End Sub
```

Although we declared our global variable inside of the file where it was used, you can also declare a global variable in a separate module to be able to use it in another module.

The Access Level of a Global Variable

Introduction

When using a global variable, the Visual Basic language allows you to control its access level. The access level of a variable is a process of controlling how much access a section of code has on the variable.

Private Variables

A variable is referred to as private if it can be accessed only by code from within the same file (the same module) where it is used. To declare such a variable, instead of **Dim**, you use the **Private** keyword. Here is an example:

```
Option Explicit

Private LastName As String

Sub Exercise()
    Dim FirstName As String

    FirstName = "Patricia"
    LastName = "Katts"
End Sub
```

Remember that a private variable can be accessed by any code in the same module. In the next lesson, we will learn how to create other sections of code.

Public Variables

A variable is referred to as public if it can be accessed by code either from within the same file (the same module) where it is declared or from code outside its module. To declare a public variable, instead of **Dim**, you use the **Public** keyword. Here is an example:

```
Option Explicit

Private LastName As String
Public FullName As String

Sub Exercise()
    Dim FirstName As String

    FirstName = "Patricia"
    LastName = "Katts"
    FullName = FirstName & " " & LastName
End Sub
```

As a reminder, a public variable is available to code inside and outside of its module. This means that you can create a module, declare a public variable in it, and access that variable in another file (module) where needed.

A private variable is available inside its module but not outside its module. If you declare a private variable in a module and try accessing it in another module, you would receive an error:

Module 1:

```
Option Explicit
```

```
Private FullName As String
```

Module 2:

```
Option Explicit
```

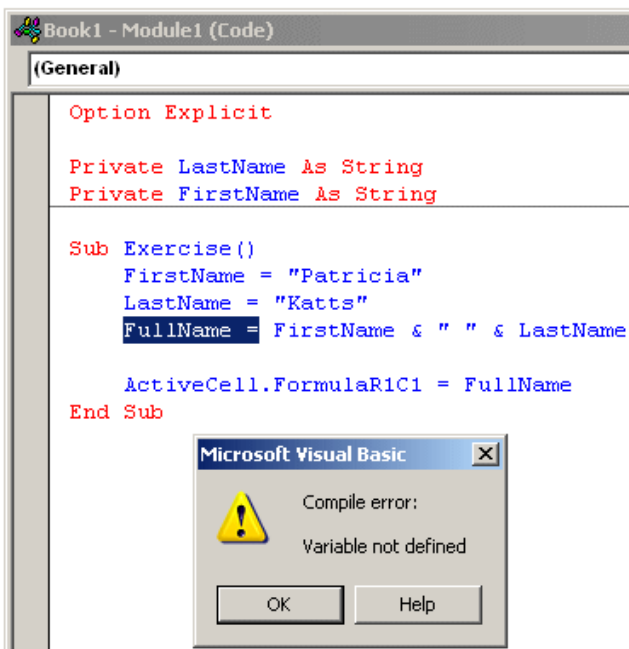
```
Private LastName As String
```

```
Private FirstName As String
```

```
Sub Exercise()
    FirstName = "Patricia"
    LastName = "Katts"
    FullName = FirstName & " " & LastName

    ActiveCell.FormulaR1C1 = FullName
End Sub
```

This would produce:



❖ Practical Learning: Closing Microsoft Excel

1. To close Microsoft Visual Basic, on the main menu, click File -> Close and Return to Microsoft Excel
2. To close Microsoft Excel, click the Office Button and click Exit Excel.
If you are asked whether you want to save the file, click No



VBA Operators and Operands

VBA Operators

Introduction

An operation is an action performed on one or more values either to modify one value or to produce a new value by combining existing values. Therefore, an operation is performed using at least one symbol and one value. The symbol used in an operation is called an operator. A variable or a value involved in an operation is called an operand.

A unary operator is an operator that performs its operation on only one operand.

An operator is referred to as binary if it operates on two operands.

Dimensioning a Variable

When interacting with Microsoft Excel, you will be asked to provide a value. Sometimes, you will be presented with a value to view or change. Besides the values you use in a spreadsheet, in the previous lesson, we learned that we could also declare variables in code and assign values to them.

In the previous lesson, we saw that we could use the **Dim** operator to declare a variable. Here is an example:

Option Explicit

```
Sub Exercise()  
    Dim Value
```

```
End Sub
```

After declaring a variable like this, we saw that we could then use it as we saw fit.

The Assignment Operator

We mentioned that you could declare a variable but not specify the type of value that would be stored in the memory area reserved for it. When you have declared a variable, the computer reserves space in the memory and gives an initial value to the variable. If the variable is number based, the computer gives its memory an initial value of 0. If the variable is string based, the computer fills its memory with an empty space, also referred to as an empty string.

Initializing a variable consists of giving it a value as soon as the variable has been declared. To initialize a variable, you use the assignment operator which is "=". You type the name of the variable, followed by =, and followed by the desired value. The value depends on the type of variable. If the variable is integral based, give it an appropriate natural number. Here is an example:

```
Sub Exercise()  
    Dim Integral As Integer
```

```
    Integral = 9578
```

```
End Sub
```

If the variable is made to hold a decimal number, initialize it with a number that can fit in its type of variable. Here is an example:

```
Sub Exercise()  
    Dim Distance As Double
```

```
    Distance = 257.84
```

```
End Sub
```

If the variable is for a string, you can initialize it with an empty string or put the value inside of double-quotes.

The Line Continuation Operator: _

If you plan to write a long piece of code, to make it easier to read, you may need to divide it in various lines. To do this, you can use the line continuation operator represented by a white space followed by an underscore.

To create a line continuation, put an empty space, then type the underscore, and continue your code in the next line. Here is an example:

```
Sub _
Exercise()

End Sub
```

The Parentheses: ()

Parentheses are used in various circumstances. The parentheses in an operation help to create sections in an operation. This regularly occurs when more than one operators are used in an operation. Consider the following operation:

$8 + 3 * 5$

The result of this operation depends on whether you want to add 8 to 3 then multiply the result by 5 or you want to multiply 3 by 5 and then add the result to 8. Parentheses allow you to specify which operation should be performed first in a multi-operator operation. In our example, if you want to add 8 to 3 first and use the result to multiply it by 5, you would write $(8 + 3) * 5$. This would produce 55. On the other hand, if you want to multiply 3 by 5 first then add the result to 8, you would write $8 + (3 * 5)$. This would produce 23.

As you can see, results are different when parentheses are used on an operation that involves various operators. This concept is based on a theory called operator precedence. This theory manages which operation would execute before which one; but parentheses allow you to completely control the sequence of these operations.

The Comma ,

The comma is used to separate variables used in a group. For example, a comma can be used to delimit the names of variables that are declared on the same line. Here is an example:

```
Sub Exercise()
    Dim FirstName As String, LastName As String, FullName As String
End Sub
```

The Double Quotes: ""

A double-quote is used to delimit a group of characters and symbols. To specify this delimitation, the double-quote is always used in combination with another double-quote, as in `""`. What ever is inside the double-quotes is the thing that need to be delimited. The value inside the double-quotes is called a string. Here is an example:

```
Sub Exercise()
    Dim FirstName As String, LastName As String, FullName As String

    FirstName = "ValÃ"re"
    ActiveCell.FormulaR1C1 = FirstName
End Sub
```

The Colon Operator :

Most of the time, to make various statements easier to read, you write each on its own line. Here are examples:

```
Sub Exercise()
    Dim FirstName As String, LastName As String

    FirstName = "ValÃ"re"
    LastName = "Edou"
End Sub
```

The Visual Basic language allows you to write as many statements as necessary on the same line. When doing this, the statements must be separated by a colon. Here is an example:

```
Sub Exercise()
    Dim FirstName As String, LastName As String

    FirstName = "ValÃ"re" : LastName = "Edou"

    ActiveCell.FormulaR1C1 = FirstName
End Sub
```

String Concatenation: &

The & operator is used to append two strings or expressions. This is considered as concatenating them. For example, it could allow you to concatenate a first name and a last name, producing a full name. The general syntax of the concatenation operator is:

Value1 & *Value2*

In the same way, you can use as many & operators as you want between any two strings or expressions. After concatenating the expressions or values, you can assign the result to another variable or expression using the assignment operator. Here are examples:

```
Sub Exercise()
    Dim FirstName As String, LastName As String, FullName As String

    FirstName = "ValÃ"re"
    LastName = "Edou"
    FullName = FirstName & " " & LastName
End Sub
```

Carriage Return-Line Feed

If you are displaying a string but judge it too long, you can segment it in appropriate sections as you see fit. To do this, you can use **vbCrLf**. Here is an example:

```
Sub Exercise()
    Dim FirstName As String, LastName As String, FullName As String
    Dim Accouncement As String

    FirstName = "ValÃ"re"
    LastName = "Edou"
    FullName = FirstName & " " & LastName
    Accouncement = "Student Registration - Student Full Name: " & _
        vbCrLf & FullName
    ActiveCell.FormulaR1C1 = Accouncement
End Sub
```

Arithmetic Operators

Positive Unary Operator: +

Algebra uses a type of ruler to classify numbers. This ruler has a middle position of zero. The numbers on the left side of the 0 are referred to as negative while the numbers on the right side of the rulers are considered positive:

-∞		-6	-5	-4	-3	-2	-1		1	2	3	4	5	6		+∞
0																
-∞		-6	-5	-4	-3	-2	-1		1	2	3	4	5	6		+∞

A value on the right side of 0 is considered positive. To express that a number is positive, you can write a + sign on its left. Examples are +4, +228, +90335. In this case the + symbol is called a unary operator because it acts on only one operand.

The positive unary operator, when used, must be positioned on the left side of its operand, never on the right side.

As a mathematical convention, when a value is positive, you don't need to express it with the + operator. Just writing the number without any symbol signifies that the number is positive. Therefore, the numbers +4, +228, and +90335 can be, and are better, expressed as 4, 228, 90335. Because the value does not display a sign, it is referred as **unsigned**.

The Negative Operator -

As you can see on the above ruler, in order to express any number on the left side of 0, it must be appended with a sign, namely the - symbol. Examples are -12, -448, -32706. A value accompanied by - is referred to as negative.

The - sign must be typed on the left side of the number it is used to negate.

Remember that if a number does not have a sign, it is considered positive. Therefore, whenever a number is negative, it MUST have a - sign. In the same way, if you want to change a value from positive to negative, you can just add a - sign to its left.

Addition +

The addition is performed with the + sign. It is used to add one value to another. Here is an example:

```
Sub Exercise()
    Dim Side#
    Dim Perimeter#
```

```
Side# = 42.58
Perimeter# = Side# + Side# + Side# + Side#
End Sub
```

Besides arithmetic operations, the + symbol can also be used to concatenate strings, that is, to add one string to another. This is done by appending one string at the end of another. Here is an example:

```
Sub Exercise()
    Dim FirstName$, LastName$, FullName$

    FirstName$ = "Danielle"
    LastName$ = "Kouma"
    FullName$ = FirstName$ + " " + LastName$

    ActiveCell.FormulaR1C1 = FullName$
End Sub
```

Multiplication *

The multiplication operation allows you to add a number to itself a certain number of times set by another number. The multiplication operation is performed using the * sign. Here is an example:

```
Sub Exercise()
    Dim Side#
    Dim Area#

    Side# = 42.58
    Area# = Side# * Side#
End Sub
```

Subtraction -

The subtraction operation is performed using the - sign. This operation produces the difference of two or more numbers. It could also be used to display a number as a negative value. To subtract 28 from 65, you express this with 65-28.

The subtraction can also be used to subtract the values of two values.

Integer Division \

Dividing an item means cutting it in pieces or fractions of a set value. Therefore, the division is used to get the fraction of one number in terms of another. The Visual Basic language provides two types of operations for the division. If you want the result of the operation to be a natural number, called an integer, use the backlash operator "\" as the divisor. The formula to use is:

```
Value1 \ Value2
```

This operation can be performed on two types of valid numbers, with or without decimal parts. After the operation, the result would be a natural number.

Decimal Division /

The second type of division results in a decimal number. It is performed with the forward slash "/". Its formula is:

```
Value1 / Value2
```

After the operation is performed, the result is a decimal number.

Exponentiation ^

Exponentiation is the ability to raise a number to the power of another number. This operation is performed using the ^ operator (Shift + 6). It uses the following formula:

$$y^x$$

In Microsoft Visual Basic, this formula is written as:

$$y^x$$

and means the same thing. Either or both y and x can be values, variables, or expressions, but they must carry valid values that can be evaluated. When the operation is performed, the value of y is raised to the power of x.

Remainder: Mod

The division operation gives a result of a number with or without decimal values, which is fine in some circumstances. Sometimes you will want to get the value remaining after a division renders a natural result.

The remainder operation is performed with keyword **Mod**. Its formula is:

Value1 Mod Value2

The result of the operation can be used as you see fit or you can display it in a control or be involved in another operation or expression.

Bit Manipulations

Introduction

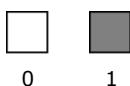
From our introduction to variables, you may remember that the computer stores its data in memory using small locations that look like boxes and each box contains a bit of information. Because a bit can be represented only either as 1 or 0, we can say that each box contains 1 or 0. Bit manipulation consists of changing the value (1 or 0, or 0 or 1) in a box. As we will see in the next few operations, it is not just about changing a value. It can involve reversing a value or kind of "moving" a box from its current position to the next position.

The operations on bits are performed on 1s and 0s only. This means that any number in decimal or hexadecimal format involved in a bit operation must be converted to binary first.

You will almost never perform some of the operations we are going to review. You will hardly perform some other operations. There is only one operation you will perform sometimes: the **OR** operation.

"Reversing" a Bit

Remember that, at any time, a box (or chunk) in memory contains either 1 or 0:



Bit reversal consists of reversing the value of a bit. If the box contains 1, you can reverse it to 0. If it contains 0, you can reverse it to 1. To support this operation, the Visual Basic language provides the **Not** Operator.

As an example, consider the number 286. The decimal number 286 converted to binary is 100011110. You can reverse each bit as follows:

286	1	0	0	0	1	1	1	1	0
Not 286	0	1	1	1	0	0	0	0	1

Bitwise Conjunction

Bitwise conjunction consists of adding the content of one box (a bit) to the content of another box (a bit). To support the bitwise conjunction operation, the Visual Basic language provides the **And** operator.

To perform the bit addition on two numbers, remember that they must be converted to binary first. Then:

- If a bit with value 0 is added to a bit with value 0, the result is 0

Bit0	0
Bit1	0
Bit0 And Bit1	0

- If a bit with value 1 is added to a bit with value 0, the result is 0

Bit0	1
Bit1	0
Bit0 And Bit1	0

- If a bit with value 0 is added to a bit with value 1, the result is 0

Bit0	0
Bit1	1
Bit0 And Bit1	0

- If a bit with value 1 is added to a bit with value 1, the result is 1

Bit0	1
Bit1	1
Bit0 And Bit1	1

As an example, consider the number 286 bit-added to 475. The decimal number 286 converted to binary is 100011110. The decimal number 4075 converted to binary is 11111101011. Based on the above 4 points, we can add these two numbers as follows:

286	0	0	0	1	0	0	0	1	1	1	1	0
4075	1	1	1	1	1	1	1	0	1	0	1	1
286 And 4075	0	0	0	1	0	0	0	0	1	0	1	0

Therefore, 286 And 4075 produces 100001010 which is equivalent to:

	Bit8	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
	256	128	64	32	16	8	4	2	1
286 And 4075	1	0	0	0	0	1	0	1	0
	256	0	0	0	0	8	0	2	0

This means that 286 And 4075 = 256 + 16 + 2 = 266

This can also be programmatically calculated as follows:

```
Sub Exercise()
    Dim Number1 As Integer
    Dim Number2 As Integer
    Dim Result As Integer

    Number1 = 286
    Number2 = 4075
    Result = Number1 And Number2

    ActiveCell.FormulaR1C1 = Result
End Sub
```

Bitwise Disjunction

Bitwise disjunction consists of disjoining one a bit from another bit. To support this operation, the Visual Basic language provides the **Or** operator.

To perform a bitwise conjunction on two numbers, remember that they must be converted to binary first. Then:

- If a bit with value 0 is added to a bit with value 0, the result is 0

Bit0	0
Bit1	0
Bit0 Or Bit1	0

- If a bit with value 1 is added to a bit with value 0, the result is 1

Bit0	1
Bit1	0
Bit0 Or Bit1	1

- If a bit with value 0 is added to a bit with value 1, the result is 1

Bit0	0
Bit1	1
Bit0 Or Bit1	1

- If a bit with value 1 is added to a bit with value 1, the result is 1

Bit0	1
Bit1	1
Bit0 Or Bit1	1

As an example, consider the number 305 bit-disjoined to 2853. The decimal number 305 converted to binary is 100110001. The decimal number 2853 converted to binary is 101100100101. Based on the above 4 points, we can disjoin these two numbers as follows:

305	0	0	0	1	0	0	1	1	0	0	0	1
2853	1	0	1	1	0	0	1	0	0	1	0	1
305 Or 2853	1	0	1	1	0	0	1	1	0	1	0	1

Therefore, 305 Or 2853 produces 101100110101 which is equivalent to:

	Bit11	Bit10	Bit9	Bit8	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
	2048	1024	512	256	128	64	32	16	8	4	2	1
305 Or 2853	1	0	1	1	0	0	1	1	0	1	0	1
	2048	0	512	256	0	0	32	16	0	4	0	1

This means that 286 And 4075 = 2048 + 512 + 256 + 32 + 16 + 4 + 1 = 2869

This can also be programmatically calculated as follows:

```
Sub Exercise()
    Dim Number1 As Integer
    Dim Number2 As Integer
    Dim Result As Integer

    Number1 = 286
    Number2 = 4075
    Result = Number1 Or Number2

    ActiveCell.FormulaR1C1 = Result
End Sub
```

Bitwise Exclusion

Bitwise exclusion consists of adding two bits with the following rules. To support bitwise exclusion, the Visual Basic language provides an operator named **Xor**:

- If both bits have the same value, the result is 0

Bit0	0	1
Bit1	0	1
Bit0 Xor Bit1	0	0

- If both bits are different, the result is 1

Bit0	0	1
Bit1	1	0
Bit0 Xor Bit1	1	1

As an example, consider the number 618 bit-excluded from 2548. The decimal number 618 converted to binary is 1001101010. The decimal number 2548 converted to binary is 10011110100. Based on the above 2 points, we can bit-exclude these two numbers as follows:

618	0	0	1	0	0	1	1	0	1	0	1	0
2548	1	0	0	1	1	1	1	1	0	1	0	0
618 Xor 2548	1	0	1	1	1	0	0	1	1	1	1	0

Therefore, 305 Or 2853 produces 101110011110 which is equivalent to:

	Bit11	Bit10	Bit9	Bit8	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
	2048	1024	512	256	128	64	32	16	8	4	2	1
618 Xor 2548	1	0	1	1	1	0	0	1	1	1	1	0
	2048	0	512	256	128	0	0	16	8	4	2	0

This means that 286 And 4075 = 2048 + 512 + 256 + 128 + 16 + 8 + 4 + 2 = 2974

This can also be programmatically calculated as follows:

```
Sub Exercise()
    Dim Number1 As Integer
    Dim Number2 As Integer
    Dim Result As Integer

    Number1 = 286
    Number2 = 4075
    Result = Number1 Xor Number2

    ActiveCell.FormulaR1C1 = Result
```

[Previous](#)

Copyright © 2008-2010 FunctionX

[Next](#)



Introduction to Procedures and Functions


Introduction to Procedures

Procedures

A procedure is a section of code created to carry an assignment, separate from a spreadsheet, whose action can be used to complement a spreadsheet. You create the procedure by writing code. One of the advantages of a procedure is that, once it exists, you can access it when necessary and as many times as you want.

There are two categories of procedures you will use in your spreadsheets: those that are already installed with Microsoft Excel and those you will create.

❖ Practical Learning: Introducing Procedures

1. Start Microsoft Excel
2. On the [Ribbon](#), click Developer
3. In the Code section, click the Visual Basic button 
4. To create a module, on the main menu, click Insert -> Module
5. If the Properties window is not available, on the main menu, click View -> Properties Windows. In the Properties window, click (Name)
6. Type **Procedures** and press Enter

In the Visual Basic language, like most other languages, there are two types of procedures: functions and sub procedures.

Introduction to Sub-Procedures

A sub procedure is an assignment that is carried but does not give back a result. To create a sub procedure, start with the **Sub** keyword followed by a name (like everything else, a procedure must have a name). The name of a procedure is always followed by parentheses. At the end of the sub procedure, you must type **End Sub**. Therefore, the primary formula to create a sub procedure is:

```
Sub ProcedureName( )
```

```
End Sub
```

The name of a procedure should follow the same rules we learned to name the variables. In addition:

- If the procedure performs an action that can be represented with a verb, you can use that verb to name it. Here are examples: show, display
- To make the name of a procedure stand, you should start it in uppercase. Examples are Show, Play, Dispose, Close
- You should use explicit names that identify the purpose of the procedure. If a procedure would be used as a result of another procedure or a control's event, reflect it on the name of the sub procedure. Examples would be: afterupdate, longbefore.
- If the name of a procedure is a combination of words, you should start each word in uppercase. An example is AfterUpdate

The section between the **Sub** and the **End Sub** lines is referred to as the body of the procedure. Here is an example:

```
Sub CreateCustomer( )
```

```
End Sub
```

In the body of the procedure, you carry the assignment of the procedure. It is also said that you define the procedure or you implement the procedure.

One of the actions you can in the body of a procedure consists of declaring a variable. There is no restriction on the type of variable you can declare in a procedure. Here is an example:

```
Sub CreateCustomer()
    Dim strFullName As String
End Sub
```

In the same way, you can declare as many variables as you need inside of a procedure. The actions you perform inside of a procedure depend on what you are trying to accomplish. For example, a procedure can simply be used to create a string. The above procedure can be changed as follows:

```
Sub CreateCustomer()
    Dim strFullName As String

    strFullName = "Paul Bertrand Yamaguchi"
End Sub
```

Calling a Sub Procedure

Once you have a procedure, whether you created it or it is part of the Visual Basic language, you can use it. Using a procedure is also referred to as calling it.

Before calling a procedure, you should first locate the section of code in which you want to use it. To call a simple procedure, type its name. Here is an example:

```
Sub CreateCustomer()
    Dim strFullName As String

    strFullName = "Paul Bertrand Yamaguchi"
End Sub

Sub Exercise()
    CreateCustomer
End Sub
```

Besides using the name of a procedure to call it, you can also precede it with the **Call** keyword. Here is an example:

```
Sub CreateCustomer()
    Dim strFullName As String

    strFullName = "Paul Bertrand Yamaguchi"
End Sub

Sub Exercise()
    Call CreateCustomer
End Sub
```

When calling a procedure, without or with the **Call** keyword, you can optionally type an opening and a closing parentheses on the right side of its name. Here is an example:

```
Sub CreateCustomer()
    Dim strFullName As String

    strFullName = "Paul Bertrand Yamaguchi"
End Sub

Sub Exercise()
    CreateCustomer()
End Sub
```

Procedures and Access Levels

Like a variable access, the access to a procedure can be controlled by an access level. A procedure can be made private or public. To specify the access level of a procedure, precede it with the **Private** or the **Public** keyword. Here is an example:

```
Private Sub CreateCustomer()
    Dim strFullName As String

    strFullName = "Paul Bertrand Yamaguchi"
End Sub
```

The rules that were applied to global variables are the same:

- **Private:** If a procedure is made private, it can be called by other procedures of the same module. Procedures of outside modules cannot access such a procedure. Also, when a procedure is private, its name does not appear in the Macros dialog box
- **Public:** A procedure created as public can be called by procedures of the same module and by procedures of other modules. Also, if a procedure was created as public, when you access the Macros dialog box, its name appears and you can run it from there

Introduction to Functions

Introduction

Like a sub procedure, a function is used to perform an assignment. The main difference between a sub procedure and a function is that, after carrying its assignment, a function gives back a result.

We also say that a function "returns a value". To distinguish both, there is a different syntax you use for a function.

Creating a Function

To create a function, you use the **Function** keyword followed by a name and parentheses. Unlike a sub procedure, because a function returns a value, you must specify the type of value the function will produce. To give this information, on the right side of the closing parenthesis, you can type the **As** keyword, followed by a data type. To indicate where a function stops, type **End Function**. Based on this, the minimum syntax used to create a function is:

```
AccessModifier Function FunctionName() As DataType
```

```
End Function
```

As seen for a sub procedure, a function can have an access modifier.

The **Function** keyword is required.

The name of a function follows the same rules and suggestions we reviewed for names of sub procedures.

The **As** keyword may be required (in the next sections, we will review the alternatives to the **As DataType** expression).

The *DataType* factor indicates the type of value that the function will return. If the function will produce a word or a group of words, you can create it as **String**. The other data types are also valid in the contexts we reviewed them in the previous lesson. Here is an example:

```
Function GetFullName() As String
```

```
End Function
```

❖ Practical Learning: Creating a Function

- Click an empty area in the Code editor and, to create a function, type the following code:

```
Option Explicit

Function GetCustomerName() As String

End Function
```

Using a Type Character

As done with variables, you can also use a type character as the return type of a function and omit the **As DataType** expression. The type character is typed on the right side of the function name and before the opening parenthesis. An example would be `GetFullName$()`. As with the variables, you must use the appropriate type character for the function:

Character	The function must return
\$	A string
%	An integral value between -32768 and 32767
&	An integer of small or large scale
!	A decimal number with single precision
#	A decimal number with double precision
@	A monetary value

Here is an example:

```
Function GetFullName$()
```

```
End Function
```

As mentioned for a sub procedure, the section between the **Function** and the **End Function** lines is the body of the function. It is used to describe what the function does. As done on a sub procedure, one of the actions you can perform in a function is to declare a (local) variable and use it as you see fit. Here is an example:

```
Function CallMe() As String
    Dim Salute As String
    Salute = "You can call me Al"
End Function
```

Returning a Value From a Function

After performing an assignment in a function, to indicate the value it returns, somewhere after the assignment and before the **End Function** line, you can type the name of the function, followed by the = sign, followed by the value that the function returns. Here is an example in which a function returns a name:

```
Function GetFullName$(
    Dim FirstName As String, LastName As String

    FirstName = "Patricia"
```

```
LastName = "Katts"
```

```
GetFullName = LastName & ", " & FirstName
```

```
End Function
```

❖ Practical Learning: Implementing a Function


1. To implement the function, change its code as follows:

```
Option Explicit
```

```
Function GetCustomerName() As String
```

```
    GetCustomerName = "Paul Bertrand Yamaguchi"
```

```
End Function
```

2. To return to Microsoft Excel, on the Standard toolbar, click the View Microsoft Excel button 

Calling a Function

As done for the sub procedure, in order to use a function in your program, you must call it. Like a sub procedure, to call a function, you can simply type its name in the desired section of the program. Here is an example:

```
Function CallMe() As String
```

```
    Dim Salute As String
```

```
    Salute = "You can call me Al"
```

```
    CallMe = Salute
```

```
End Function
```

```
Sub Exercise()
```

```
    CallMe
```

```
End Sub
```

When calling the function, you can optionally type the parentheses on the right side of its name.

The primary purpose of a function is to return a value. To better take advantage of such a value, you can assign the name of a function to a variable in the section where you are calling the function. Here is an example:

```
Function GetFullName$()
```

```
    Dim FirstName As String, LastName As String
```

```
    FirstName = "Patricia"
```

```
    LastName = "Katts"
```

```
    GetFullName = LastName & ", " & FirstName
```

```
End Function
```

```
Sub Exercise()
```

```
    Dim FullName$
```

```
    FullName = GetFullName()
```

```
    ActiveCell.FormulaR1C1 = FullName
```

```
End Sub
```

Calling a Function in a Spreadsheet

By now, we have seen that the primary (if not the only) difference between a function and a sub procedure is that a function returns a value. Because a sub procedure does not return a value, it cannot be directly accessed from a spreadsheet and you cannot use it with the **ActiveCell.FormulaR1C1 = Value** we have been using since the previous lesson. On the other hand, since a function returns a value, you can retrieve that value and assign it to our **ActiveCell.FormulaR1C1** routine. Here is an example:

```
Function GetFullName$()
```

```
    Dim FirstName As String, LastName As String
```

```
    FirstName = "Patricia"
```

```
    LastName = "Katts"
```

```
    GetFullName = LastName & ", " & FirstName
```

```
End Function
```

```
Sub Exercise()
```

```
    Dim FullName$
```

```
    FullName = GetFullName()
```

```
    ActiveCell.FormulaR1C1 = FullName
```

```
End Sub
```

Better yet, if/when possible, you do not have to first declare a variable that would hold the value returned by a function. You can directly assign the function to the **ActiveCell.FormulaR1C1** routine. Here is an example:

```
Function GetFullName$()
```

```
    Dim FirstName As String, LastName As String
```


```

FirstName = "Patricia"
LastName = "Katts"

GetFullName = LastName & ", " & FirstName
End Function

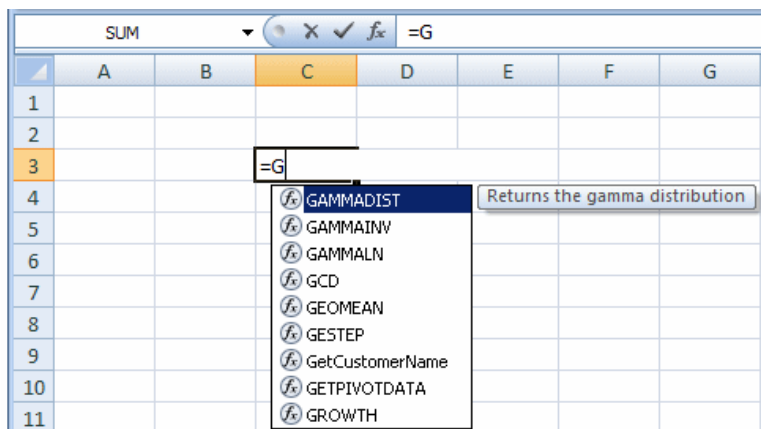
Sub Exercise()
    ActiveCell.FormulaR1C1 = GetFullName()
End Sub


```

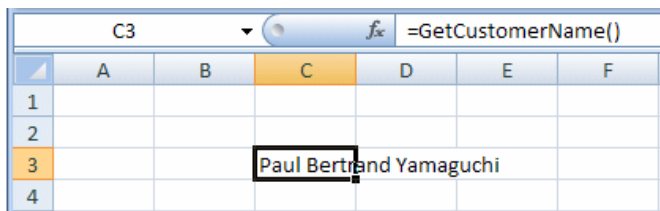
In the same way, since a function returns a value, you can use it directly in your spreadsheet. To do this, click any box in the work area. After clicking the box, type =, followed by the name of the function. As you are typing the name of the function, Microsoft Excel would present a list of functions that match that name. If you see the name of the function, you can double-click it, or you can just keep typing. After typing the name of the function, type its parentheses, and press Enter or click the Enter button  on the Formula Bar.

❖ Practical Learning: Calling a Function

1. In Microsoft Excel, click any box
2. To call the function we had created, type =G and notice the suggested list of functions:



3. If you see GetCustomerName in the list, double-click it. Otherwise, complete it with =GetCustomerName() and, on the Formula Bar, click the Enter button .



4. On the Ribbon, click Visual Basic

A Function and a Procedure

Depending on an author, in the Visual Basic language, the word "procedure" means either a sub-procedure created with the **Sub** keyword, or a function created with the **Function** keyword. In the same way, for the rest of our lessons, the word procedure will be used to represent both types. Only when we want to be precise will we use the expression "a sub-procedure" to explicitly mean the type of procedure that does not return a value. When the word "function" is used in our lessons, it explicitly refers to the type of procedure that returns a value.

Arguments and Parameters

A Review of Local and Global Variables

In the previous lesson, we saw that you could declare a global variable outside of any procedure. When using various procedures in a module, one of the characteristics of a global variable is that it is automatically accessible to other procedures:

- **Private:** A private global variable can be accessed by any procedure of the same module. No procedure of another module, even of the same program, can access it
- **Public:** A public global variable can be accessed by any procedure of its module and any procedure of another module

Based on this characteristic of the procedures of a module having access to global variables of the same module, you can declare such variables and initialize or modify them in any procedure of the same code file.

Here is an example:

```
Option Explicit
```

```
Private Length As Double
Private Width As Double
```

```
Private Sub GetLength()
    Length = 48.24
End Sub
```

```
Private Sub GetWidth()
    Width = 25.82
End Sub
```

```
Private Function CalculatePerimeter() As Double
    GetLength
    GetWidth
    CalculatePerimeter = (Length + Width) * 2
End Function
```

Introduction to Arguments

So far, to use a value in a procedure, we had to declare it. In some cases, a procedure may need an external value in order to carry its assignment. A value that is supplied to a procedure is called an argument.

When creating a procedure that will use an external value, declare the argument that represents that value between the parentheses of the procedure. For a sub procedure, the syntax you use would be:

```
Sub ProcedureName(Argument)

End Sub
```

If you are creating a function, the syntax would be:

```
Function ProcedureName(Argument) As DataType
```

```
Function Sub
```

The argument must be declared as a normal variable, omitting the **Dim** keyword. Here is an example that creates a function that takes a string as argument:

```
Function CalculatePayroll(strName As String) As Double

Function Sub
```

While a certain procedure can take one argument, another procedure can take more than one argument. In this case, in the parentheses of the procedure, separate the arguments with a comma. Here is an example of a sub procedure that takes two arguments:

```
Sub EvaluateInvoice(EmplName As String, HourlySalary As Currency)

End Sub
```

In the body of a procedure that takes one or more arguments, use the argument(s) as you see fit as if they were locally declared variables. For example, you can involve them with values inside of the procedure. You can also exclusively use the values of the arguments to perform the assignment.

❖ Practical Learning: Creating a Function With Arguments

- To create functions that take arguments, type the following

```
Option Explicit

Public Function CalculatePerimeter(Length As Double, _
    Width As Double) As Double
    Dim Perimeter As Double

    Perimeter = (Length + Width) * 2
    CalculatePerimeter = Perimeter
End Function

Public Function CalculateArea(Length As Double, Width As Double) As Double
    Dim Area As Double

    Area = Length * Width
    CalculateArea = Area
End Function
```

Calling a Procedure With Argument

The value provided for an argument is also called a parameter. To call a procedure that takes an argument, type its name. Then you have various options to access its argument(s).

Earlier, we saw that, to call a procedure, you could just use its name. After the name of the procedure, you can type the opening parenthesis "(", followed by the name of the

argument, followed by =, and the value of the argument. If the procedure takes more than one argument, separate them with commas. Here is an example:

```
Private Function GetFullName$(First As String, Last As String)
    Dim FName As String

    FName = First & Last
    GetFullName = FName
End Function

Sub Exercise()
    Dim FirstName As String, LastName As String
    Dim FullName As String

    FirstName = "Patricia "
    LastName = "Katts"

    FullName = GetFullName(FirstName, LastName)

    ActiveCell.FormulaR1C1 = FullName
End Sub
```

As mentioned previously, you can also use the **Call** keyword to call a procedure.

When you call a procedure that takes more than one argument, you must provide the values of the arguments in the order they are listed inside of the parentheses. Fortunately, you don't have to. If you know the names of the arguments, you can type them in any order and provide a value for each. To do this, in the parentheses of the procedure you are calling, type the name of the argument whose value you want to specify, followed by the := operator, and followed by the desired value for the argument. Here is an example:

```
Private Function GetFullName$(First As String, Last As String)
    Dim FName As String

    FName = First & Last
    GetFullName = FName
End Function

Sub Exercise()
    Dim FullName$

    FullName$ = GetFullName>Last:="Roberts", First:="Alan ")

    ActiveCell.FormulaR1C1 = FullName
End Sub
```

The above technique we have just seen for using the parentheses is valid for sub procedures and functions. If the procedure you are calling is a sub, you can omit the parentheses. If calling a sub procedure, after the name of the procedure, put an empty space, followed by the name of the argument assigned the desired value. Here is an example:

```
Private Sub ShowResult(ByVal Result As Double)
    Result = 145.85
End Sub

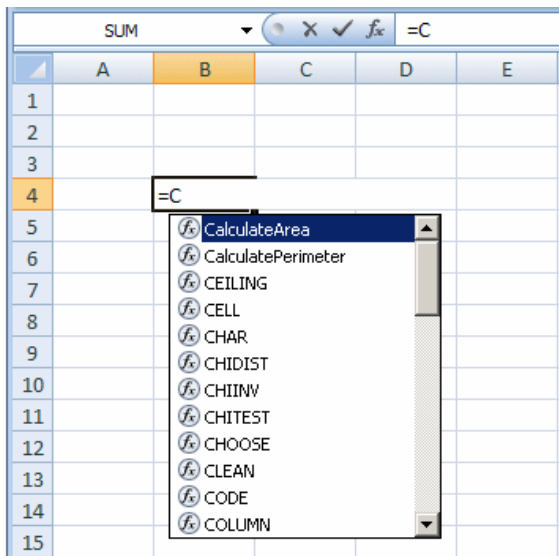
Public Sub Exercise()
    Dim Number As Double

    ShowResult Number
End Sub
```

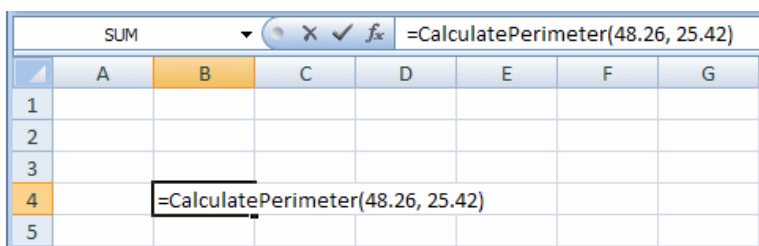
If the sub procedure is taking more than one argument, separate them with commas.

❖ Practical Learning: Calling a Procedure With Argument

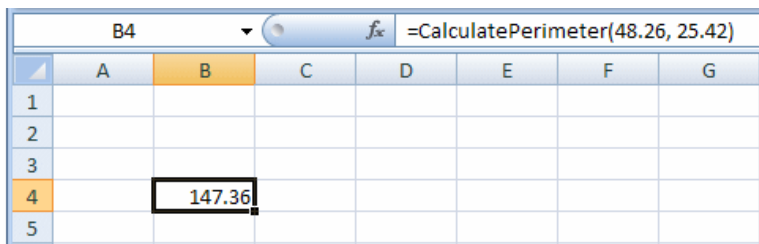
1. In Microsoft Excel, click any box
2. To call the function we had created, type =C and notice the suggested list of functions:



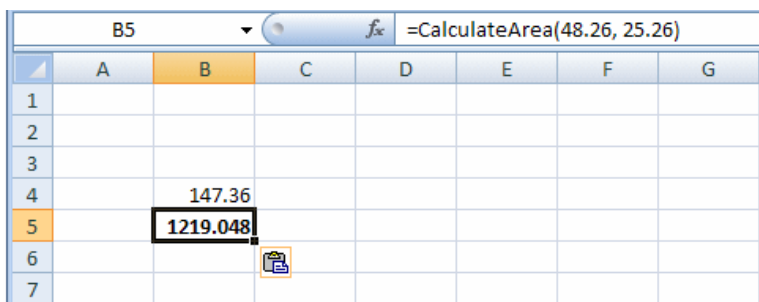
3. In the list of suggested functions, double-click CalculatePerimeter. If you don't see it, complete the typing with **=CalculatePerimeter(**
4. After the opening parenthesis, type 48.26, 25.42 as the arguments, then type the closing parenthesis ")"



5. On the Formula Bar, click the Enter button



6. Press Enter
7. Type **=CalculateArea(48.26, 25.26)** and press Enter



8. On the Ribbon, click Visual Basic

Techniques of Passing Arguments

Passing Arguments By Value

When calling a procedure that takes an argument, we were supplying a value for that argument. When this is done, the procedure that is called makes a copy of the value of the argument and makes that copy available to the calling procedure. That way, the argument itself is not accessed. This is referred to as passing an argument by value. To show this, type the **ByVal** keyword on the left side of the argument. Here are examples:

```
Private Function GetFullName$(ByVal First As String, ByVal Last As String)
    Dim FName As String

    FName = First & Last
    GetFullName$ = FName
End Function
```


If you create a procedure that takes an argument by value and you have used the **ByVal** keyword on the argument, when calling the procedure, you do not need to use the **ByVal** keyword; just the name of the argument is enough, as done in the examples on arguments so far. Here is an example:

```
Private Function GetFullName$(ByVal First As String, ByVal Last As String)
    Dim FName As String

    FName = First & Last
    GetFullName$ = FName
End Function

Sub Exercise()
    Dim FirstName As String, LastName As String
    Dim FullName As String

    FirstName = "Raymond "
    LastName = "Kouma"

    FullName = GetFullName(FirstName, LastName)

    ActiveCell.FormulaR1C1 = FullName
End Sub
```

❖ Practical Learning: Passing Arguments By Value


1. To specify that the arguments are passed by value, change the functions as follows:

```
Public Function CalculatePerimeter(ByVal Length As Double, _
    ByVal Width As Double) As Double
    Dim Perimeter As Double

    Perimeter = (Length + Width) * 2
    CalculatePerimeter = Perimeter
End Function

Public Function CalculateArea(ByVal Length As Double, _
    ByVal Width As Double) As Double
    Dim Area As Double

    Area = Length * Width
    CalculateArea = Area
End Function
```

2. To return to Microsoft Excel, on the toolbar, click the View Microsoft Excel button 

Passing Arguments By Reference

An alternative to passing arguments as done so far is to pass the address of the argument to the called procedure. When this is done, the called procedure does not receive a simple copy of the value of the argument: the argument is accessed by its address; that is, at its memory address. With this technique, any action carried on the argument will be kept by the argument when the procedure ends. If the value of the argument is modified, the argument would now have the new value, dismissing or losing the original value it had. This technique is referred to as passing an argument by reference. Consider the following code:

```
Private Sub ShowResult(ByVal Result As Double)
    Result = 145.85
End Sub

Public Sub Exercise()
    Dim Number As Double

    ShowResult Number

    ActiveCell.FormulaR1C1 = Number
End Sub
```

When the Exercise() procedure starts, a variable named Number is declared and its value is set to 0 (the default value of a newly declared Double variable). When the ShowResult variable is called, it assigns a value to the variable but since the variable is declared by value, when the procedure exits, the variable comes back with its original value, which was 0. As a result, when this code is run, the Number variable keeps its 0 value.

If you want a procedure to change the value of an argument, you can pass the argument by reference. To pass an argument by reference, on its left, type the **ByRef** keyword. This is done only when creating the procedure. When you call the procedure, don't include the **ByRef** keyword. When the called procedure finishes with the argument, the argument would keep whatever modification was made on its value. Now consider the same program as above but with arguments passed by reference:

```
Private Sub ShowResult(ByRef Result As Double)
    Result = 145.85
End Sub
```

```
Public Sub Exercise()  
    Dim Number As Double  
  
    ShowResult Number  
  
    ActiveCell.FormulaR1C1 = Number  
End Sub
```

When the Exercise() procedure starts, the Number variable is declared and its value is set to 0. When the ShowResult variable is called, it assigns a value to the variable. Since the variable is declared by reference, when the procedure exits, the variable comes back with the new value it was given. As a result, when this code runs, the Number variable has a new value.

Using this technique, you can pass as many arguments by reference and as many arguments by value as you want. As you may guess already, this technique can be used to make a sub procedure return a value, which a regular sub routine cannot do. Furthermore, passing arguments by reference allows a procedure to return as many values as possible while a regular function can return only one value.

❖ Practical Learning: Closing Microsoft Excel

1. To close Microsoft Excel, click the Office Button and click Exit Excel
2. When asked whether you want to save the file, click No

[Previous](#)

Copyright © 2008-2010 FunctionX

[Next](#)



Introduction to Objects

Classes and Objects

Introduction

The Microsoft Visual Basic language uses the concept of class to identify or manage the parts of an application. Consider an object like a house. It has such characteristics as its type (single family, townhouse, condominium, etc), the number of bedrooms, the number of bathrooms, etc:



These characteristics are used to describe a house to somebody who wants to buy it. To get such an object, you must first define the criteria that describe it. Here is an example:

```
House
[
  Address
  Type of House
  Number of Bedrooms
  Number of Bathrooms
  Has Indoor Garage
  The Living Room is Covered With Carpet
  The Kitchen Has an Island Stove
]
```

This information is used to describe a house. Based on this, House is called a class. To actually describe a real house, you must provide information for each of the above characteristics. Here is an example:

```
House: Langston
[
  Address: 6802 Leighton Ave
  Type of House: Single Family
  Number of Bedrooms: 4
  Number of Bathrooms: 3
  Has Indoor Garage: Yes
  The Living Room is Covered With Carpet: Yes
  The Kitchen Has an Island Stove: No
]
```

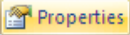
In this case, Langston is not a class anymore, it is a real house and is explicitly described. Therefore, Langston is called an object. Based on this, a class is a technique used to provide the criteria to define an object. An object is the result of a description based on a class.

❖ Practical Learning: Introducing Objects

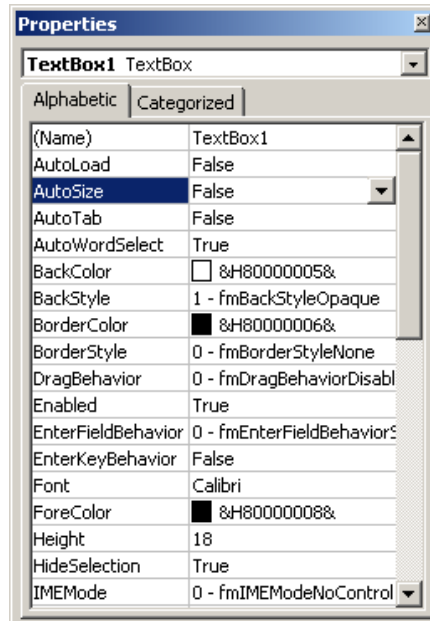
1. Start Microsoft Excel
2. On the [Ribbon](#), click Developer
3. In the Controls section, click Insert
4. Under ActiveX Controls, click any object and click the main area of the spreadsheet

In our example of a house, we used words to describe it. Examples are: Address, Type of House, Number of Bedrooms, Number of Bathrooms. In computer programming, the characteristics used to describe an object are referred to as its properties.

To display the characteristics of a Windows control, in Microsoft Excel:

- You can right-click the control and click Properties
- If the control is selected in the work area, in the Controls section of the Ribbon, click the Properties button 

Any of these two actions would display the Properties window for the control that was right-clicked:



The Properties window would stay on the screen of Microsoft Excel as long as you want. To show the properties of another control, simply click it in the work area.

If you are working in Microsoft Visual Basic, to show the characteristics of a control, right-click it and click Properties. This also would display the Properties window and show the characteristics of the selected control. While the Properties window in Microsoft Excel floats and does not hold a specific position, by default, in Microsoft Visual Basic, the Properties window is position on the lower-left side.

You can move it by dragging its title bar.

❖ Practical Learning: Introducing Properties

- Right-click the object you added and click Properties

The Methods of an Object

Introduction

While most objects only provide characteristics to describe them, other objects can perform actions. For example, a house can be used to protect people when it is raining outside. In computer programming, an action that an object can perform is referred to as method.

Earlier, we defined a House class with its properties. Unlike a property, a method must display parentheses on this right side to differentiate it from a property. An example would be:

```
House
[
    Address
    TypeOfHouse
    NumberOfBedrooms
    NumberOfBathrooms
    HasIndoorGarage
    LivingRoomCoveredWithCarpet
    KitchenHasIslandStove
    ProtectFromOutside()
]
```

When an object has a method, to access that method, type the name of the object, followed by a period, followed by the name of the method, and followed by parentheses. For example, if you

have a House object named Langston and you want to ask it to protect its inside from outside rain, you would type:

```
Langston.ProtectFromOutside()
```

This is also referred to as *calling* a method.

Methods and their Arguments

When asked to perform an action, a method may need one or more values to work with. If a method needs a value, such a value is called an argument. While a certain method may need one argument, another method would need more than one. The number of arguments of a method depends on its goal. The arguments of a method are provided in parentheses.

Suppose you have a House object and you want it to protect what is inside. There may be different reasons why the inside needs to be protected: may be from the rain, may be from the windy dust, may be at night time from too much light that prevents from sleeping, etc. Based on this, you may have to provide additional information to indicate why or how the inside should be protected. For this reason, when such a method is called, this additional information must be provided, in the parentheses of the method. Here is an example:

```
House
[
    Address
    TypeOfHouse
    NumberOfBedrooms
    NumberOfBathrooms
    HasIndoorGarage
    LivingRoomCoveredWithCarpet
    KitchenHasIslandStove
    ProtectFromOutside(Reason)
]
```

As mentioned above, a method can be created to take more than one argument. In this case, the arguments are separated with commas. Here is an example:

```
House
[
    Address
    TypeOfHouse
    NumberOfBedrooms
    NumberOfBathrooms
    HasIndoorGarage
    LivingRoomCoveredWithCarpet
    KitchenHasIslandStove
    ProtectFromOutside(Reason, WhenToProtect)
]
```

The arguments are used to assist the object with performing the intended action. Once a method has been created, it can be used. Once again, using a method is referred to as calling it. If a method takes one argument, when calling it, you must provide a value for the argument, otherwise the method would not work.

To call a method that takes an argument, type the name of the method followed by the opening parenthesis “(”, followed by the value that will be the argument, followed by a closing parenthesis “)”. The argument you pass can be a regular constant value or it can be the name of another object.

If the method is taking more than one argument, to call it, type the values for the arguments, in the exact order indicated, separated from each other by a comma.

Default Arguments

We have mentioned that, when calling a method that takes an argument, you must supply a value for the argument. There is an exception. Depending on how the method was created, it may be configured to use its own value if you fail, forget, or choose not, to provide one. This is known as the default argument. Not all methods follow this rule.

If a method that takes one argument has a default value for it, then you don't have to supply a value when calling that method. Such an argument is considered optional.

If a method takes more than one argument, some argument(s) may have default values while some others do not. The arguments that have default values can be used and you don't have to supply them.

We will mention default arguments when we come to a method that takes some.

Me

In previous lessons and sections, we saw that an object was made of properties and methods. We also saw how to access a property of an object. For example, imagine you have a House class defined as follows:

```
House
[
    Address
    TypeOfHouse
    NumberOfBedrooms
    NumberOfBathrooms
    HasIndoorGarage
    LivingRoomCoveredWithCarpet
    KitchenHasIslandStove
    ProtectFromOutside()
]
```

If you have an object named Camden and that is of type House. To access some of its properties, you would use code as follows:

```
Camden.Address
Camden.TypeofHouse
```

If you are working inside of a method of the class, for example if you are working in the body of the ProtectFromOutside method, you can also access the properties the same way, this time without the name of the object. This could be done as follows:

```
ProtectFromOutside()
    Address
    TypeofHouse
    NumberOfBedrooms
    NumberOfBathrooms
End
```

When you are accessing a member of a class inside of one of its own methods, you can precede that member with the Me object. You must include the period operator between Me and the member of the class. Here is an example:

```
ProtectFromOutside()
    Me.Address
    Me.TypeofHouse
    Me.NumberOfBedrooms
    Me.NumberOfBathrooms
End
```

Remember that the **Me** object is used to access the members of an object while you are inside of another member of the object.

With

We have seen that you can use the name of an object to access its members. Here is an example:

```
Camden.Address
Camden.TypeOfHouse
Camden.NumberOfBedrooms
Camden.NumberOfBathrooms
Camden.HasIndoorGarage
```

Instead of using the name of the object every time, you can start a section with the **With** keyword followed by the name of the object. In another line, end the section with the End With expression:

```
With Camden
End With
```

Between the With and the End With lines, to access a member of the class that the object is built from, type a period followed by the desired member. This would be done as follows:

```
With Camden
    .Address
    .TypeOfHouse
    .NumberOfBedrooms
    .NumberOfBathrooms
    .HasIndoorGarage
End With
```

As you access a member, you can perform on it any action you judge necessary.

The Properties window allows you view or change a characteristic of the control. The properties of an object can be changed when designing it or by writing code. The time you are designing an application is referred to as design time. The time the application (form) displays to the user is referred to as run time.

You can manipulate the characteristics of a control both at design and at run times. This means that you can set some properties at design time and some others at run time.

❖ Practical Learning: Closing Microsoft Excel

1. To close Microsoft Excel, click the Office Button and click Exit Excel
2. When asked whether you want to save the file, click No

[Previous](#)

Copyright © 2008-2010 FunctionX, Inc.

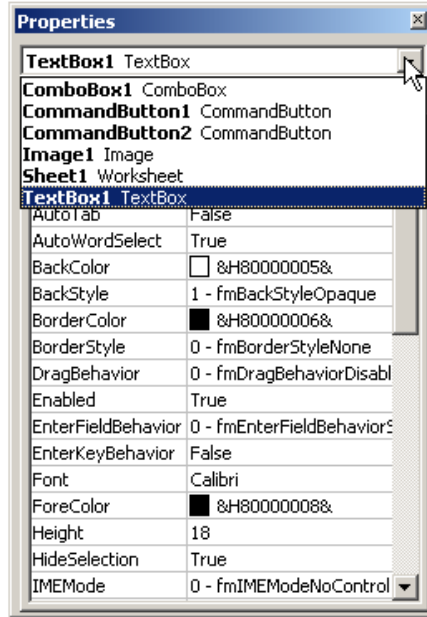


The Properties Window

The Appearance of the Properties Window

Introduction

To manipulate an object, you can use the Properties window:



❖ Practical Learning: Introducing Properties

1. Start Microsoft Excel
2. On the Ribbon, click Developer
3. In the Controls section, click Insert
4. Under ActiveX Controls, click any object and click the main area of the spreadsheet
5. Right-click the object you added and click Properties

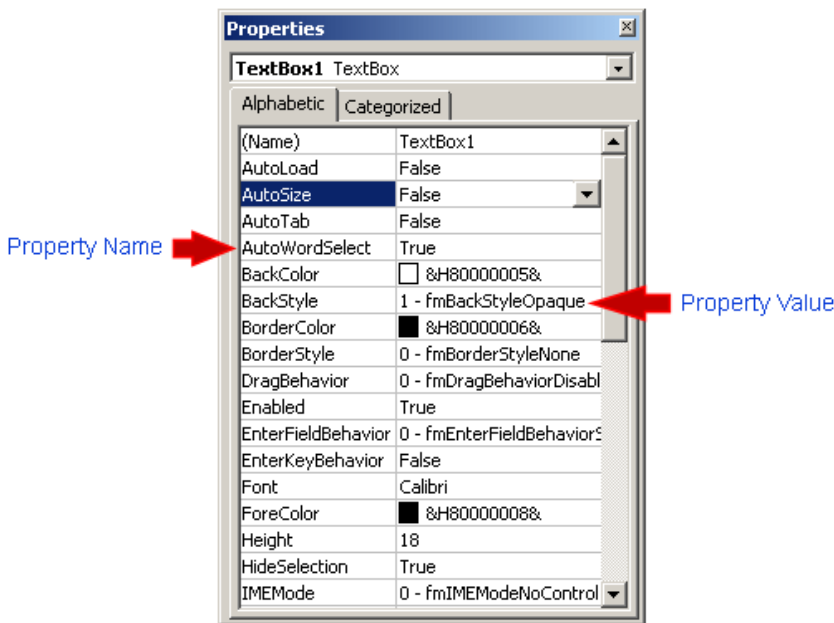
The Name of a Property

After adding a control to your application, you can manipulate its characteristics. If you are working in Microsoft Excel, to put a control into edit mode, in the Controls section of the [Ribbon](#),



click the Design Mode button

Each field in the Properties window has two sections: the property's name and the property's value:



The name of a property is represented in the left column. This is the official name of the property. Notice that the names of properties are in one word. Based on this, our House class would have been defined as follows:

```
House
[
    Address
    TypeOfHouse
    NumberOfBedrooms
    NumberOfBathrooms
    HasIndoorGarage
    LivingRoomCoveredWithCarpet
    KitchenHasIslandStove
]
```

You can use this same name to access the property in code.

Accessing a Control's Property

To access a property of a control using code, type the name of the control, followed by a period, followed by the name of the property. Based on this, if you have a House object named Langston, to access its TypeOfHouse property, you would write:

```
Langston.TypeOfHouse
```

The Value of a Property

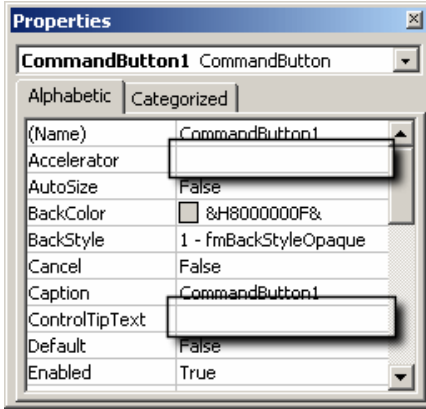
The box on the right side of each property name represents the value of the property that you can set for an object. There are various kinds of fields you will use to set the properties. To know what particular kind a field is, you can click its name. To set or change a property, you use the box on the right side of the property's name: the property's value, also referred to as the field's value.

The Default Value of a Control's Property

To programmatically change the value of a property, type the name of the control, followed by a period, followed by the name of the property, followed by =. Then, on the right side of equal, you must provide the value but this depends on the type of value.

The people who developed the controls also assigned some primary values to their properties. This is the type of value that a property either is most likely to have or can use unless you decide to change it. The primary value given to a property is referred to as its default value. Most of the time, you can use that property. In some other assignments, the default value will not be suitable.

Empty Fields



By default, these fields don't have a default value. Most of these properties are dependent on other settings of project.

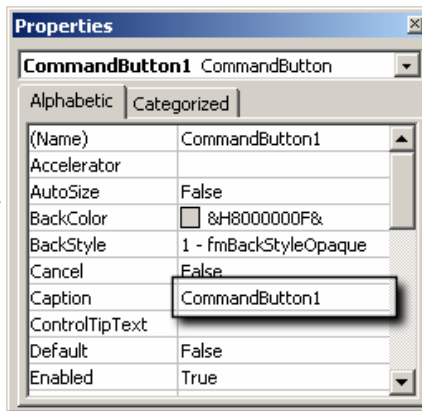
To set the property on such a field, you can type in it or sometimes you will need to select from a list.

Text Fields

There are fields that expect you to type a value. Most of these fields have a default value.

To change the value of the property, click the name of the property, type the desired value, and press Enter or Tab. While some properties, such as the **Caption**, would allow anything, some other fields expect a specific type of text, such as a numeric value.

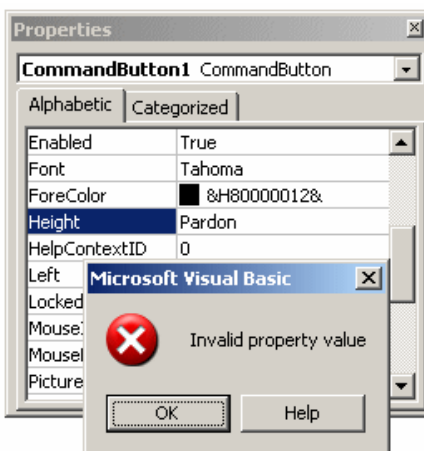
To programmatically change the value of a text-based property, on the right side of the = sign, you can type the value in double quotes. For example, suppose you have a House object named Langston. If you want to specify its address, you would write:



```
Langston.Address = "6802 Leighton Ave"
```

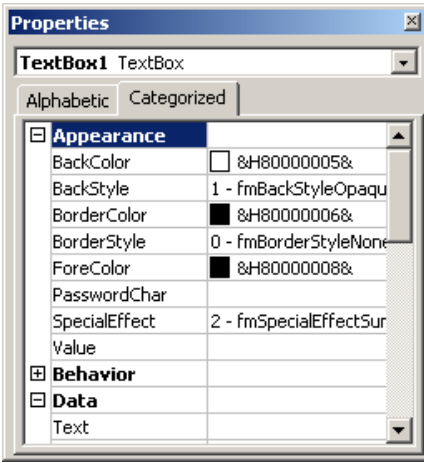
Numeric Fields

Some fields expect a numeric value. In this case, you can click the name of the field and type the desired value. If you type an invalid value, you would receive a message box notifying you of the error:



When this happens, click OK and type a valid value. If the value is supposed to be an integer, make sure you don't type it with a fractional part.

Expandable Fields



Some fields have a - or a + button. This indicates that the property has a set of sub-properties that actually belong to the same property and are defined together. To expand such a field, click its + button and a “” button will appear.

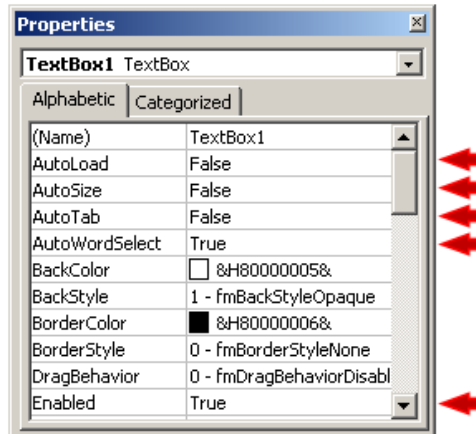
To collapse the field, click the “” button. Some of the properties are numeric based. With such a property, you can click its name and type the numeric value. Some other properties are created from a sub-list. If you expand such a field, it would display various options. With such a property, you should select from a list.

Boolean Fields

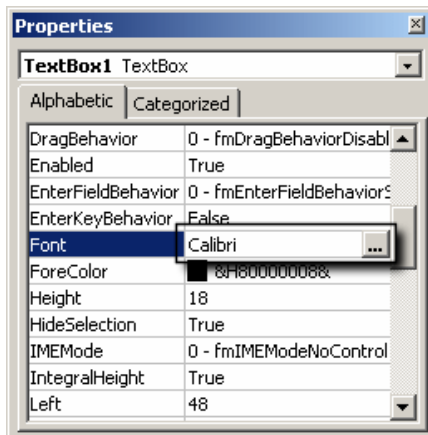
Some fields can have only a **True** or **False** value. These are Boolean fields. To change their value, you can either select from the combo box or double-click the property to switch to the other value.

To programmatically specify the value of a Boolean property, on the right side of the = symbol, type **True** or **False**. Here is an example:

```
Langston.HasIndoorGarage= True
```



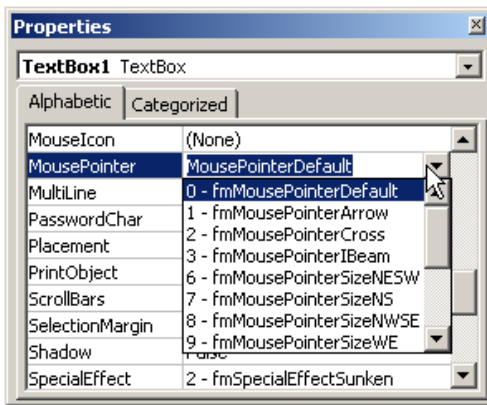
Intermediary Fields



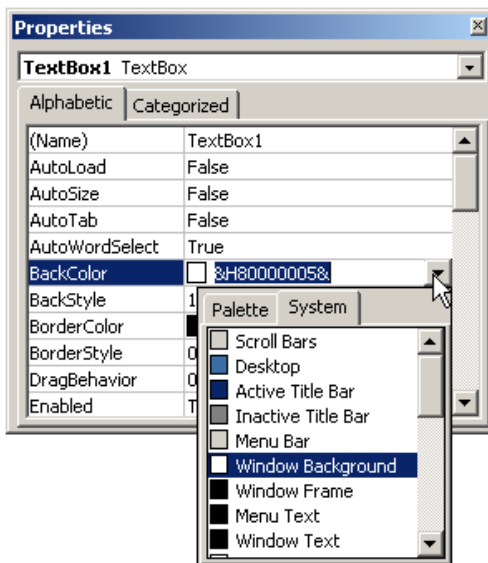
Some fields use a value that can be set through an intermediary action. Such fields display a browse button "...". When you click the button, a dialog box would come up and you can set the value for the field.

List-Based Fields

To change the value of some of the fields, you would first click the arrow of their combo box to display the available values. After clicking the arrow, a list would display:



There are various types of list-based fields. Some of them display just two items. To change their value, you can just double-click the field. Some other fields have more than two values in the list. To change them, you can click their arrow and select from the list. You can also double-click a few times until the desired value is selected. Some other items would display a window from where you would click the option you want:



To programmatically specify the value of a list-based property, you must use one from a list. For example, suppose you had defined a list of types of house as `tpeSingleFamily`, `tpeTownHouse`, and `tpeCondominium`. To use one of these values for a House object named `Langston`, you would type:

```
Langston.TypeOfHouse = tpeSingleFamily
```

In most cases, each member of such a list also uses a natural number. An example would be:

TypeOfHouse	Value
0	tpeSingleFamily
1	tpeTownHouse
2	tpeCondominium

Although we used 0, 1, and 2 in this list, there are no predefined rules as to the number allocated for each member of the list. The person who created the list also decided what number, if any, each member of the list would have (if you are curious, in most programming languages or libraries, these types of properties are created using an enumeration (in C++ or in the the .NET Framework) or a set (Borland VCL)). Based on this, the above code would also be written as:

```
Langston.TypeOfHouse = 0
```

❖ Practical Learning: Closing Microsoft Excel

1. To close Microsoft Excel, click the Office Button and click Exit Excel
2. When asked whether you want to save the file, click No



Introduction to Forms

Forms Fundamentals

Introduction to Forms

A computer application, such as those that run on Microsoft Windows, is equipped with objects called Windows controls. These are the objects that allow a person to interact with the computer.

The primary control used on most applications is called a form.

❖ Practical Learning: Introducing Forms

1. Start Microsoft Excel
2. On the [Ribbon](#), click Developer
3. In the Code section, click Visual Basic

Creating a Form

To create a form, on the main menu of Microsoft Visual Basic, you can click Insert -> UserForm. This would add a form to your project. In the same way, you can add as many forms as you want.

The form is primarily used as a platform on which you add other controls. For this reason, a form is referred to as a container. By itself, a form is not particularly useful. You should add other objects to it.

When you create or add a form, a module is also automatically created for it. To access the module associated with a form, you can right-click the form and click View Code.

❖ Practical Learning: Creating a Form

1. To create a form, on the main menu, click Insert -> UserForm
2. To access its associated module, right-click the form and click View Code
3. To return to the form, on the main menu, click Window and click the menu item that has (UserForm).


Using a Form

Showing a Form

Although you create a form in Microsoft Visual Basic, you view its results in Microsoft Excel. You have various options.

A form is referred to as modal if the user cannot access anything from the same application while the form is displaying. A form is called modeless if the user can click something of the same application behind that form while the form is displaying.

To display the run-time view of a form in modal view:

- While in Microsoft Visual Basic, you can press F5
- On the main menu of Microsoft Visual Basic, you can click Run -> Run Sub/UserForm
- On the Standard toolbar of Microsoft Visual Basic, you can click the Run Sub/UserForm button . This would send the form to Microsoft Excel and display it in the normal view

You can also programmatically display a form. To support this, the UserForm object is equipped with a method named **Show**. Its syntax is:

```
Public Sub UserForm.Show(Optional ByVal Modal As Boolean)
```

This method takes one Boolean argument that is optional. If you omit it, the form would display as modal and the user cannot do anything else in Microsoft Excel as long as the form is not closed. That's the default behavior. If you want to display the form as modeless, pass the argument as False. Here is an example:

```
Private Sub Exercise()  
    UserForm1.Show False  
End Sub
```

Printing a Form

If you have equipped a form with many aesthetic objects you want to see on a piece of paper, you

can print it. To support printing, the UserForm object is equipped with a method named PrintForm. Its syntax is:

```
Public Sub PrintForm
```

This method takes no argument. When called, it sends the (body of the) form directly to the printer. Here is an example of calling it:

```
Private Sub Exercise()  
    UserForm1.PrintForm  
End Sub
```

Hiding a Form

As opposed to displaying a form, if it is already showing, you can hide it. To allow you to hide a form, the UserForm object is equipped with a method named Hide. Its syntax is:

```
Public Sub UserForm.Hide
```

This method takes no argument. When called, it hides the form (without closing it). Here is an example of calling it:

```
Private Sub Exercise()  
    UserForm1.Hide  
End Sub
```

Closing a Form

After using a form, the user can close it by clicking the system close button. To programmatically close a form, use the End statement. Here is an example:

```
Private Sub Exercise()  
    End  
End Sub
```

The Characteristics of a Form

The Name of a Form

Like every object on a computer, the primary characteristic of a form is its name. After creating a form, access its Properties window, click (Name), type the desired name and press Enter

❖ Practical Learning: Naming a Form

- If the Properties window is not displaying, right-click the form and click Properties window. In the Properties window, click (Name) and type frmCleaningOrders

The location of a Form

When a form displays in normal view to the user, it is usually centered. The user can then drag its title bar to move it around. You too can move a form.

If you want to specify the position a form would assume when it displays, at design time, access its Properties window. Then change the values of the Left and the Top properties. You can also programmatically control the location of a form. You have two options. You can assign the desired values to its **Left** and/or **Top** properties. Here is an example:

```
Private Sub Exercise()  
    UserForm1.Left = 400  
End Sub
```

Al alternative is to call the **Move** method. Its syntax is:

```
Public Sub UserForm.Move(ByVal Left As Single, ByVal Top As Single, Optional ...)
```

This method takes two required arguments. The first represents the left distance and the second is the top distance. Here is an example:

```
Private Sub Exercise()  
    UserForm1.Move 200, 200  
End Sub
```

The Size of a Form

When you have just added a new form, it assumes a default size. If that size doesn't fit your intentions, you can change.

To change the size of a form, you can access its Properties window. Then change the values of the Height and Width. To programmatically change the size of a form, you have many options. You can assign the desired values to its Height and/or to its Width. Here is an example:

```
Private Sub Exercise()  
    UserForm1.Width = 600  
End Sub
```

Another option is to call the Move method. In reality, this method takes two required arguments

and two optional arguments. Its actual syntax is:

```
Public Sub UserForm.Move(ByVal Left As Single, ByVal Top As Single, _  
    Optional ByVal Width As Single, Optional ByVal Height As Single)
```

The last two optional arguments allow you to specify the size of the form. Here is an example:

```
Private Sub Exercise()  
    UserForm1.Move 200, 200, 1040, 600  
End Sub
```

❖ Practical Learning: Resizing a Form

1. Position the mouse in the lower-right section of the form
2. Click and drag right and down
3. Return to Microsoft Excel
4. To close Microsoft Excel, click the Office Button and click Exit Excel
5. When asked whether you want to save the file, click No

[Previous](#)

Copyright © 2008-2010 FunctionX, Inc.

[Next](#)



Introduction to Controls

Controls Fundamentals

Introduction

By itself, a form means nothing. Its role is revealed in the objects it holds. You can add such objects to a form or the body of a spreadsheet.

❖ Practical Learning: Introducing Properties

1. Start Microsoft Excel
2. On the Ribbon, click Developer
3. In the Code section, click Visual Basic

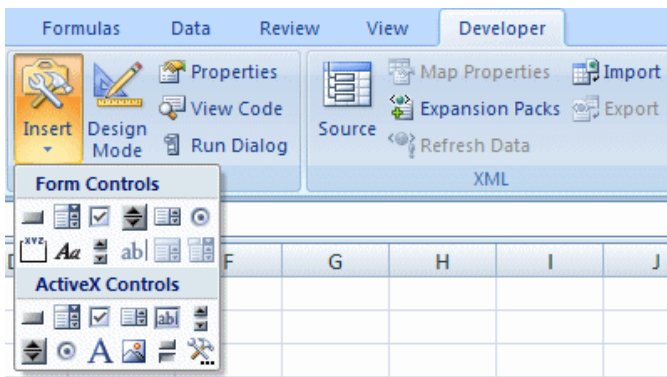
The Spreadsheet

When working in Microsoft Excel, you can use Windows controls either on the work area or in a form in Microsoft Visual Basic. Therefore, just like a form, a spreadsheet also is a container of controls.

Introduction to Windows Controls

The main objects used to help a person interact with the computer are Windows controls. There are two main ways you can access these objects when working with Microsoft Excel.

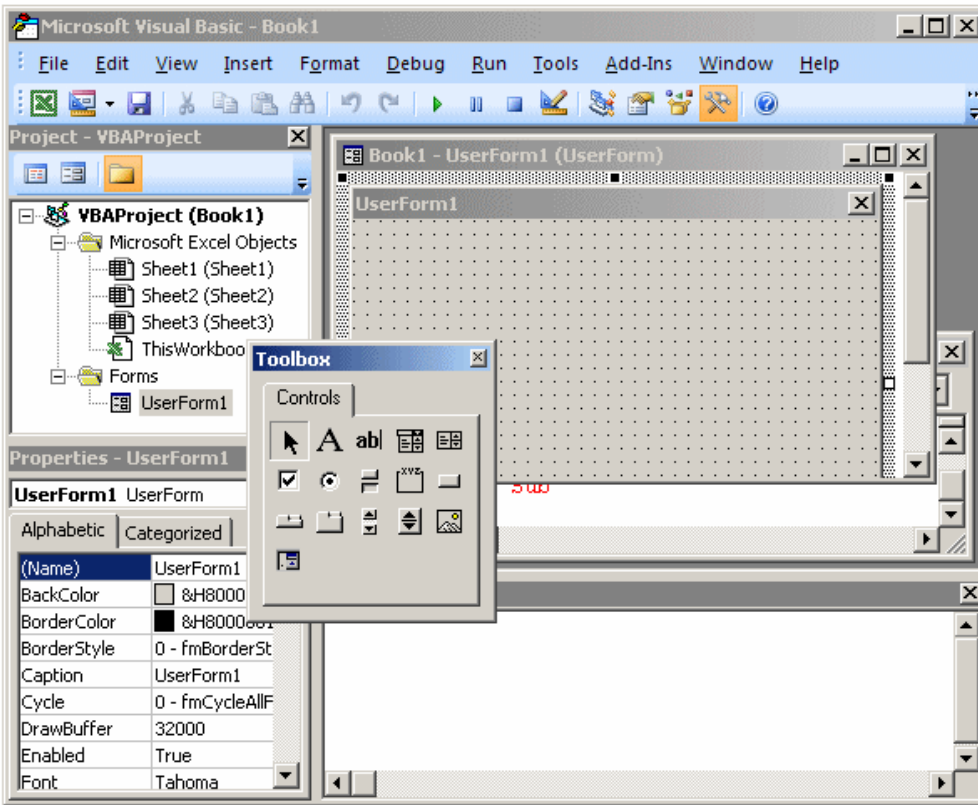
If you are working in Microsoft Excel, you can add or position some Windows controls on the document. To do that, on the Ribbon, click Developer. In the Control section, click Insert:



This would display the list of controls available in Microsoft Excel. The controls appear in two sections: Form Controls and ActiveX Controls. If you are working on a spreadsheet in Microsoft Excel, you should use only the controls in the ActiveX Controls section. If you are working on a form in Microsoft Visual Basic, a Toolbox equipped with various controls will appear.

❖ Practical Learning: Accessing Windows Controls

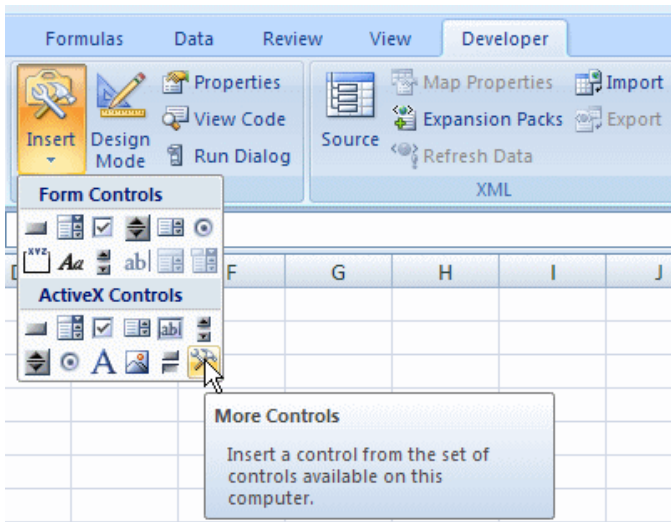
1. To create a form, on the main menu, click Insert -> UserForm
2. To access its associated module, right-click the form and click View Code
3. To return to the form, on the main menu, click Window and click the menu item that has (UserForm).
4. In Microsoft Visual Basic, notice that a Toolbox appears next to the form



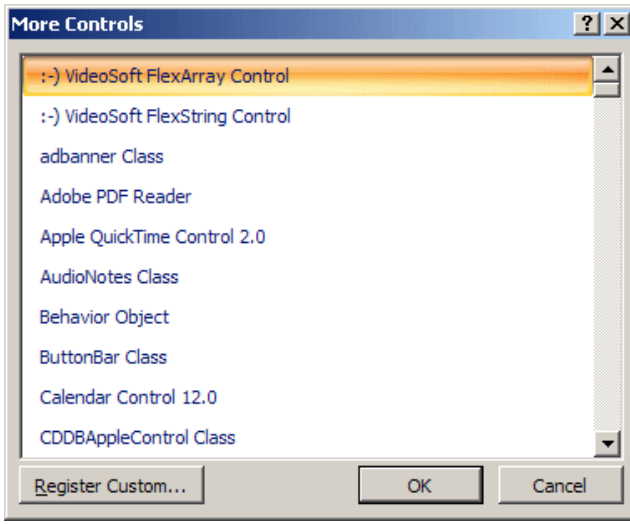
5. To return to Microsoft Excel, on the Taskbar, click Microsoft Excel
6. To display the controls, in the Controls section of the [Ribbon](#), click Insert

Using Additional Objects

The Developer tab of the Ribbon in Microsoft Excel provides the most regularly used controls. These controls are enough for any normal spreadsheet you are developing. Besides these objects, other controls are left out but are still available. To use one or more of these left out controls, in the Controls section of the Ribbon, click Insert and click the More Controls button:

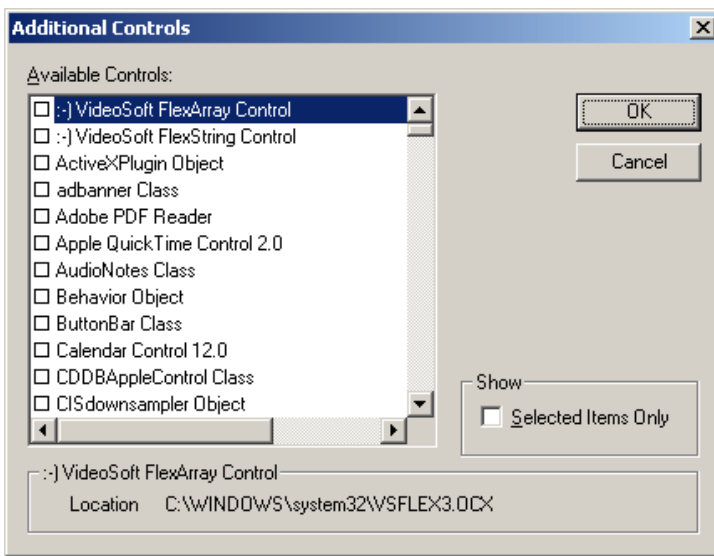


This would open the More Controls dialog box:



You can scroll up and down in the window to locate the desired control. If you see a control you want to use, click it and click OK.

In Microsoft Visual Basic, to access more controls, on the main menu, you can click Tools -> Additional Controls... This would open the Additional Controls dialog box:



To select a control, you can click its check box. After selecting the controls, click OK.

The Names of Controls

Every control in the Developer tab of the Ribbon or in the Toolbox of Microsoft Visual Basic has a specific name. You may be familiar with some of these controls. If you are not sure, you can position the mouse on a control and a tool tip would come up. In our lessons, we will use the tool tip of a control to name it. The names we will use are:

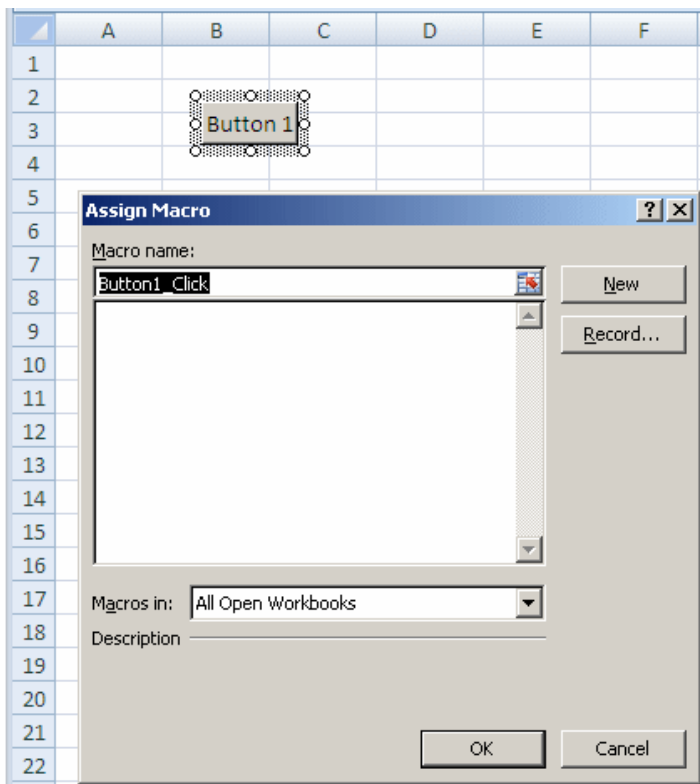
ActiveX Controls	Name	Forms Controls	Name
	Command Button		Label
	Combo Box		Toggle Button
	Check Box		
	List Box		TabStrip
	Text Box		MultiPage
	Scroll Bar		ScrollBar
	Spin Button		Text Box
	Option Button		Image
	Label		RefEdit
	Image		Frame
	Toggle Button		

Adding a Control to a Container

Adding One Control

To use one of the controls from the Ribbon or the Toolbox, you can click it. If you then simply click its container, the control would be added where you clicked and with some default dimensions.

In Microsoft Excel, if you click the button control in the Form Controls section and click the work area, just after the control has been added the Assign Macro dialog box would come up:



In our lessons, when working in Microsoft Excel, we will avoid using the objects in the Form Controls section.

If you want, instead of clicking and releasing the mouse, you can click and drag. This allows you to "draw" the control and give the dimensions of your choice. If the control has already been added but you want it to assume different dimensions, you can click it to select it and then drag one of its border handles.

To programmatically add a control to a spreadsheet, use the following formula:

```
Private Sub Exercise()  
    Worksheets(1).OLEObjects.Add "Forms.ControlName.1"  
End Sub
```

The only thing you need to know and change about this formula is the *ControlName* factor. We will learn about Worksheets(1) in **Lesson 12**. Use the following names:




Use this Name	To Get a		Use this Name	To Get
CheckBox	Check Box		ComboBox	Combo Box
CommandButton	Command Button		Label	Label
ListBox	List Box		Image	Image
OptionButton	Option Button		ScrollBar	Scroll Bar
SpinButton	Spin Button		TextBox	Text Box
ToggleButton	Toggle Button			

Here is an example that creates and positions a text box on a spreadsheet:

```
Private Sub Exercise()  
    Worksheets(1).OLEObjects.Add "Forms.TextBox.1"  
End Sub
```

Adding Many Controls

The above technique is used to add one control at a time. If you want to add the same control again, you must click it on the Ribbon or in the Toolbox and click its container again. If you plan to add the same control many times, in the Toolbox of Microsoft Visual Basic, double-click the control and click the form as many times as necessary. When you have reached the desired number of copies of that control, to dismiss it, in the Toolbox, click the same control again, click another control, or click the Select Objects button.

1. To add a control to the document, Under ActiveX Controls, click the Command Button button  and click a box (any box in the work area)
2. To add another control, in the Controls section of the Ribbon, click the arrow under Insert and click the Combo Box button  and click somewhere in the work area away from the previously added button
3. On the Taskbar, click Microsoft Visual Basic to return to it
4. On the Toolbox, click the CommandButton and click somewhere in the top-left section of the form (no need for precision at this time)
5. On the Toolbox, click the ComboBox and click somewhere in the middle-center section of the form (no need for precision at this time)
6. On the Toolbox, click the CheckBox and click somewhere in the lower-right section of the form (no need for precision at this time)
7. To return to Microsoft Excel, click the View Microsoft Excel button 


Control Selection

Single Control Selection

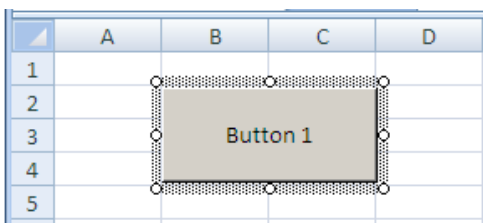
After you have added a control to a container, in order to perform any type of configuration on the control, you must first select it. Sometimes you will need to select a group of controls.

To select a control in the work area in Microsoft Excel, first, in the Controls section

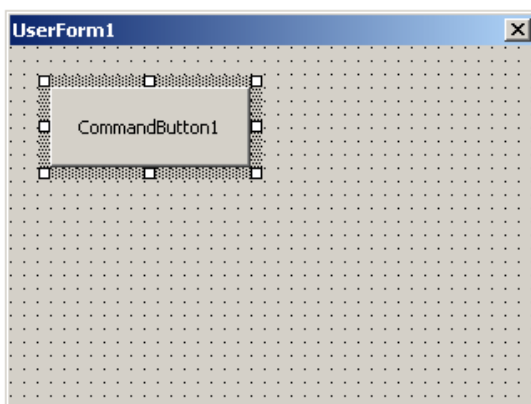


of the Ribbon, click the Design Mode button . After clicking it, right-click the control and press Esc. If you are working in Visual Basic, to select a control, click it on the form.

In Microsoft Excel, when a control is selected, it is surrounded by 8 small circles, also called handles. Here is an example:



In Microsoft Visual Basic, when a control is selected, it is surrounded by 8 small squares:



❖ Practical Learning: Selecting Controls

1. Position the mouse on CommandBut that was positioned on the form and click. Notice that you are able to select the button



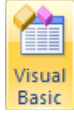
2. In the Controls section of the Ribbon, click the Design Mode button

3. In the work area, click the CommandBut button again
4. Click the combo box. Notice that, this time, you cannot select the controls
5. To return to controls to edit mode, in the Controls section of the Ribbon, click



the Design Mode button

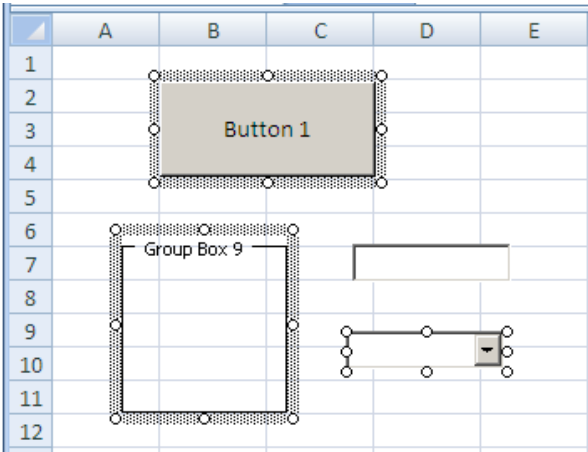
6. To return to Microsoft Visual Basic, in the Code section of the Ribbon, click the



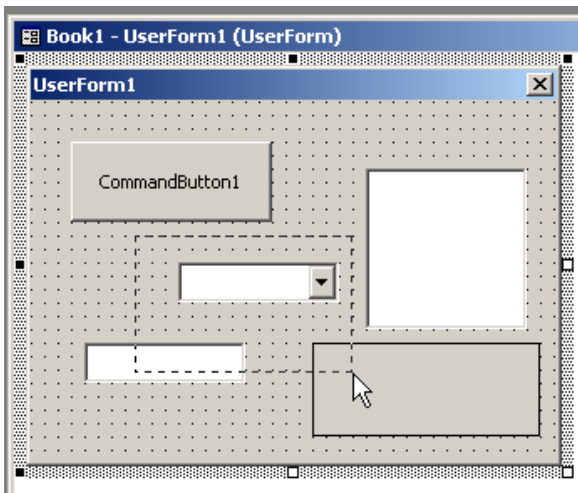
Visual Basic button

Multiple Control Selection

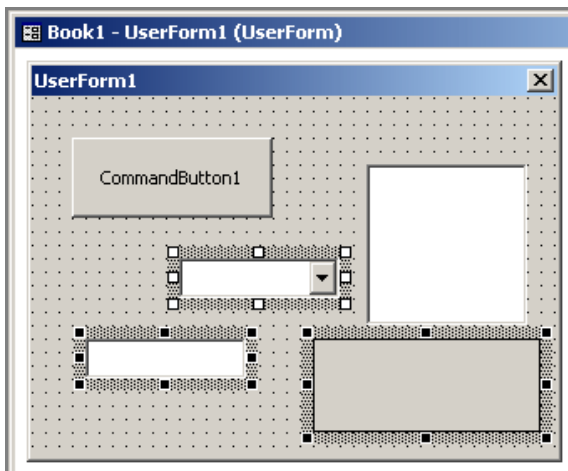
To select more than one control on a spreadsheet, click the first. Press and hold Shift. Then click each of the desired controls. If you click a control that should not be selected, click it again. After selecting the group of controls, release Shift:



If you are working on a form in Microsoft Visual Basic, first click one of the controls you want to select, then press and hold Ctrl. Click each of the desired controls. If you click a control that should not be selected, click it again. After selecting the group of controls, release Ctrl that you were holding. As another technique you can use to select various controls, click an unoccupied area on the form, hold the mouse down, drawing a fake rectangle that would either include each of the desired controls or would touch each, then release the mouse:



Every control touched by the fake rectangle or included in it would be selected:



When a group of controls is selected, the first selected control displays 8 handles but its handles are white while the others are dark.

❖ Practical Learning: Selecting and Using Various Controls


1. On the form, click one of the controls
2. Press and hold Ctrl
3. Click one of the other controls
4. Release Ctrl
5. To dismiss the selection, press Esc

Control Deletion

If there is a control on your form or your work area but you don't need it, you can remove it. To delete a control, first select it and then press Delete. You can also right-click a control and click Cut.

To remove a group of controls, first select them, then press Delete or right-click the selection and click Cut.

❖ Practical Learning: Deleting Controls

1. While still in Microsoft Visual Basic, press Ctrl + A to select all controls
2. Press Delete to remove them
3. To display them again, press Ctrl + Z
4. To return to Microsoft Excel, click the View Microsoft Excel button 

[Previous](#)

Copyright © 2004-2009 FunctionX

[Next](#)



Form and Control Design

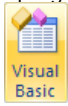
The Location of a Control on a Form

Introduction

In the previous lesson, we saw that a characteristic, also called a property, of a control is information used to describe or define an object. The description can be done visually or programmatically. Some of the visual description is done by designing the object; that is, by changing its aspects using the mouse, the keyboard, and the tools provided by Microsoft Excel and Microsoft Visual Basic.

❖ Practical Learning: Introducing Controls Design

1. Start Microsoft Excel and, on the [Ribbon](#), click the Developer tab
2. To launch the programming environment, in the Code section of the Ribbon, click the Visual



Basic button

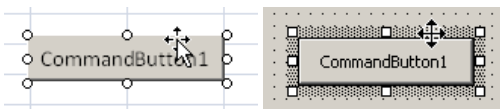
3. To create a form, on the main menu of Visual Basic, click Insert -> UserForm
4. On the Toolbox, click the CommandButton and click somewhere on the form (no need for precision)
5. On the Toolbox, click ComboBox and click the form away from the previously added CommandButton

Moving a Control

When you add a control to the work area in Microsoft Excel or to a form in Microsoft Visual Basic, it assumes a position on its container. If you want, you can change that location by moving the control.

To move a control, click it and drag in the direction of your choice. To move a group of controls, first select them. Click it and drag the selection in the direction of your choice

When a control has been selected, as your mouse moves over it, its pointer displays a different cursor. One of these cursors can be used to move a control. This cursor is represented as a cross with four arrows:



To move a control, click its border and hold the mouse down, drag in the direction of your choice. When you get to the desired location, release the mouse.

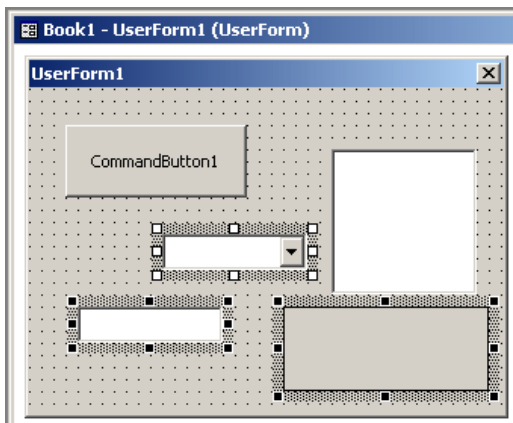
Control Centering Towards the Middle of the Form

You can also position one or more controls in the middle of the form. To do that, select the control, then, on the main menu of Visual Basic, click Format -> Center In Form -> Vertically.

Aligning Controls

Horizontal Alignment

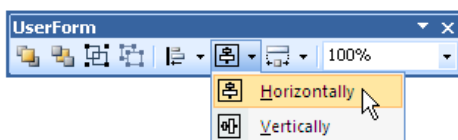
When many controls are selected on a form, one of the controls has dark handles:



In our descriptions, the control with the dark handles will be referred to as the base control.

During form design, to better position the controls, you can use the main menu with the Format group. Microsoft Visual Basic also provides the UserForm toolbar to assist you. To display the UserForm toolbar, on the main menu of Microsoft Visual Basic, you can click View -> Toolbars -> UserForm.

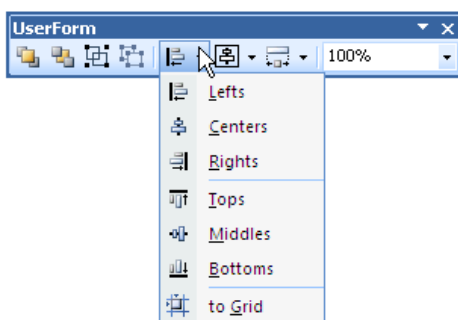
If you have a certain control on the form and want to position it exactly at equal distance between the left and the right borders of the form, select the control, then, on the main menu of Microsoft Visual Basic, click Format -> Center in Form -> Horizontally. To get the same option, on the UserForm toolbar, click the arrow of the Center button and click Horizontally:



Horizontal alignment affects controls whose distance from the left border of the form must be the same. To perform this type of alignment, you can use the main menu where you would click Format -> Align, and select one of the following options:

- **Lefts:** All selected controls will have their left border coincide with the left border of the base control
- **Centers:** The middle handles of the selected controls will coincide with the middle handles of the base control
- **Rights:** All selected controls will have their right border coincide with the right border of the base control

To get the same options using the UserForm toolbar, click the arrow of the Align button and select the desired option: **Lefts**, **Centers**, or **Rights**:



Vertical Alignment

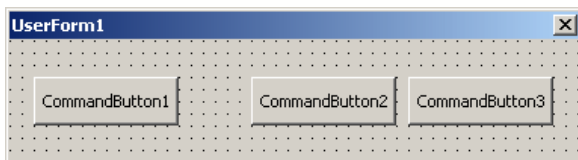
Another option you have consists of moving controls up or down for better alignment. Once again you must first select the controls. Then, on the main menu, click Format -> Align, and click one of the following options:

- **Tops:** All selected controls will have their top border coincide with the top border of the base control but their left border would have the same distance with the left border of the parent
- **Middles:** The top handles of the selected controls will align vertically with the top handle of the base control
- **Bottoms:** All selected controls will have their bottom border coincide with the bottom border of the base control but their left border would have the same distance with the left border of the parent

To get the same options using the UserForm toolbar, click the arrow of the Align button and select the desired option: **Tops**, **Middles**, or **Bottoms**.

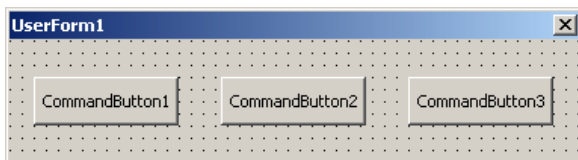
Horizontal Spacing and Alignment

Suppose you have a group of horizontally aligned controls as follows:

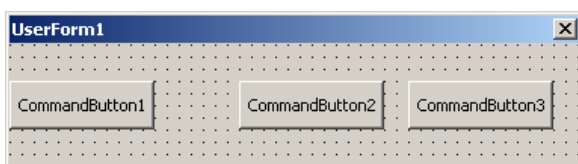


Obviously the controls on this form are not enjoying the most professional look. The Format group of the main menu allows you to specify a better horizontal alignment of controls with regards to each other. To use it, first select the controls. Then, on the main menu of Microsoft Visual Basic, click Format -> Horizontal Spacing, and click one of the following options:

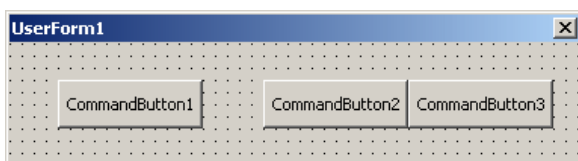
- **Make Equal:** Microsoft Visual Basic will calculate the horizontal distances that separate each combination of two controls and find their average. This average is applied to the horizontal distance of each combination of two controls:



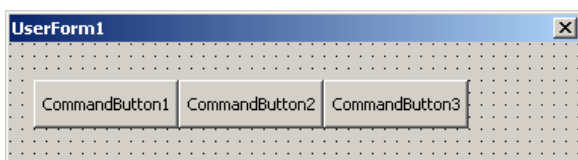
- **Increase:** Microsoft Visual Basic will move each control horizontally, except the base control (the control that has white handles) by one unit away from the base control. This will be done every time you click the Increase Horizontal Spacing button or the Format -> Horizontal Spacing -> Increase menu item



- **Decrease:** Microsoft Visual Basic will move each control horizontally, except the base control (the control that has white handles) by one unit towards the base control. This will be done every time you click the Decrease Horizontal Spacing button or the Format -> Horizontal Spacing -> Decrease menu item

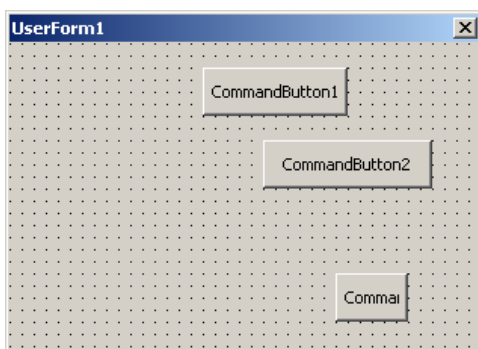


- **Remove:** Microsoft Visual Basic will move all controls (horizontally), except for the left control, to the left so that the left border of a control touches the right border of the next control



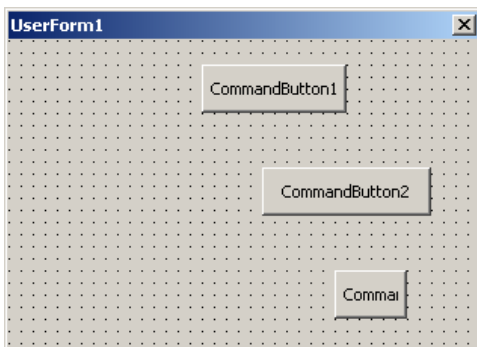
Vertical Spacing and Alignment

Suppose you have a group of horizontally aligned controls as follows:

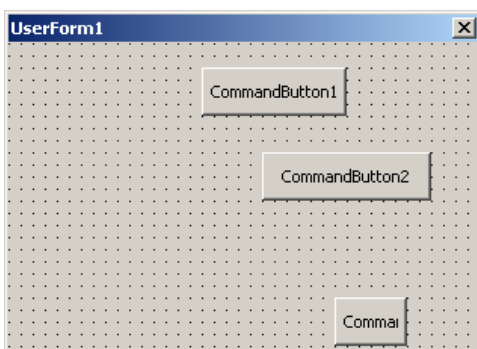


The controls on this form are not professionally positioned with regards to each other. Once again, the Format group of the main menu allow you to specify a better vertical alignment of controls relative to each other. To align them, on the main menu of Microsoft Visual Basic, click Format -> Vertical Spacing and click one of the following options:

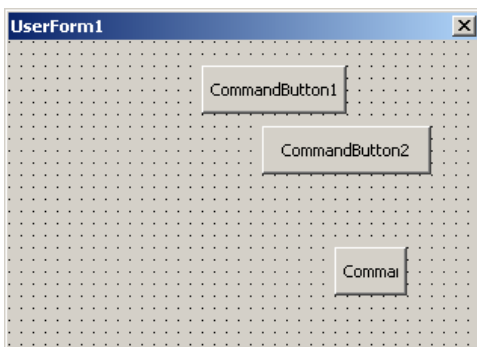
- **Make Equal:** Microsoft Visual Basic will calculate the total vertical distances that separate each combination of two controls and find their average. This average is applied to the vertical distance of each combination of two controls



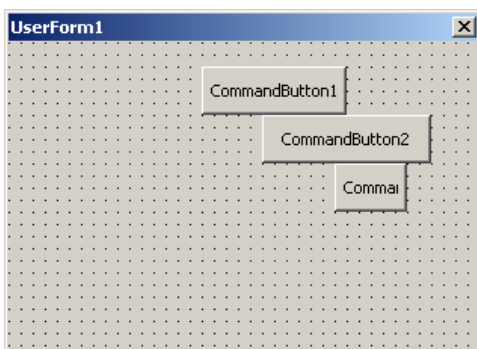
- **Increase:** Microsoft Visual Basic will move each control vertically, except the base control (the control that has darker handles) by one unit away from the base control. This will be done every time you click the Increase Horizontal Spacing button or the Format -> Horizontal Spacing -> Increase menu item



- **Decrease:** Microsoft Visual Basic will move each control, except the base control (the control that has darker handles) by one unit towards the base control. This will be done every time you click the Decrease Horizontal Spacing button or the Format -> Horizontal Spacing -> Decrease menu item




- **Remove:** Microsoft Visual Basic will move all controls vertically, except for the top control, to the top so that the top border of a control touches the bottom border of the next control towards the top



The Widths of Controls

Introduction

If you click a control's button on the Toolbox and click a UserForm, the control assumes a default width. The width of a control is the distance from its left to its right borders.


To visually specify the width of a control, click it, position the mouse on one of its left or right handles until the mouse cursor appears with a horizontal bar with two arrows . Then click and drag left or right in the direction of your choice. When you get the desired width, release the mouse.

The distance from the left border to the right border of a control is referred to as its **Width** property. Therefore, to specify the width of a control with precision, click the control. In the

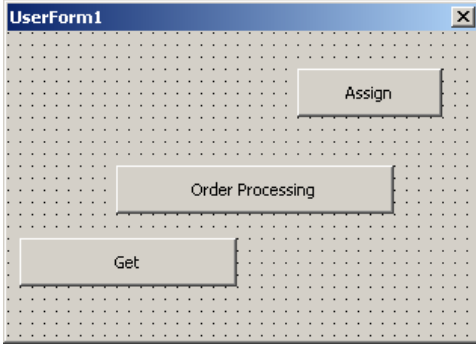
Properties window, click Width and type the desired value. To programmatically specify the width of a control, access it using its name, type the period, followed by **Width**, the assignment operator, and the desired value.

If a control displays or contains text, such as the caption of a button, click the control. On the main menu of Microsoft Visual Basic, click Format and click Size to Fit.

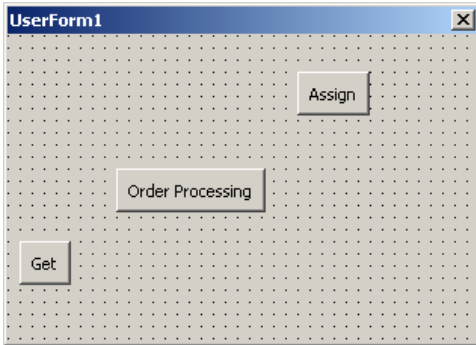
Enlarging or Narrowing Controls

Instead of one, you can also resize a group of controls at the same time. To enlarge or shrink many controls, select them. Position the mouse on the left or right handle of one of the selected controls to get the desired cursor . Click and drag left or right until you get the desired widths, then release the mouse.

Consider the following form:

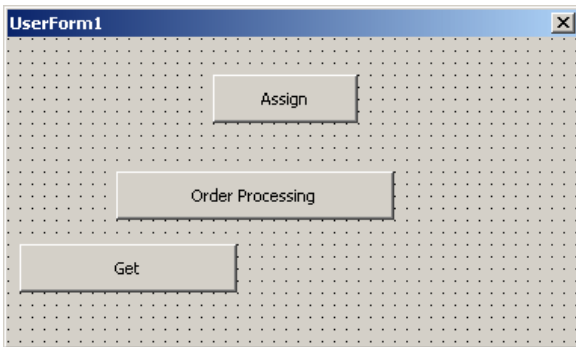


Imagine you would like each of these buttons to have just enough space to accommodate its caption. First select the controls that will be resized. To resize the controls, on the main menu of Microsoft Visual Basic, click Format and click Size to Fit. If you do, the controls will be resized based on the contents of their value:

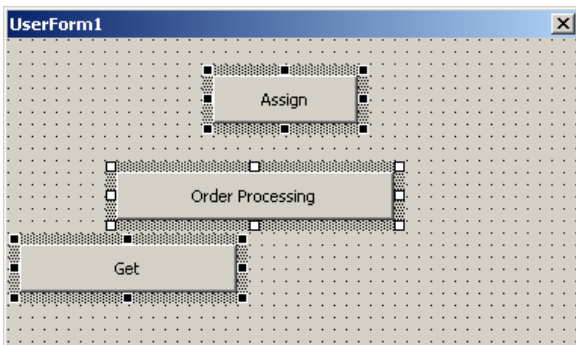


If all the controls are text boxes, their widths would be reduced to be able to hold a character.

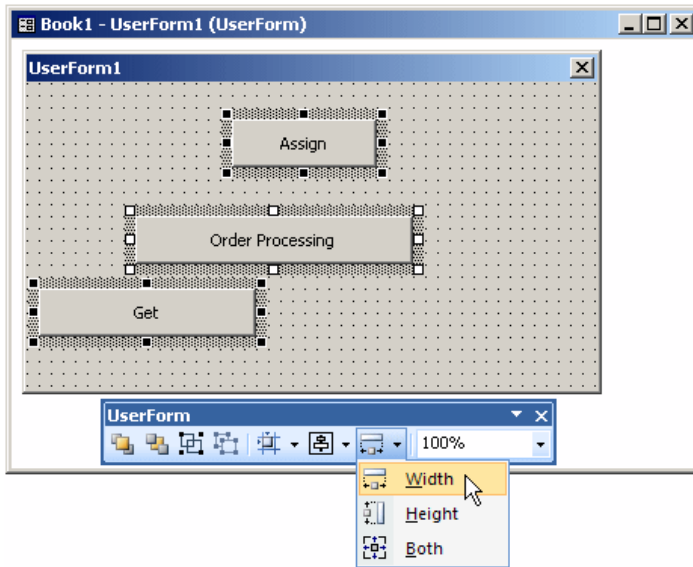
Consider the following form:



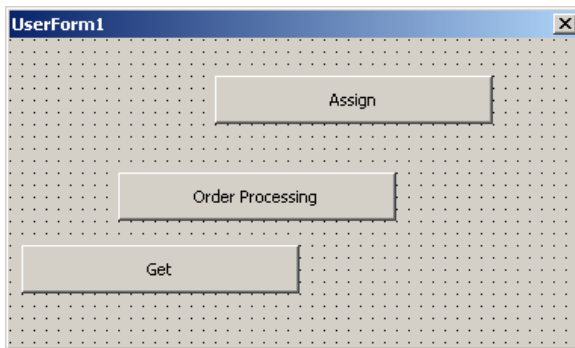
Imagine one of the controls has a certain width and you want to apply that width to the other controls. Select the controls but make as the base control the object that has the width you want. Here is an example where the button labeled Order Processing is selected as the base:



On the main menu, you can click Format -> Make Same Size -> Width. Alternatively, on the UserForm toolbar, you can click the arrow of the right button and click Width:



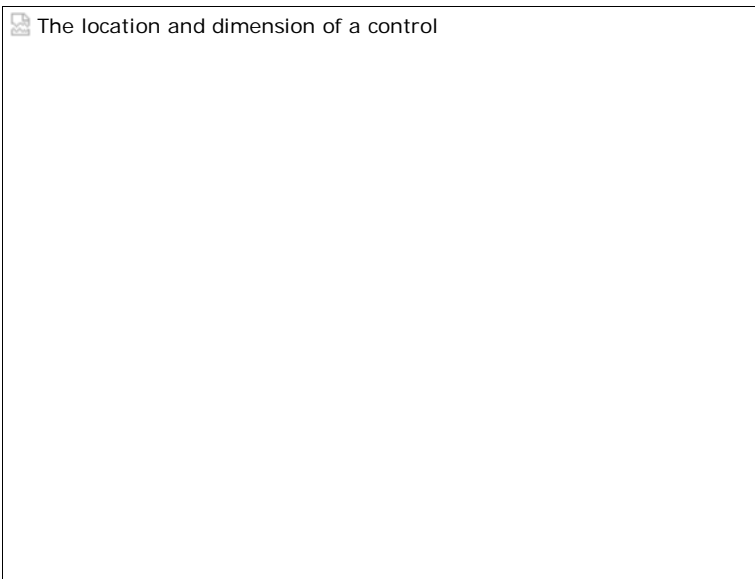
After doing this, the controls would be enlarged or narrowed based on the width of the control that was set as the base:




The Heights of Controls

Introduction

The height of a control is the distance from its top to its bottom borders. This can be illustrated as follows:




To visual specify the height of a control, click it, position the mouse on one of its top or bottom handle until the mouse cursor appears with a vertical bar with two arrows . Then click and drag up or down in the direction of your choice until you get the desired height. Then release the mouse.

To specify the width of a control with precision, click the control. In the Properties window, click Height and type the desired value. To programmatically specify the height of a control, access it using its name, type the period, followed by **Height**, followed by =, and the desired value.

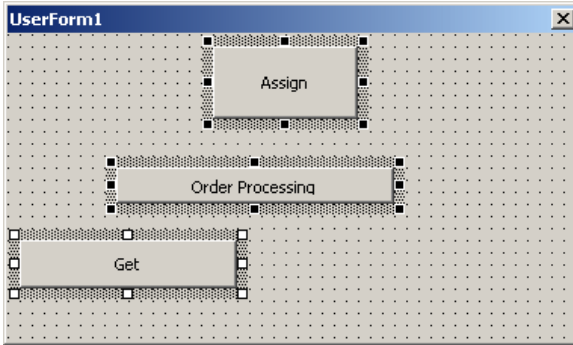
If a control displays or contains text, such as the caption of a button, click the control. On the main menu of Microsoft Visual Basic, click Format and click Size to Fit.

To programmatically specify the height of a control, type its name, access its **Height** property and assign the desired value.

Shrinking or Heightening Various Controls

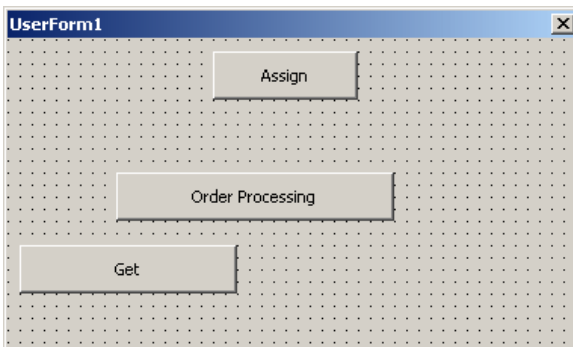
You can resize many controls at the same time. To do this, select the controls. Position the mouse on the top or bottom handle of one of the selected controls to get the desired cursor . Click and drag up or down. All of the selected controls would be resized.

You can shrink or heighten many controls based on the height of one of the controls. To start, select the controls but use as base the control that has the height you would like to apply on the other controls. Here is an example where the button labeled Get is set as the base:



On the main menu, you can click Format -> Make Same Size -> Height. Or, on the UserForm toolbar, you can click the arrow of the right button and click Height.

After doing this, the controls would get shrunk or tall based on the width of the control that was set as the base:




The Widths and Heights of Controls

Resizing a Control






Instead of separately setting the width or the height of a control or a group of controls, you can specify both pieces of information at the same time.

To visually specify both the width and the height of a control:

- Click and hold the mouse on a control. Drag in the direction of your choice
- Click the control to select it. Position the mouse on one of its borders but not on the handles until the mouse cursor appears as a cross with four arrows . Click and drag in the direction of your choice

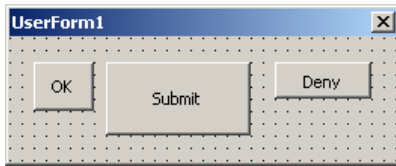
When you get to the desired position, release the mouse.

To resume, to resize a control, click it to select it. Position the mouse on a border, a handle, or a corner of the selected control. Use the appropriate mouse cursor:

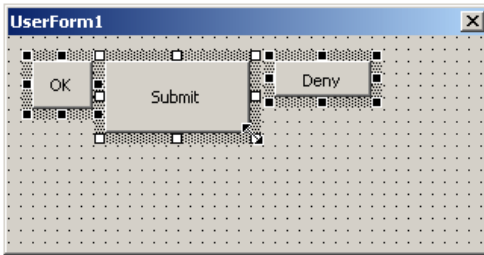
Cursor	Role
	Moves the seized border in the North-West <-> South-East direction
	Shrinks or heightens the control
	Moves the seized border in the North-East <-> South-West direction
	Narrows or enlarges the control
	Changes both the width and height of a control

Resizing the Controls

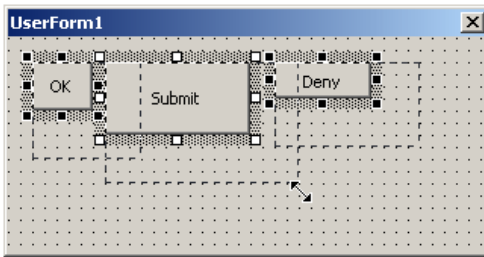
Imagine you have added three controls to a form and the design appears as follows:



To visually resize various controls, first select them. Position the mouse on the border or corner of one of the selected controls until you get the cursor that would resize to the direction of your choice:



Click and drag in the direction of your choice:



Once you get the desired size, release the mouse.

To precisely change the sizes of various controls at the same time, first select them. Then, in the Properties window, change the values of the Width and Height properties. The new value would be applied to all selected controls. Alternatively, Microsoft Visual Basic provides tools to automatically do this for you.

Control Maintenance

Copying a Control

If you had applied some design on a control and you want to replicate that design, you can copy the control. This is mostly a simple operation of copy n' paste. You can copy a control on the work area or on a form and paste it on the same container (you are not allowed to copy a control from the work area to a form and vice versa). You can also copy a control from one work area and paste it in another work area. You can copy a control from one form and paste it in another form.

When you copy and paste a control, there are some characteristics it would retain and some others it would lose. Normally, it would keep its aesthetic characteristics (such as the color) and its size but it will lose some others (such as its location and its programmatic characteristics such as its name).

To copy a control:

- Right-click the control and click Copy
- Click the control to select it and press Ctrl + C

To copy a group of controls, first select the controls:

- Right-click in the selection and click Copy
- Press Ctrl + C

To paste the copied controls, in the work area or on a form:

- Right-click the destination (work area or form) and click Paste
- Press Ctrl + V

Deleting Controls

If you have added a control to the work area or a form but you don't need it anymore, you can remove it from the container. You can also delete a group of controls in one step.

To remove a control from a work area or from a form:

- Click the control and press Delete
- Right-click the control and click Cut

To remove a group of controls, select them:

- Press Delete

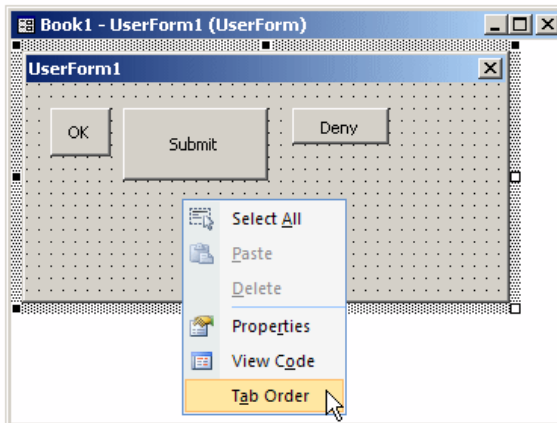
- Right-click one of the selected controls and click Cut

Tab Ordering

When using the controls of a form, you can press Tab to move from one control to another. For example, after entering a value in a text box of a form, if there is another text box on the right side, when you press Tab, the caret should move to that right control. If there is no control on the right side, the caret should move to the control under the one previously used. If the caret or focus is in the last bottom control on a form and you press Tab, the caret should move back to the first record. This follows the arranged sequence of the controls on the form. For this reason, the controls on a form should be aligned in the order of a logical sequence.

When you add a control to a form that already has other controls, it is sequentially positioned at the end of the existing controls. The sequence of controls navigation is set using the Tab Order dialog box. To access the Tab Order dialog box:

- Right-click the form and click Tab Order



- On the main menu of Microsoft Visual Basic, click View -> Tab Order

Primary Characteristics of Windows Controls

The Name of a Control

Every object used in a computer must have a name. This allows you and the operating system to know at any time what object you are referring to. When you add a new control to the work area in Microsoft Excel or to a form in Microsoft Visual Basic, the object receives a default name. For example, the first CommandButton you add is named CommandButton1. If you add another button, it would be named CommandButton2, and so on. The default name assigned may not be indicative enough for the role a control is playing, especially when you use many controls on the same container. Therefore, you should assign your own custom names to the controls you use.

In the Properties window, the name of a control is represented with the (Name) field. To change the name of a control, click (Name) and type the desired name. There are rules you must follow when naming your controls. The name of a control:

- Must start with a letter
- After the first letter, can contain letters, digits, and underscores only
- Cannot contain space

Based on these rules, you can adapt your own.

❖ Practical Learning: Naming Windows Controls

1. On the form, right-click CommandButton1 and click Properties
2. In the Properties window, click (Name) and type **cmdSubmit**
3. On the form, click the combo box
4. In the Properties window, click (Name) and type **cboSizes**

Border Style

Some controls display a border when they are drawn and some others don't. Some of these controls allow you to specify a type of border you want to show surrounding the control. This characteristic is controlled by the **BorderStyle** property.

The Text or Caption of a Control

The Caption or Text of a Control

Some controls are text-based, meaning they are meant to display or sometimes request text from the user. For such controls, this text is referred to as caption while it is simply called text for some other controls. This property is not available for all controls.

If a control displays text, it may have a property called **Caption** in the Properties window. After adding such a control to a work area or a form, its **Caption** field would display the same text as its name. At design time, to change the caption of the control, click its **Caption** field in the Properties window and type the desired value. For most controls, there are no strict rules to follow for this text. Therefore, it is your responsibility to type the right value. Some other controls have this property named **Text**. For such a control, when you add it to a work area or a form, its **Text** field in the Properties window may be empty. If you want, you can click the **Text** field and type the desired text.

The text provided in **Caption** or a **Text** field of a text-based control can only be set at design time. If you want the text to change while the application is running, you can format it. For example, such a control can display the current time or the name of the user who is logged in. These format attributes cannot be set at design time. To change the text of a text-based control at run time, either assign a simple string or provide a formatted string to the **Caption** or the **Text** property.

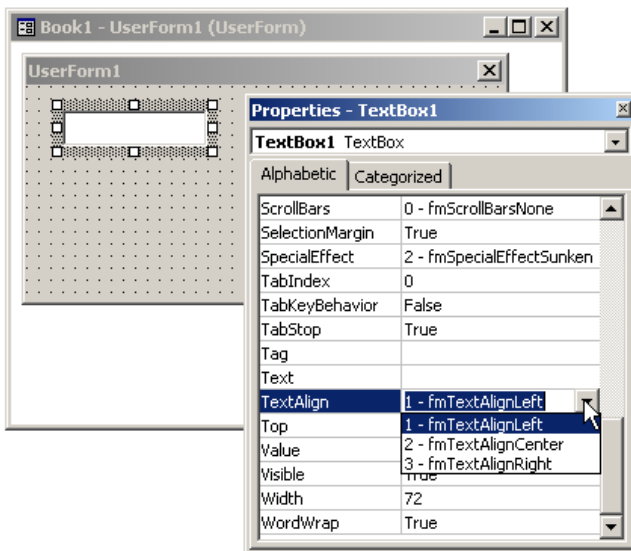
❖ Practical Learning: Setting Controls Text

1. On the form, click the button
2. In the Properties window, click **Caption** and type **Submit**
3. On the form, click the combo box
4. In the Properties window, click **Text** field and type **Large**

The Alignment of the Text or Caption of a Control

If a control is text-based, when you provide text to it or when you type text in it, by default, text is positioned to the left side of the control. This is appropriate if the value entered is a string (regular text). In some cases, such as numbers, you may prefer the value to be position in the center or on the right side of the control. This characteristic is referred to as the alignment of text. Once again, not all controls have this option.

The ability to control the alignment of text is done through the **TextAlign** property:




It provides three options:

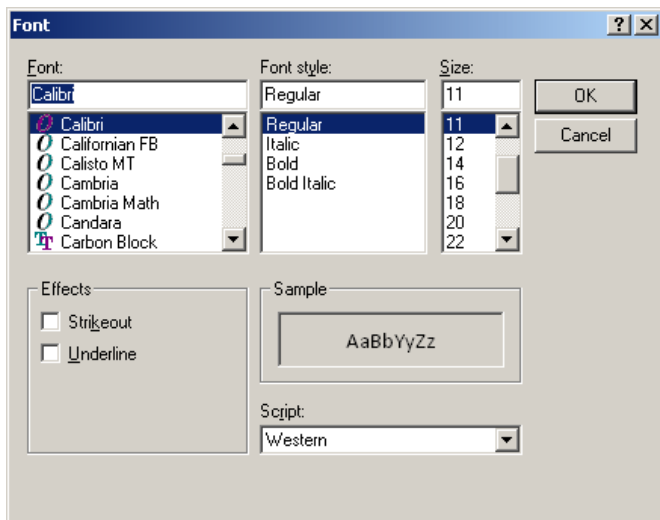
TextAlign	Result
1 - fmTextAlignLeft	Text will be aligned to the left of the control
2 - fmTextAlignCenter	Text will be position in the center of the control
3 - fmTextAlignRight	Text will be aligned to the left of the control

To programmatically specify the text alignment of a control that supports this characteristics, assign the desired option to this property. Here is an example:

```
TextBox1.TextAlign = fmTextAlignRight
```

The Font of Text of a Control

The font specify what face, size, and style a control should use to display its text. To specify or change the font of a control, click it to select in. In the Properties window, click **Font** and click the browse button . This would display the Font dialog box:

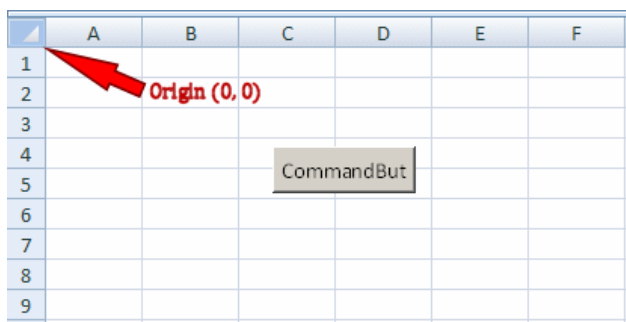


From this dialog box, you can select the font name, the style, the size, and the effect(s). Once you are ready, click OK.

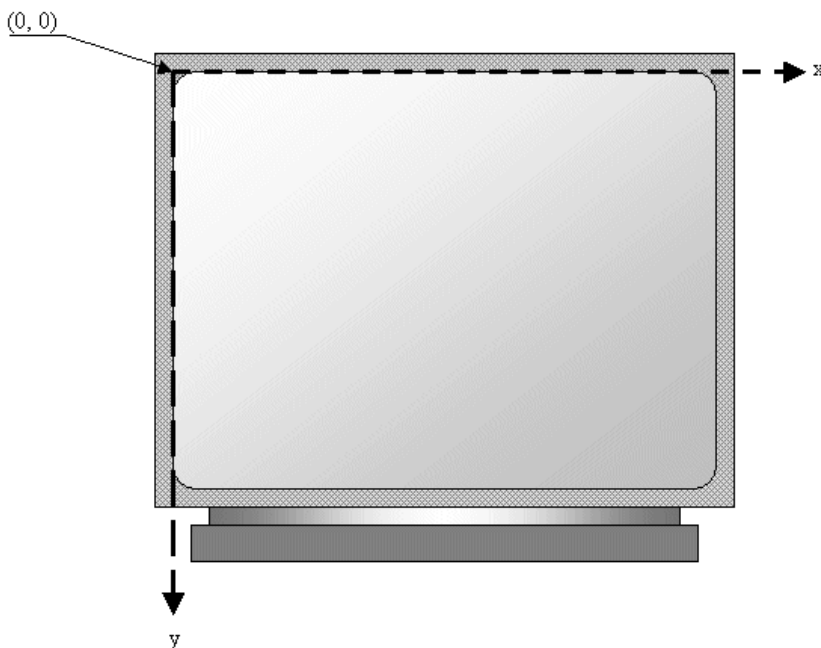
The Location of a Control

Introduction

We saw when you add a control to the work area or to a form, it gets a default position. After adding the control, it is positioned in the body of the parent using a Cartesian coordinate system whose origin is located on the top-left corner of the parent window. If the parent is the work area in Microsoft Excel, the origin is under the small boxes under the Formula Bar:



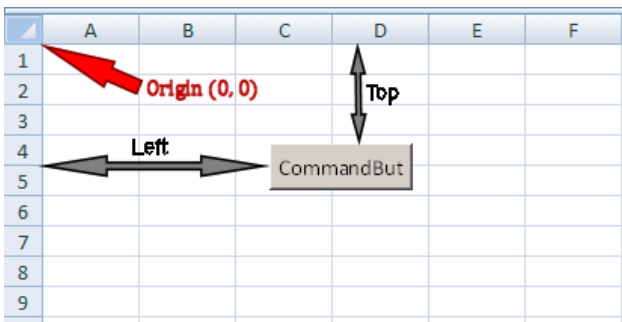
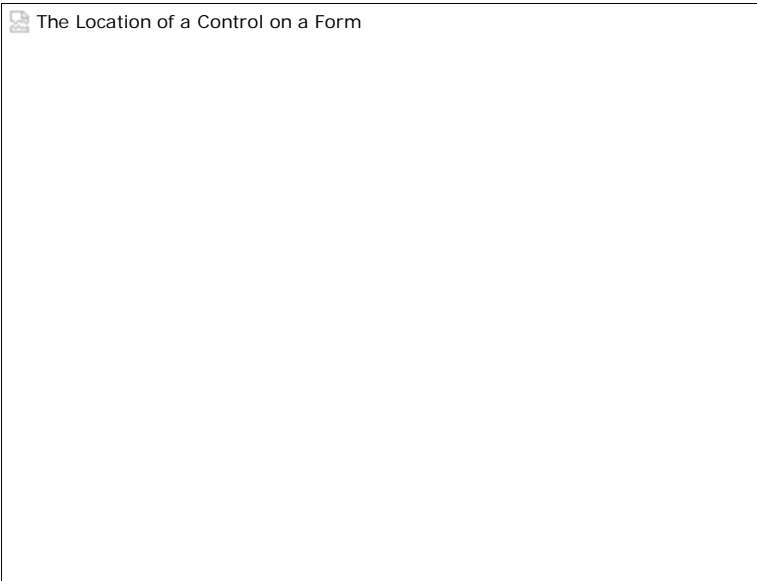
If you create a form in Microsoft Visual Basic, the origin of its location is located just under the title bar to the left:



The horizontal measurements move from the origin to the right. The vertical measurements move from the origin to the bottom. The location of a control is both:

- The distance between the top border of the work area or of the form and the top border of the control
- The distance from the left border of the work area or of the form to the left border of the control

In the Properties window, the distance between the top border of the work area or of the form and the top border of the control is represented by the **Top** property. The distance between the left border of the form and the left border of the control is represented by the **Left** property:

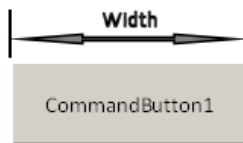


To move a control with precision, click it to select it and access its Properties window. In the Properties window, change either or both the **Left** and the **Top** values. To programmatically specify the location of a control, access it using its name. Then access its **Width** or its **Height** properties and assign the desired value.

The Size of a Control

The Width of a Control

We saw different ways of visually resizing a control. As seen already, the width of a control is the distance from its left to its right borders:



The width of a control is represented by the **Width** property. Therefore, to specify the width of a control with precision, access it using its name, type the period, followed by **Width**, the assignment operator, and the desired value.

The Height of a Control

As described already, the height of a control is the distance from its top to its bottom borders:



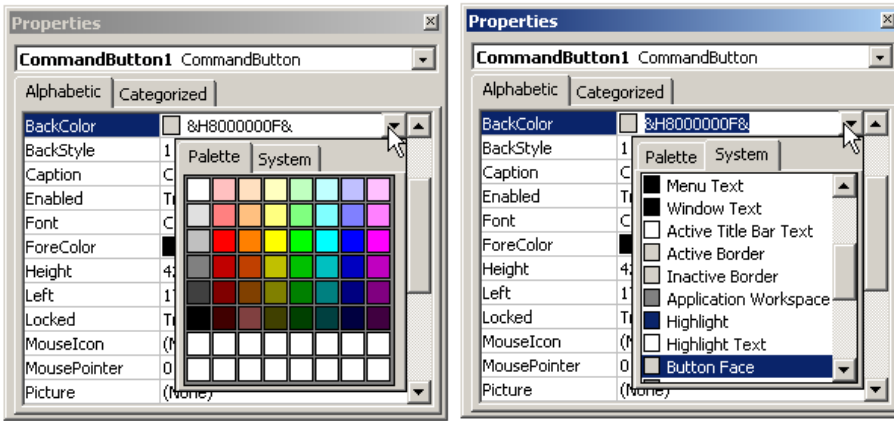
To programmatically specify the height of a control, access it using its name, type the period, followed by **Height**, followed by =, and the desired value.

The Colors of a Control

Introduction

Colors are used to paint something about a control. For example, you can change the color of a control or just the color of the text that a control is displaying. Both Microsoft Excel and Microsoft

To visually change a color, you can use the Properties window. In the Properties window, the fields that support the color options are provided as a combo box. When you click the arrow of the combo box, a window made of two parts would display:



The color window is divided into two property pages labeled Palette and System. The Palette property page is probably the easiest section to specify a color because it provides small boxes that each show its color. The colors are represented each by a name. Those are official names recognized by the Microsoft Windows operating systems but you should not use those colors in your code.

To programmatically support colors, Microsoft Visual Basic provided two systems. Microsoft Visual Basic provides a few constants values you can use as colors. These constants are:

Constants	Resulting Color
vbBlack	Black
vbBlue	Blue
vbCyan	Cyan
vbGreen	Green
vbMagenta	Magenta
vbRed	Red
vbWhite	White
vbYellow	Yellow

As you can see, this is a limited list. Obviously there should be other ways to specify a color. In Microsoft Windows operating systems, a color is recognized as a number made of three parts. The first part is a small number that ranges from 0 to 255. This part represents the red section. The second part also is a number from 0 to 255 and represents the green value. The third part also is a number from 0 to 255 and represents the blue part. To support this, the Visual Basic language provides a function named RGB and whose syntax is:

```
Function RGB(Red As Byte, Green As Byte, Blue As Byte) As Long
```

This function takes three arguments. Each argument should be a number between 0 and 255. If the arguments are valid, the function would produce a Long value that represents a color recognized by Microsoft Windows. Here is an example:

```
BackColor = RGB(28,174, 77)
```

As mentioned already, the RGB() function produces a Long integer that represents a color. If you already know the number that represents the color, you can use it as the color. For example, you can assign it to the colored property. Here is an example:

```
BackColor = 4912394
```

This number is provided in decimal format. As an alternative, you can provide it in hexadecimal format. Here is an example:

```
BackColor = &HF420DD
```

The Background of a Control

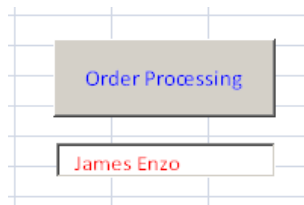
When you add a new control to a work area or a form, the control is painted with a certain color but this depends on the control. The background color of a control is the color used to paint the surface of the control.

To change the background color of a control, first select it. In the Properties window, click **BackColor** and select the desired color.

The Text Color of a Control

To make its text visible, a control shows it in a certain color that, by default, is black. If you want, you can change that color.

To support the color used to display its text, each control is equipped with a property named ForeColor. Therefore, to visually change the color of text of a control, select that control. In the Properties window, click ForeColor and select the desired color. Here are examples:



To programmatically specify or change the text color of a control, access it. Then access its `ForeColor` property and assign it the desired color.

The Border Color of a Control

Almost every control has a border. This shows where the control starts and where it ends. The controls that show a border paint it with a certain color. Most controls that have a border use a type of 3-D effect. This depends on the control. To control the color of the border of a control, click it to select it. In the Properties window, click `BorderColor` and select the desired color. To programmatically specify or change the border color of a control, assign the desired color to its `BorderColor` property.

Operating System Characteristics of Controls

The Tab Stop of a Control

You can navigate through controls using the Tab key. When that key is pressed, the focus moves from one control to the next. By their designs, not all controls can receive focus and not all controls can participate in tab navigation. Even controls that can receive focus must be primarily included in the tab sequence.

The participation to tab sequence is controlled by the Boolean `TabStop` property in the Properties window. Every visual control that can receive focus is already configured to have this property set to `True`. If you want to remove a control from this sequence, set its `TabStop` value to `False`.

The Tab Index of a Control

If a control has the `TabStop` property set to `True`, to arrange the navigation order of controls, you can click a control on the form. Then, in the Properties window, set or change the value of its `TabIndex` field. The value must be a positive natural number.

Control's Visibility

A control is referred to as visible if it can be visually located on the screen. You can use a control only if you can see it. You have the role of deciding whether a control must be seen or not and when. The visibility of an object is controlled by the its `Visible` property.

At design time, when you add a control to the work area or to a form, it is visible by default. This is because its `Visible` property is set to `True` in the Properties window. If you don't want a control to primarily appear when the form comes up, you can set its `Visible` property to `False`.

Control's Availability

To be able to use a control, it must allow operations on it. For example, if a control is supposed to receive text, you can enter characters in it only if this is made possible. To make a control available, the object must be enabled. The availability of an object is controlled by the `Enabled` property.

By default, after adding a control to a form, it is enabled and its `Enabled` property in the Properties window is set to `True`. An enabled control displays its text or other characteristics in their normal settings. If you want to disable a control, set its `Enabled` property to `False`.

❖ Practical Learning: Designing a Form

1. Click each control on the form and press Delete
2. Design the form as follows:

Georgetown Dry Cleaning Services

Order Identification

Employee #:

Customer Phone:

Date Left: Time Left:

Date Expected: Time Expected:

Date Picked Up: Time Picked Up:

Items to Clean

Item	Unit Price	Qty	Sub-Total
Shirts	1.50	0	0.00
Pants	2.25	0	0.00
None	0.00	0	0.00
None	0.00	0	0.00
None	0.00	0	0.00
None	0.00	0	0.00

Order Summary

Cleaning Total:

Tax Rate: %







Tax Amount:

Order Total:

Order Status:

Close

Control		Caption/Text	Name	Other Properties
Frame		Order Identification		
Label		Employee #:		
TextBox			txtEmployeeNumber	
TextBox			txtEmployeeName	
Label		Customer Phone:		
TextBox			txtCustomerPhone	
TextBox			txtCustomerName	
Label		Date Left:		
TextBox			txtDateLeft	
Label		Time Left:		
TextBox			txtTimeLeft	
Label		Date Expected:		
TextBox			txtDateExpected	
Label		Time Expected:		
TextBox			txtTimeExpected	
Label		Date Picked Up:		
TextBox			txtDatePickedUp	
Label		Time Picked Up:		
TextBox			txtTimePickedUp	
Frame		Items to Clean		
Label		Item		
Label		Unit Price		
Label		Qty		
Label		Sub-Total		
Label		Shirts		
TextBox		1.50	txtUnitPriceShirts	TextAlign: 3 - fmTextAlignRight
TextBox		0	txtQuantityShirts	TextAlign: 3 - fmTextAlignRight

TextBox		0.00	txtSubTotalShirts	TextAlign: 3 - fmTextAlignRight
Label		Pants		
TextBox		2.25	txtUnitPricePants	TextAlign: 3 - fmTextAlignRight
TextBox		0	txtQuantityPants	TextAlign: 3 - fmTextAlignRight
TextBox		0.00	txtSubTotalPants	TextAlign: 3 - fmTextAlignRight
ComboBox		None	cbxNameItem1	
TextBox		0.00	txtUnitPriceItem1	TextAlign: 3 - fmTextAlignRight
TextBox		0	txtQuantityItem1	TextAlign: 3 - fmTextAlignRight
TextBox		0.00	txtSubTotalItem1	TextAlign: 3 - fmTextAlignRight
ComboBox		None	cbxNameItem2	
TextBox		0.00	txtUnitPriceItem2	TextAlign: 3 - fmTextAlignRight
TextBox		0	txtQuantityItem2	TextAlign: 3 - fmTextAlignRight
TextBox		0.00	txtSubTotalItem2	TextAlign: 3 - fmTextAlignRight
ComboBox		None	cbxNameItem3	
TextBox		0.00	txtUnitPriceItem3	TextAlign: 3 - fmTextAlignRight
TextBox		0	txtQuantityItem3	TextAlign: 3 - fmTextAlignRight
TextBox		0.00	txtSubTotalItem3	TextAlign: 3 - fmTextAlignRight
ComboBox		None	cbxNameItem4	
TextBox		0.00	txtUnitPriceItem4	TextAlign: 3 - fmTextAlignRight
TextBox		0	txtQuantityItem4	TextAlign: 3 - fmTextAlignRight
TextBox		0.00	txtSubTotalItem4	TextAlign: 3 - fmTextAlignRight
Frame		Order Summary		
Label		Cleaning Total:		
TextBox		0.00	txtCleaningTotal	TextAlign: 3 - fmTextAlignRight
Label		Tax Rate:		
TextBox		5.75	txtTaxRate	TextAlign: 3 - fmTextAlignRight
Label		%		
Label		Tax Amount:		
TextBox		0.00	txtTaxAmount	TextAlign: 3 - fmTextAlignRight
Label		Order Total:		
TextBox		0.00	txtOrderTotal	TextAlign: 3 - fmTextAlignRight
Label		Order Status:		
ComboBox			cbxOrderStatus	
Button		Close	btnClose	

3. Return to Microsoft Excel
4. To save the file, press Ctrl + S
5. In the Save As Type combo box, select Excel Macro-Enabled
6. Change the File Name to **gdcs1**
7. Click Save

Controls' Methods: Giving Focus

On a form that has many controls, at one particular time, only one control can receive input from the user. The control that is currently receiving input or actions from the user is said to have

To give focus to a control, the user can click the intended control or press Tab a few times until the control receives focus. To programmatically give focus to a control, type the name of the control, followed by the period operator, followed by the **SetFocus** method. An example would be:

```
Private Sub Example()  
    txtAddress.SetFocus  
End Sub
```




Messages and Events of Windows Controls

Controls Messages

Introduction

You can add Windows controls to a work area or to a form to help a user interact your application. When a control is used, it must communicate with the operating system. For example, when a user clicks, the object that was clicked must inform the operating system that it has been clicked. This is the case for every control used in an application. Because a typical application can involve many controls, a mechanism was designed to manage the occurrence of actions.

To communicate its intention to the operating system, a Windows control must compose a message and send it (to the operating system).

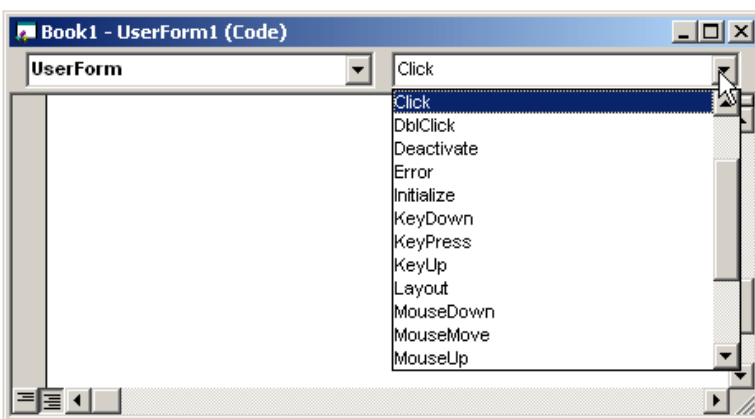
The Parts of a Message

In order to make a message clear, the control that wants to send it must provide three important pieces of information:

- WHO sent the message? When a control sends a message, it must identify itself because there can be many controls sending different messages at the same time. The operating system will need to know where the message came from. This is one of the reasons why every control must have a name. Also because each message is particular to the control that sent it, the message is considered a private matter. Based on this, the code of a message starts with **Private Sub**, followed by the name of the control that is sending the message:

Private Sub *ControlName*

- WHAT message? When a control is sending a message, it must specify the type of message. A control can be able to send various types of messages. For example, when a control gets clicked, it sends a Click message. If the same control receives focus but you press a key, the control sends a keyboard-related message. When the mouse passes over that same control, it sends a different type of message. Every message a control can send has a name. To see the types of message available for a particular control, open Microsoft Visual Basic. In the Object combo box, select the name of the control. Then, click the arrow of the Procedure combo box:



By convention, the name of the message follows the name of the control but they are separated with an underscore. It would appear as:

Private Sub *ControlName_Push*

- Arguments: An argument is additional information needed to process a message. When a control sends a message, it may need to accompany it with some information. For example, if you position the mouse on a control and click, the operating system may need to know what button of the mouse was used to click. On the other hand, if you select an object and start dragging, the operating system may need to know if a key such as Shift or Ctrl was held down while you were dragging.

An additional piece of information that the control provides is provided as an argument. While some messages may need to provide only one piece of information, some messages would require more than one argument. Some other messages don't need any additional information at all: the name of the message would completely indicate how the message must be processed.

The arguments of a message are provided in parentheses. They would appear as:

Private Sub *ControlName_Push(Argument1, Argument2, Argument_n)*

After specifying the message, you can type code that tells the operating system what to do to process the message. To indicate the end of the code that relates to a message, type **End Sub**


```
Private Sub ControlName_Push(Argument1, Argument2, Argument_n)
```

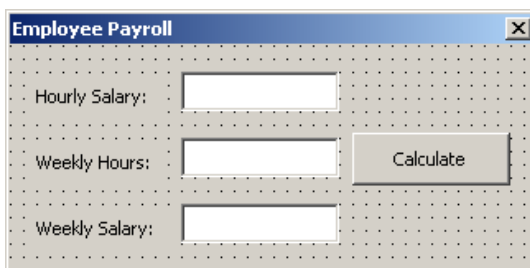
```
End Sub
```

As mentioned earlier, a message must be composed and sent. The action of sending a message is called an event. It is also said that the controls "fires" an event. Based on the above descriptions, to compose and send a message, in the Object combo box, you can select the name of the control that will send the message, then select the desired message in the Procedure combo box. When you do this, Microsoft Visual Basic will write the first line that specifies the name of the control, the name of the event, its arguments if any, and would write End Sub for you. You can then enter the necessary code between those two lines.

Most Windows control have a special event referred to as the default. This is the even that is the most obvious that the control can fire. For example, when you think of a button, the first action that comes to mind is Click. For this reason, Click is the default event of a button. If you add a control to a work area or to a form and double-click the control, its default event would be invoked and the skeleton of that event would be written in the corresponding module. If you don't want to use that event or to fires another event for the same control, you can simply select the event in the Procedure combo box.

❖ Practical Learning: Introducing Messages and Events

1. Start Microsoft Excel
2. Save the document as **Messages**
3. To open Microsoft Visual Basic, on the [Ribbon](#), click Developer and, in the Code section, click Visual Basic
4. To create a form, on the Standard toolbar, click the Insert UserForm button 
5. Right-click the form and click Properties
6. In the Properties window, click (Name) and type **frmPayroll**
7. On the Toolbox, click the TextBox and click the form
8. Complete the design of the form as follows:



Control	Name	Caption	Other Properties
Label	lblHourlySalary	Hourly Salary:	
TextBox	txtHourlySalary		TextAlign: 3 - frmTextAlignRight
Label	lblWeeklyHours	Weekly Hours:	
TextBox	txtWeeklyHours		TextAlign: 3 - frmTextAlignRight
CommandButton	cmdCalculate		
Label	lblWeeklySalary	Weekly Salary:	
TextBox	txtWeeklySalary		TextAlign: 3 - frmTextAlignRight

9. Save the file

Common Events of Windows Controls

Click

To interact with the computer, one of the most usually performed actions is to click. The mouse is equipped with two buttons. The most clicked button is the left one.

Because the action simply consists of clicking, when you press this button, a simple event, called **Click** is sent or fired. When you press the (left) button on the mouse, the mouse pointer is usually

on a Windows control. Based on this, the control that is clicked "owns" the event and must manage it. Therefore, no detailed information is provided as part of the event. The operating system believes that the control that fired the event knows what to do. For this reason, whenever you decide to code an **OnClick** event, you should make sure you know what control sent or fired the event. This is (one of) the most common events of Windows controls.

❖ Practical Learning: Generating a Click Event

1. To generate a Click event for the button, on the form, double-click the Calculate button and notice that its Click event has been generated
2. Implement the Click event as follows:

```
Private Sub cmdCalculate_Click()
    Dim HourlySalary As Currency
    Dim WeeklyHours As Double
    Dim WeeklySalary As Currency

    HourlySalary = CCur(txtHourlySalary.Text)
    WeeklyHours = CDbl(txtWeeklyHours.Text)
    WeeklySalary = HourlySalary * WeeklyHours

    txtWeeklySalary.Text = CStr(WeeklySalary)
End Sub
```

3. To test the form, on the main menu of Visual Basic, click Run -> Run Sub/UserForm
4. Enter **15.48** in the Hourly Salary and **36.50** in the Weekly Hours text boxes and click Calculate

5. Close the form

Double-Click

Another common action you perform on a control may consist of double-clicking it. This action causes the control to fire an event known as **DbIcIck**.

❖ Practical Learning: Generating a Double-Click Event

1. On the form, right-click the Calculate button and click View Code
2. In the Object combo box, select UserForm
3. In the Procedure combo box, select DbIcIck and notice the structure of the event:

```
Private Sub UserForm_DbIcIck(ByVal Cancel As MSForms.ReturnBoolean)
    lblHourlySalary.BackColor = vbBlue
    lblWeeklyHours.BackColor = vbBlue
    lblWeeklySalary.BackColor = vbBlue

    lblHourlySalary.ForeColor = vbWhite
    lblWeeklyHours.ForeColor = vbWhite
    lblWeeklySalary.ForeColor = vbWhite

    BackColor = vbBlue
End Sub
```

4. To test the form, on the main menu of Visual Basic, click Run -> Run Sub/UserForm
5. Double-click the form

6. Close the form

Just as an application can have many forms, a form can be equipped with various controls. Such is the case for any data entry form. On a form that is equipped with many controls, only one control can be changed at a time. Such a control is said to have focus. To give focus to a control, you can click it or can keep pressing Tab until the desired control indicates that it has focus. In a form with many controls, the control that has focus may display a caret or a dotted line around its selection or its caption.

When a form or a control receives focus, it fires the **Enter** event. We mentioned that a user can give focus to a control by clicking it. If the control is text-based, then a caret blinking in the control indicates that the control has focus.

The **Enter** event does not take any argument:

```
Private Sub TextBox1_Enter()  
End Sub
```

Exiting a Control

After using a control, you can switch to another control either by clicking another or by pressing Tab. This causes the focus to shift from the current control to another. If the focus shifts to another control, the control that had focus fires an **Exit** event.

The **Exit** event takes one argument, :

```
Private Sub TextBox1_Exit(ByVal Cancel As MSForms.ReturnBoolean)  
End Sub
```

Keyboard Events

Word processing consists of manipulating text and characters on your computer until you get the fantastic result you long for. To display these characters, you press some keys on your keyboard. If the application is configured to receive text, your pressing actions will display characters on the screen. The keyboard is also used to perform various other actions such as accepting what a dialog box displays or dismissing it.

When you press the keys on a keyboard, the control in which the characters are being typed sends one or more messages to the operating system. There are three main events that Microsoft Windows associates to the keyboard.

KeyDown: When you press a key on the keyboard, an event called **KeyDown** is fired. The KeyDown event takes two arguments:

```
Private Sub TextBox1_KeyDown(ByVal KeyCode As MSForms.ReturnInteger, _  
                             ByVal Shift As Integer)  
End Sub
```

KeyUp: When you release a key that was pressed, an event called **KeyUp** fires. These two previous events apply to almost any key on the keyboard, even if you are not typing; that is, even if the result of pressing a key did not display a character on the document.

```
Private Sub TextBox1_KeyUp(ByVal KeyCode As MSForms.ReturnInteger, _  
                           ByVal Shift As Integer)  
End Sub
```

KeyPress: The **KeyPress** event fires if the key you pressed is recognized as a character key; that is, a key that would result in displaying a character in a document.

```
Private Sub TextBox1_KeyPress(ByVal KeyAscii As MSForms.ReturnInteger)  
End Sub
```

Some keys on the keyboard don't display anything on a document. Instead, they perform (only) an action. Examples of such keys are Enter, Tab, Esc. Therefore, if you mean to find out what key you pressed, use the **KeyDown** event instead of the **KeyPress** event.

Pressing a Mouse Button Down

A mouse is equipped with buttons, usually two, that you press to request an action. Compared to the keyboard, the mouse claims many more events that are directly or indirectly related to pressing one of its buttons.

When you press one of the buttons on the mouse, an event, called **MouseDown** fires. This event carries enough information as three arguments.

```
Private Sub txtFirstName_MouseDown(Button As Integer, Shift As Integer,
    X As Single, Y As Single)
```

```
End Sub
```

- The operating system needs to know what button was pressed; this is represented as the left or the right button. The left button is known as **vbLeftButton**. The right button is referenced as **vbRightButton**. If the mouse is equipped with a middle button, it would be **vbMiddleButton**. In reality, these buttons have (constant) numeric values of 0, 1, and 2 respectively.
- Secondly, the operating system needs to know whether a special key, Shift, Ctrl, or Alt, was pressed. These buttons are called **vbShiftMask**, **vbCtrlMask**, and **vbAltMask** respectively. In reality, they are represented with 1, 2, and 4 respectively.
- Lastly, the operating system needs to know the screen coordinates of the mouse pointer, that is, the coordinates of the point where the mouse landed. X represents the distance from the top left corner of the parent to the mouse. Y represents the vertical measure of the point from the top-left corner down.

Releasing a Mouse Button

When you release a button that was pressed on the mouse, a new event fires. This event is called **MouseUp**. It provides the same types of information as the **MouseDown** event:

```
Private Sub txtFirstName_MouseUp(Button As Integer, Shift As Integer,
    X As Single, Y As Single)
```

```
End Sub
```

Moving the Mouse

The **MouseMove** event is sent while you are moving the mouse on a control. It provides the same pieces of information as the **MouseDown** and the **MouseUp** events:

```
Private Sub txtFirstName_MouseMove(Button As Integer, Shift As Integer,
    X As Single, Y As Single)
```

```
End Sub
```

Changing Text

One of the most important messages of a text box occurs when its content changes. That is, when the text it has is deleted, added to, or edited. When you click in a text box control and start typing it or change its text, the control fires a Change event.

❖ Practical Learning: Using the Change Event of a Text Box

1. To add a new form, on the main menu of Microsoft Visual Basic, click Insert -> UserForm
2. In the Properties window, change its (Name) to **frmEmployeeInformation**
3. Design the form as follows:

Control	Name	Caption
Label		First Name:
TextBox	txtFirstName	

Label		Last Name:
TextBox	txtLastName	
Label		Full Name:
TextBox	txtFullName	

4. Double-click the top text box and implement its Change event as follows:

```
Private Sub txtFirstName_Change()
    Dim FirstName As String
    Dim LastName As String
    Dim FullName As String

    FirstName = txtFirstName.Text
    LastName = txtLastName.Text
    FullName = FirstName & " " & LastName

    txtFullName.Text = FullName
End Sub
```

5. In the Object combo box, select txtLastName and implement its **Change** event as follows:

```
Private Sub txtLastName_Change()
    Dim FirstName As String
    Dim LastName As String
    Dim FullName As String

    FirstName = txtFirstName.Text
    LastName = txtLastName.Text
    FullName = FirstName & " " & LastName

    txtFullName.Text = FullName
End Sub
```

6. To test the form, on the main menu of Visual Basic, click Run -> Run Sub/UserForm
7. Click the top text box, type **Julienne** and press Tab
8. In the other text box, start typing Pal and notice that the Full Name text box is changing

The screenshot shows a window titled "Employee Information" with a close button (X) in the top right corner. Inside the window, there are three text boxes arranged vertically. The first text box is labeled "First Name:" and contains the text "Julienne". The second text box is labeled "Last Name:" and contains the text "Pal". The third text box is labeled "Full Name:" and contains the text "Julienne Pal".

9. Complete it with Palace and close the form



Objects and Collections

Fundamentals of Collections

Introduction

A collection is a series of items where each item has the same characteristics. In other words, all items can be described the same way. Programmatically, a collection is a series of items where all items share the same properties and methods, if any. For example, a collection can be made of employees of a company where each employee can be described with a characteristics such as a name.

❖ Practical Learning: Introducing Objects

1. Start Microsoft Excel and open the **gdcs1** document you created in [Lesson 10](#)
2. To open Microsoft Visual Basic, on the Ribbon, click Developer and, in the Code section, click Visual Basic:


Item	Unit Price	Qty	Sub-Total
Shirts	1.50	0	0.00
Pants	2.25	0	0.00
None	0.00	0	0.00
None	0.00	0	0.00
None	0.00	0	0.00
None	0.00	0	0.00

3. Right-click the form and click View Code

Creating a Collection

In our lessons, we will not create new collections. We will only use two categories: the **Collection** class and the built-in collection.

To support collections, the Visual Basic language is equipped with a class named **Collection**.

 Actually, the **Collection** class we are going to study here is the one defined in VBA. The parent Visual Basic language has a somewhat different **Collection** class with additional functionality not found in the VBA's version.

This class can be used to create a collection. To do this, declare a variable of type **Collection**. Here is an example:

```
Sub Exercise()
    Dim Employees As Collection
End Sub
```

After declaring the variable, to allocate memory for it, use the **Set** operator to assign a **New** instance to the variable. Here is an example:

```
Sub Exercise()
    Dim Employees As Collection

    Set Employees = New Collection
End Sub
```

Instead of always creating a new collection unless you have to, VBA for Microsoft Excel comes equipped with many collections so that you almost may never need to create your own collection. The collections that are already built in the VBA language are referred to as built-in collections.

The built-in collection classes are derived from the Visual Basic's **Collection** class. As a result, all of their primary functionality comes from the **Collection** class. This also means that everything we will mention for the **Collection** class applies to any built-in collection.

To use a built-in collection, you can declare a variable for it. Here is an example:

```
Sub Exercise()

    Dim CurrentSheets As Worksheets

End Sub
```

In reality, and as we will next in the next lessons, when Microsoft Excel starts, most (if not all) of the built-in collection classes are already available so that you do not have to declare their variable before using them.

Characteristics of, and Operations on, a Collection

Adding an Item to a Collection

The primary operation to perform on a collection consists of adding items to it. To support this, the **Collection** class is equipped with a method name **Add**. Its syntax is:

```
Public Sub Add( _
    ByVal Item As Object, _
    Optional ByVal Key As String, _
    Optional ByVal { Before | After } As Object = Nothing _
)
```

This method takes three arguments. Only the first is required. The *Item* argument specifies the object to be added to the collection. Here is an example:

```
Sub Exercise()
    Dim Employees As Collection

    Set Employees = New Collection

    Employees.Add "Patricia Katts"
End Sub
```

In the same way, you can add as many items as you want:

```
Sub Exercise()
    Dim Employees As Collection

    Set Employees = New Collection

    Employees.Add "Patricia Katts"
    Employees.Add "James Wiley"
    Employees.Add "Gertrude Monay"
    Employees.Add "Helene Mukoko"
End Sub
```

Remember that if you are using one of the built-in collection classes, you do not have to declare a variable for it. You can just call the **Add** method on it to add an item to it. Here is an example:

```
Sub Exercise()
    Worksheets.Add
End Sub
```

❖ Practical Learning: Introducing Objects

1. In the Object combo box, make sure UserForm is selected. In the Procedure combo box, select Activate
2. Type the code as follows:


```

Private Sub UserForm_Activate()
    Rem Create the "other" cleaning items
    cbxNameItem1.AddItem "None"
    cbxNameItem1.AddItem "Women Suit"
    cbxNameItem1.AddItem "Dress"
    cbxNameItem1.AddItem "Regular Skirt"
    cbxNameItem1.AddItem "Skirt With Hook"
    cbxNameItem1.AddItem "Men 's Suit 2Pc"
    cbxNameItem1.AddItem "Men 's Suit 3Pc"
    cbxNameItem1.AddItem "Sweaters"
    cbxNameItem1.AddItem "Silk Shirt"
    cbxNameItem1.AddItem "Tie"
    cbxNameItem1.AddItem "Coat"
    cbxNameItem1.AddItem "Jacket"
    cbxNameItem1.AddItem "Swede"

    cbxNameItem2.AddItem "None"
    cbxNameItem2.AddItem "Women Suit"
    cbxNameItem2.AddItem "Dress"
    cbxNameItem2.AddItem "Regular Skirt"
    cbxNameItem2.AddItem "Skirt With Hook"
    cbxNameItem2.AddItem "Men 's Suit 2Pc"
    cbxNameItem2.AddItem "Men 's Suit 3Pc"
    cbxNameItem2.AddItem "Sweaters"
    cbxNameItem2.AddItem "Silk Shirt"
    cbxNameItem2.AddItem "Tie"
    cbxNameItem2.AddItem "Coat"
    cbxNameItem2.AddItem "Jacket"
    cbxNameItem2.AddItem "Swede"

    cbxNameItem3.AddItem "None"
    cbxNameItem3.AddItem "Women Suit"
    cbxNameItem3.AddItem "Dress"
    cbxNameItem3.AddItem "Regular Skirt"
    cbxNameItem3.AddItem "Skirt With Hook"
    cbxNameItem3.AddItem "Men 's Suit 2Pc"
    cbxNameItem3.AddItem "Men 's Suit 3Pc"
    cbxNameItem3.AddItem "Sweaters"
    cbxNameItem3.AddItem "Silk Shirt"
    cbxNameItem3.AddItem "Tie"
    cbxNameItem3.AddItem "Coat"
    cbxNameItem3.AddItem "Jacket"
    cbxNameItem3.AddItem "Swede"

    cbxNameItem4.AddItem "None"
    cbxNameItem4.AddItem "Women Suit"
    cbxNameItem4.AddItem "Dress"
    cbxNameItem4.AddItem "Regular Skirt"
    cbxNameItem4.AddItem "Skirt With Hook"
    cbxNameItem4.AddItem "Men 's Suit 2Pc"
    cbxNameItem4.AddItem "Men 's Suit 3Pc"
    cbxNameItem4.AddItem "Sweaters"
    cbxNameItem4.AddItem "Silk Shirt"
    cbxNameItem4.AddItem "Tie"
    cbxNameItem4.AddItem "Coat"
    cbxNameItem4.AddItem "Jacket"
    cbxNameItem4.AddItem "Swede"

    Rem Create the orders status
    cbxOrderStatus.AddItem "Processing"
    cbxOrderStatus.AddItem "Ready"
    cbxOrderStatus.AddItem "Picked Up"
End Sub

```

3. Close Microsoft Visual Basic
4. Save the document

Accessing an Item in a Collection

The items of a collection are organized in an arranged sequence where each item holds a specific index. the first item in the collection holds an index of 1. The second item holds an index of 2, and so on.

To give you access to the items of a collection, the **Collection** class is equipped with a property named **Item**. There are two ways you can use this property.

To formally use the **Item** property, type the name of the collection object, followed by the period operator, followed by **Item** and optional parentheses. After the Item property or inside its parentheses, type the index of the desired item. Here is an example:

```

Sub Exercise()
    Dim Employees As Collection

```

```

Set Employees = New Collection

Employees.Add "Patricia Katts"
Employees.Add "James Wiley"
Employees.Add "Gertrude Monay"
Employees.Add "Helene Mukoko"

Employees.Item 2
End Sub

```

Remember that you can also use parentheses:

```

Sub Exercise()
Dim Employees As Collection

Set Employees = New Collection

Employees.Add "Patricia Katts"
Employees.Add "James Wiley"
Employees.Add "Gertrude Monay"
Employees.Add "Helene Mukoko"

Employees.Item (2)
End Sub

```

Instead of using the Item property, you can apply the index directly to the collection object. Here are examples:

```

Sub Exercise()
Dim Employees As Collection

Set Employees = New Collection

Employees.Add "Patricia Katts"
Employees.Add "James Wiley"
Employees.Add "Gertrude Monay"
Employees.Add "Helene Mukoko"

Employees.Item 2
Employees.Item (2)

Employees 2
Employees (2)
End Sub

```

All these four techniques (notations) give you access to the item whose index you provided.

Removing an Item From a Collection

As opposed to adding a new item, you can delete one. To support this operation, the Collection class is equipped with a method name Remove. Its syntax is:

```
Public Sub Remove(Index As Integer)
```

This method takes one argument. When calling it, pass the index of the item you want to delete. Here is an example:

```

Sub Exercise()
Dim Employees As Collection

Set Employees = New Collection

Employees.Add "Patricia Katts"
Employees.Add "James Wiley"
Employees.Add "Gertrude Monay"
Employees.Add "Helene Mukoko"

Employees.Remove 2
End Sub

```

This code deletes the second item in the collection.

The Number of Items in a Collection

When you start a new collection, obviously it is empty and its number of items is 0. To keep track of the number of items in a collection, the **Collection** class is equipped with a property named **Count** whose type is an integer. Remember that all built-in collection classes inherit their behavior from the Collection class. This means that the built-in collection classes are equipped with a property named **Count** to hold their number of items.

We saw how you can add new items to a collection. Every time you add a new item to the collection, the **Count** property increases by 1.

We also know how to remove an item from a collection. Whenever an existing item is deleted, the value of the **Count** property is decreased by 1.

At anytime, to know the number of items that a collection is currently holding, get the value of its **Count** property.

[Previous](#)

Copyright © 2008-2010 FunctionX



Workbooks

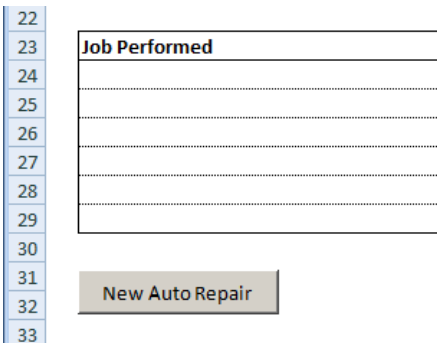
Workbooks Fundamentals

Introduction

When you start Microsoft Excel, it immediately creates a new workbook. You can start working on it and, eventually, you can save it. You are then said to save the workbook. On the other hand, if you have an existing workbook somewhere in the computer or from an attached document sent from a message to you, you can open it as a document.

❖ Practical Learning: Introducing Workbooks

1. Start Microsoft Excel
2. From the resources that accompany these lessons, open the CPAR1 workbook
3. To save it, press F12
4. In the Save As Type combo box, select Excel Macro-Enabled Workbook (*.xlsm)
5. Accept the name of the file and click Save
6. On the ribbon, click Developer
7. In the Controls section, click Insert
8. In the ActiveX Controls section, click Command Button (ActiveX Control)
9. Click the empty area in the lower-left section of the worksheet
10. Right-click the newly added button and click Properties
11. Change its properties as follows:
(Name): cmdNewAutoRepair
Caption: New Auto Repair
12. Move and enlarge the button



13. Right-click the button and click View Code
14. Write the code as follows:

```
Option Explicit

Private AutoRepairExists As Boolean

Private Sub cmdNewAutoRepair_Click()
    AutoRepairExists = False
End Sub
```

15. Return to Microsoft Excel

Referring to a Workbook

In the VBA language, a workbook is an object that belongs to a collection called **Workbooks**. Each workbook of the **Workbooks** collection is an object of type **Workbook**, which is a class.

As seen in the previous lesson with regards to collections, each workbook of the **Workbooks** collection can be identified using the **Item** property. To programmatically refer to a workbook, access the **Item** property and pass either the index or the file name of the workbook to it.

After referring to a workbook, if you want to perform an action on it, you must get a reference to it. To do this, declare a **Workbook** variable and assign the calling **Item()** to it. This would be done as follows:

```
Private Sub cmdSelectWorkbook_Click()
    Dim SchoolRecords As Workbook
```

```
Set SchoolRecords = Workbooks.Item(2)
End Sub
```

Creating a Workbook

When it starts, Microsoft Excel creates a default blank workbook for you. Instead of using an existing workbook or while you are working on another workbook, at any time, you can create a new workbook.

As mentioned already, a workbook is an object of type **Workbook** and it is part of the **Workbooks** collection. To support the ability to create a new workbook, the **Workbooks** collection is equipped with a method named **Add**. Its syntax is:

```
Workbooks.Add(Template) As Workbook
```

You start with the **Workbooks** class, a period, and the **Add** method. This method takes only one argument but the argument is optional. This means that you can call the method without an argument and without parentheses. Here is an example:

```
Private Sub cmdNewWorkbook_Click()
    Workbooks.Add
End Sub
```

When the method is called like this, a new workbook would be created and presented to you. After creating a workbook, you may want to change some of its characteristics. To prepare for this, notice that the **Add()** method returns a **Workbook** object. Therefore, when creating a workbook, get a reference to it. To do this, assign the called method to a **Workbook** variable. Here is an example:

```
Private Sub cmdNewWorkbook_Click()
    Dim SchoolRecords As Workbook

    Set SchoolRecords = Workbooks.Add
End Sub
```

After doing this, you can then use the new variable to change the properties of the workbook.

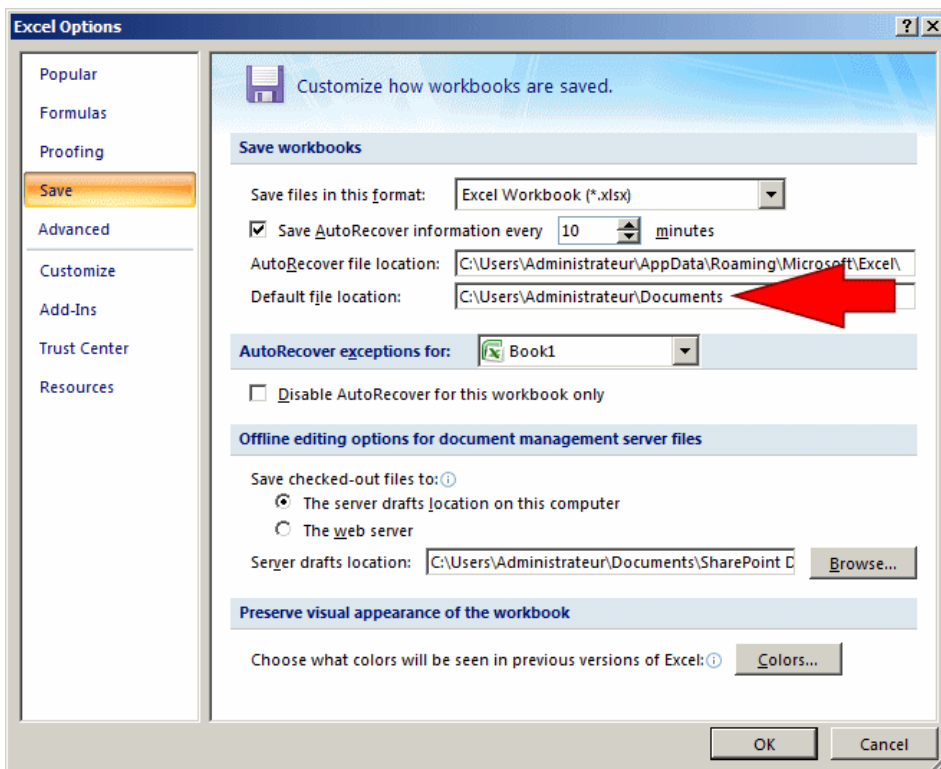
Saving or Opening a Workbook

Introduction

After working on a new workbook, you can save it. After programmatically creating a workbook, if you want to keep it when the user closes Microsoft Excel or when the computer shuts down, you must save it. You and the user have the option of using the Save As dialog box.

The Default File Location

When the user starts saving a file, the Save As dialog box displays, showing the contents of the (My) Documents folder. To find out what the default folder is, you can click the Office button and click Excel Options. In the Excel Options dialog box, check the content of the Default File Location text box:



To support the ability to programmatically change the default folder, the **Application** class is equipped with a property named **DefaultFilePath**. Therefore, to programmatically specify the default folder, assign its string to the **Application.DefaultFilePath** property. Here is an example:

```
Private Sub Exercise()
    Application.DefaultFilePath = "C:\Georgetown Dry Cleaning Services"
End Sub
```

Saving a Workbook

To visually save a workbook, you can click the Office Button and click Save. You can also press Ctrl + S. If the document was saved already, it would be saved behind the scenes without your doing anything else.

To support the ability to programmatically save a workbook, the **Workbook** class is equipped with a method named **Save**. Its syntax is:

```
Workbook.Save()
```

As you can see, this method takes no argument. If you click the Office Button and click Save or if you call the **Workbook.Save()** method on a work that was not saved yet, you would be prompted to provide a name to the workbook.

To save a workbook to a different location, you can click the Office Button, position the mouse on Save As and select from the presented options. You can also press F12. To assist you with programmatically saving a workbook, the **Workbook** class is equipped with a method named SaveAs. Its syntax is:

```
Workbook.SaveAs(FileName,
                 FileFormat,
                 Password,
                 WriteResPassword,
                 ReadOnlyRecommended,
                 CreateBackup,
                 AccessMode,
                 ConflictResolution,
                 AddToMru,
                 TextCodepage,
                 TextVisualLayout,
                 Local)
```

The first argument is the only required one. It holds the name or path to the file. Therefore, you can provide only a name of the file with extension when you call it. Here is an example:

```
Private Sub cmdNewWorkbook_Click()
    Dim SchoolRecords As Workbook

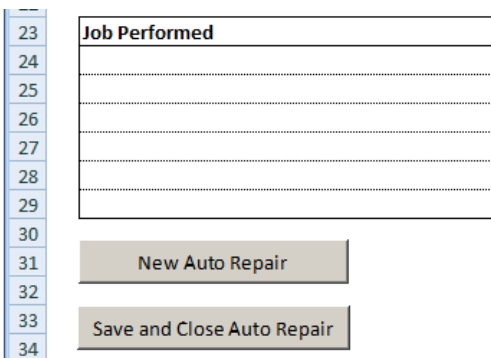
    Set SchoolRecords = Workbooks.Add

    SchoolRecords.SaveAs "SchoolRecords.xlsx"
End Sub
```

If you provide only the name of a file when calling this method, the new workbook would be saved in the current directory or in My Documents (Documents in Windows Vista). If you want, an alternative is to provide a complete path to the file.

❖ Practical Learning: Saving a Workbook

1. In the Controls section of the Ribbon, click Insert
2. In the ActiveX Controls section, click Command Button
3. On the worksheet, click under the previously added button
4. Using the Properties window, change the characteristics of the button as follows:
(Name): cmdSaveAutoRepair
Caption: Save and Close Auto Repair
5. Move and enlarge the button appropriately:



6. Right-click the button and click View Code
7. Write its code as follows:

```
Private Sub cmdSaveAutoRepair_Click()
    ActiveWorkbook.Save
End Sub
```

8. Return to Microsoft Excel

Saving a Workbook for the Web

To save a workbook for the web, pass the first and the second argument of the **Workbook.SaveAs()** method:

```
Workbook.SaveAs(FileName, FileFormat)
```

In this case, pass the second argument as xHTML. Here is an example:

```
Sub Exercise()
    Workbooks(1).SaveAs "Affiche10.htm", xlHtml
End Sub
```

Opening a Workbook

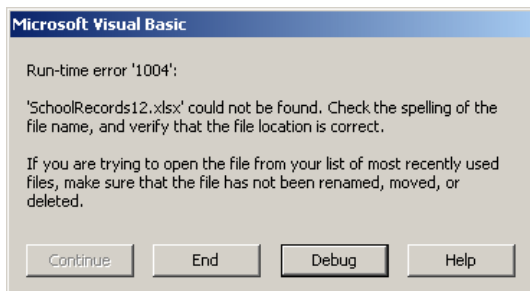
Microsoft Excel is a multiple document interface (MDI) application. This means that you can open many workbooks at the same time and be limited only by the memory on your computer. For this reason, the ability to programmatically open a workbook is handled by the **Workbooks** collection. To support this, the **Workbooks** class is equipped with a method named **Open**. Its syntax is:

```
Workbooks.Open(FileName,
    UpdateLinks,
    ReadOnly,
    Format,
    Password,
    WriteResPassword,
    IgnoreReadOnlyRecommended,
    Origin,
    Delimiter,
    Editable,
    Notify,
    Converter,
    AddToMru,
    Local,
    CorruptLoad)
```

FileName is the only required argument. When calling this method, you must provide the name of the file or its path. This means that you can provide a file name with its extension. Here is an example:

```
Private Sub cmdOpenWorkbook_Click()
    Workbooks.Open "SchoolRecords.xlsx"
End Sub
```

If you provide only the name of a file, Microsoft Excel would look for it in the current directory or in My Documents (Documents in Windows Vista). If Microsoft Excel cannot find the file, you would receive an error:



As you can imagine, a better alternative is to provide a complete path to the file.

❖ Practical Learning: Opening a Workbook

1. In the Controls section of the [Ribbon](#), click Insert
2. In the ActiveX Controls section, click Command Button
3. On the worksheet, click on the right side of Invoice #
4. Using the Properties window, change the characteristics of the button as follows:
(Name): cmdOpenAutoRepair
Caption: open Auto Repair
5. Move and enlarge the button appropriately:

Part #	Part Name	Unit Price	Qty	Sub Total

Job Performed	Rate

Total Parts:	\$ -
Total Labor:	\$ -
Tax Rate:	
Tax Amount:	\$ -
Order Total:	\$ -

- Right-click the button and click View Code
- Write its code as follows:

```
Private Sub cmdOpenAutoRepair_Click()
    Workbooks.Open = "1000.xlsm"
End Sub
```

Closing Workbooks

Closing a Workbook

After using a workbook or to dismiss a document you don't need, you can close it. To support this operation, the Workbook class is equipped with a method named Close. Its syntax is:

```
Public Sub Close(Optional ByVal SaveChanges As Boolean,
    Optional ByVal Filename As String,
    Optional ByVal RouteWorkbook As Boolean)
```

All three arguments are optional. The first argument indicates whether you want to save the changes, if any have been made on the workbook since it was opened. If no change had been made since the time the workbook was created or since the last time it was opened, this argument is not considered.

If the first argument is set to True and the workbook has changes that need to be save, the second argument specifies the name of the file to save the workbook to.

The third argument specifies whether the workbook should be sent to the next user.

❖ Practical Learning: Closing a Workbook

- In the Object combo box, select cmdSaveAutorepair
- Change its code as follows:

```
Private Sub cmdSaveAutoRepair_Click()
    ActiveWorkbook.Save
    ActiveWorkbook.Close
End Sub
```

- Return to Microsoft Excel

If you have many workbooks you don't need, you can close all of them. To support this operation, the `Workbooks` collection class is equipped with a method named `Close`. Its syntax is:

```
Public Sub Workbooks.Close()
```

This method takes no argument. When called, it closes all workbooks that are currently opened in Microsoft Excel.

Microsoft Excel as an MDI

Microsoft Excel is a multiple document interface (MDI). This means that the application allows you to switch from one workbook to another, or be able to display all of them sharing the same screen.

When many workbooks have been opened in, to display many of them, you can arrange them in:

- **Tiled:**

```
Sub Exercise()
    Windows.Arrange ArrangeStyle:=xlTiled
End Sub
```

- **Horizontal:**

```
Sub Exercise()
    Windows.Arrange ArrangeStyle:=xlHorizontal
End Sub
```

- **Vertically:**

```
Sub Exercise()
    Windows.Arrange ArrangeStyle:=xlVertical
End Sub
```

- **Cascade:**

```
Sub Exercise()
    Windows.Arrange ArrangeStyle:=xlCascade
End Sub
```

Accessing a Workbook

To access a workbook, the `Workbook` class is equipped with a method named `Activate`. Its syntax is:

```
Workbook.Activate()
```

This method takes no argument. Therefore, to call it, you can get a reference to the workbook you want to access, then call the `Activate()` method. Here is an example:

```
Private Sub cmdSelectWorkbook_Click()
    Dim SchoolRecords As Workbook

    Set SchoolRecords = Workbooks.Item(2)
    SchoolRecords.Activate
End Sub
```

You can also do this with less code by applying the index directly to the `Workbooks` collection. Here is an example:

```
Private Sub cmdSelectWorkbook_Click()
    Workbooks(2).Activate
End Sub
```

Viewing Many Workbooks

If you create or open many workbooks and while you are working on them, each is represented on the taskbar by a button. To programmatically refer to a workbook, access the `Item` property and pass either the index or the file name of the workbook to it. Here is an example:

```
Private Sub cmdSelectWorkbook_Click()
    Workbooks.Item (2)
End Sub
```

After referring to a workbook, if you want to perform an action on it, you must get a reference to it. To do this, declare a `Workbook` variable and assign the calling `Item()` to it. This would be done as follows:

```
Private Sub cmdSelectWorkbook_Click()
    Dim SchoolRecords As Workbook

    Set SchoolRecords = Workbooks.Item(2)
End Sub
```




Worksheets

Worksheets Fundamentals

Introduction

A worksheet is a document in Microsoft Excel. A worksheet is an object created inside a workbook. That is, a workbook is a series of worksheets that are treated as a group.

❖ Practical Learning: Introducing Worksheets

1. Start Microsoft Excel or a new document
2. To save the document, press Ctrl + S
3. Save it as **ROSH1**
4. On the [Ribbon](#), click Developer



5. In the Code section, click the Visual Basic button
6. To create a form, on the main menu of Microsoft Visual Basic, click Insert -> UserForm
7. Right-click the form and click Properties
8. Change its Caption to **Red Oak High School - Management**

Identifying a Worksheet

A worksheet is an object of type **Worksheet**. The various worksheets you will use are stored in a collection called **Worksheets**. Another name for the collection that contains the worksheets is called **Sheets**. In most cases, you can use either of these two collections. Each worksheet is an object of type **Worksheet**.

Referring to a Worksheet

In the [previous lesson](#), we saw that, if you have only one workbook opened, to refer to it, you can pass an index of 1 to the **Item** property of the **Workbooks** collection to access its **Workbook** object. Here is an example:

```
Sub Exercise()  
    Workbooks.Item(1)  
End Sub
```

You can omit the Item name if you want and you would get the same result:

```
Sub Exercise()  
    Workbooks(1)  
End Sub
```

Because the worksheets of a document are part of the workbook that is opened, to support them, the **Workbook** class is equipped with a property named **Worksheets** or **Sheets**. Therefore, after identifying the workbook, use the period operator to access the **Worksheets** or the **Sheets** property. Here is an example:

```
Sub Exercise()  
    Workbooks.Item(1).Sheets  
End Sub
```

As mentioned already, the worksheets are stored in the **Worksheets** collection, which is actually a class. Each worksheet can be located based on an indexed property named **Item**. The **Item** property is a natural number that starts at 1. The most left worksheet has an index of 1. The second worksheet from left has an index of 2, and so on. To access a worksheet, type one of the **Worksheets** or **Sheets** collections, followed by the period operator, followed by **Item()** and, between the parentheses, type the index of the worksheet you want. For example, the following code will access the second worksheet from left:

```
Private Sub Exercise()  
    Workbooks.Item(1).Sheets.Item(2)  
End Sub
```

Just as we saw that you can omit the **Item** word on the **Workbooks** object, you can also omit it on the **Worksheets** or the **Sheets** object. This can be done as follows:

```
Sub Exercise()  
    Workbooks.Item(1).Worksheets(2)  
End Sub
```

Or as follows:

```
Sub Exercise()
    Workbooks(1).Worksheets(2)
End Sub
```

Each tab of a worksheet has a label known as its name. By default, the most left tab is labeled Sheet1. The second tab from left is labeled Sheet2. To refer to a worksheet using its label, call the **Worksheets** or the **Sheets** collection and pass the label of the tab you want, as a string. Here is an example that refers to the worksheet labeled Sheet3:

```
Sub Exercise()
    Workbooks.Item(1).Sheets.Item("Sheet3")
End Sub
```

On all the code we have written so far, we were getting a worksheet from the currently opened workbook. As mentioned already, by default, when Microsoft Excel starts, it creates a default workbook and gets a **Workbooks.Item(1)** reference. This means that you do not have to indicate that you are referring to the current workbook: it is already available. Consequently, in your code, you can omit **Workbooks.Item(1)** or **Workbooks(1)**. Here is an example:

```
Sub Exercise()
    Sheets.Item("Sheet3")
End Sub
```

Getting a Reference to a Worksheet

In the above code segments, we assumed that you only want to perform an action on a worksheet and move on. Sometimes you may want to get a reference to a worksheet. To do this, declare a variable of type Worksheet. To initialize it, access the desired worksheet from the workbook using the **Item** property and assign it to the variable using the **Set** operator. Here is an example that gets a reference to the second worksheet of the currently opened workbook and stores that reference to a variable:

```
Sub Exercise()
    Dim Second As Worksheet

    Set Second = Workbooks.Item(1).Sheets.Item(2)
End Sub
```

Selecting a Worksheet

To select a worksheet, access the **Sheets** collection, pass the name of the desired worksheet as string, and call **Select**. Here is an example that selects a worksheet labeled **Sheet1**:

```
Private Sub Exercise()
    Sheets("Sheet1").Select
End Sub
```

The worksheet that is selected and that you are currently working on is called the active worksheet. It is identified as the **ActiveSheet** object (it is actually a property of the current document).

Worksheets Names

To rename a worksheet, pass its index or its default name as a string to the **Sheets** (or the **Worksheets**) collection, then access the **Name** property of the collection and assign the desired name. Here is an example:

```
Private Sub Exercise()
    Sheets("Sheet1").Name = "Employees Records"
End Sub
```

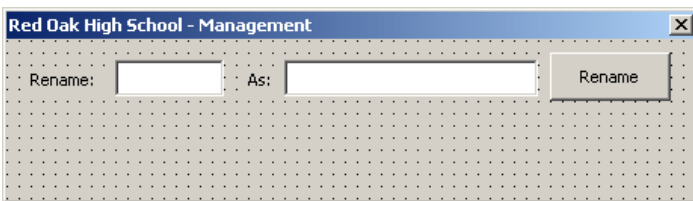
This code will change the name of the Sheet1 worksheet to Employees Records.

As we saw earlier, you can refer to, or select, a worksheet, using its name. If you had renamed a worksheet, you can use that name to select it. Here is an example that selects a worksheet named Tuition Reimbursement:

```
Private Sub Exercise()
    Sheets("Tuition Reimbursement").Select
End Sub
```

❖ Practical Learning: Naming Worksheets

1. Design the form as follows:



Control	Name	Caption
Label		Rename:
TextBox	txtSheetOldName	
Label		As

TextBox	txtNewName	
CommandButton	cmdRename	Rename

2. Double-click the button and implement its Click event as follows:

```
Private Sub cmdRename_Click()
    Worksheets(txtSheetOldName.Text).Name = txtSheetNewName.Text
    txtSheetOldName.Text = ""
    txtSheetNewName.Text = ""
End Sub
```

3. To use the form, on the main menu of Visual Basic, click Run -> Run Sub/UserForm
 4. In the Rename text box, type **Sheet1**
 5. In the As text box, type **Student Registration**

6. Click Rename and notice that the To rename the first worksheet, double-click the Sheet1 tab to put it in edit mode
 7. In the Rename text box, type **Sheet2**
 8. In the As text box, type **Emergency Information** and click Rename

9. Close the form and return to Microsoft Visual Basic

Working on Many Worksheets

Freezing a Cell or More Rows

You can use a column to freeze its cells. To freeze or unfreeze a cell, call the **ActiveWindow** object and access its **FreezePanes** property, which is Boolean. If you set it to True, the window is divided in four parts based on the cell that either is currently selected or you will have indicated. Here is an example of using it:

```
Sub Freezing()
    ActiveWindow.FreezePanes = True
End Sub
```

Splitting the Interface

To split a worksheet, use the **ActiveWindow** object and access its Boolean **Split** property. To split, set this property to true:

```
Sub Splitting()
    ActiveWindow.Split = True
End Sub
```

To un-split, set this property to False.

The Sequence of Worksheets

To move a worksheet, use the **Move()** method of the **Worksheets** or the **Sheets** collection. The syntax of this method is:

```
Worksheets(Index).Move(Before, After)
```

Both arguments are optional. If you don't specify any argument, Microsoft Visual Basic would create a new workbook with one worksheet using the index passed to the collection with a copy of that worksheet. Suppose you are (already) working on a workbook that contains a few worksheets named Sheet1, Sheet2, and Sheet3. If you call this method on a collection with the index set to one of these worksheets, Microsoft Excel would make a copy of that worksheet, create a new workbook with one worksheet that contains a copy of that worksheet. For example, the following code will create a new workbook that contains a copy of the Sheet2 of the current workbook:

```
Private Sub CommandButton1_Click()
    Sheets.Item("Sheet2").Move
End Sub
```

In this case, the name of the worksheet you are passing as argument must exist. Otherwise you would receive an error. Instead of using the name of the worksheet, you can pass the numeric index of the worksheet that you want to copy. For example, the following code will create a new workbook that contains one worksheet named Sheet3:

```
Private Sub CommandButton1_Click()
    Sheets.Item(3).Move
End Sub
```

If calling the **Item** property, make sure the index is valid, otherwise you would receive an error.

To actually move a worksheet, you must specify whether it would be positioned to the left or the right of an existing worksheet. To position a worksheet to the left of a worksheet, assign it the *Before* factor. To position a worksheet to the right of a worksheet, assign it the *After* argument. Consider the following code:

```
Private Sub cmdMove_Click()
    Worksheets("Sheet3").Move After:=Worksheets("Sheet1")
End Sub
```

This code will move the worksheet named Sheet3 to the right of a worksheet named Sheet1.

Adding New Worksheets

To create a new worksheet, you can specify whether you want it to precede or succeed an existing worksheet. To support creating a new worksheet, call the **Add()** method of the **Worksheets** or the **Sheets** collection. Its syntax is:

```
Workbook.Sheets.Add(Before, After, Count, Type)
```

All of these arguments are optional. This means that you can call this method as follows:

```
Private Sub cmdNewWorksheet_Click()
    Sheets.Add
End Sub
```

If you call the method like that, a new worksheet would be created and added to the left side of the active worksheet.

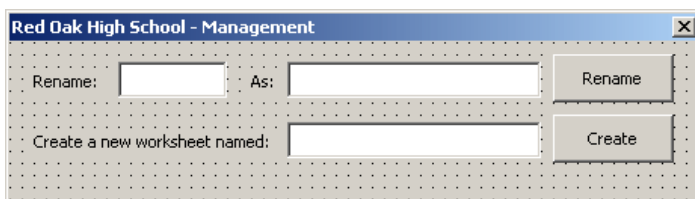
If you want to create a new worksheet on the left side of any worksheet you want, you can first select that worksheet and call the **Add()** method. For example, suppose you have three worksheets named Sheet1, Sheet2, and Sheet3 from left to right and you want to insert a new worksheet between Sheet2 and Sheet3, you can use code as follows:

```
Private Sub cmdNewWorksheet_Click()
    Sheets("Sheet2").Select
    Sheets.Add
End Sub
```

To be more precise, you can specify whether the new worksheet will be positioned to the left or to the right of another worksheet used as reference.

❖ Practical Learning: Creating Worksheets

1. Change the design of the form as follows:

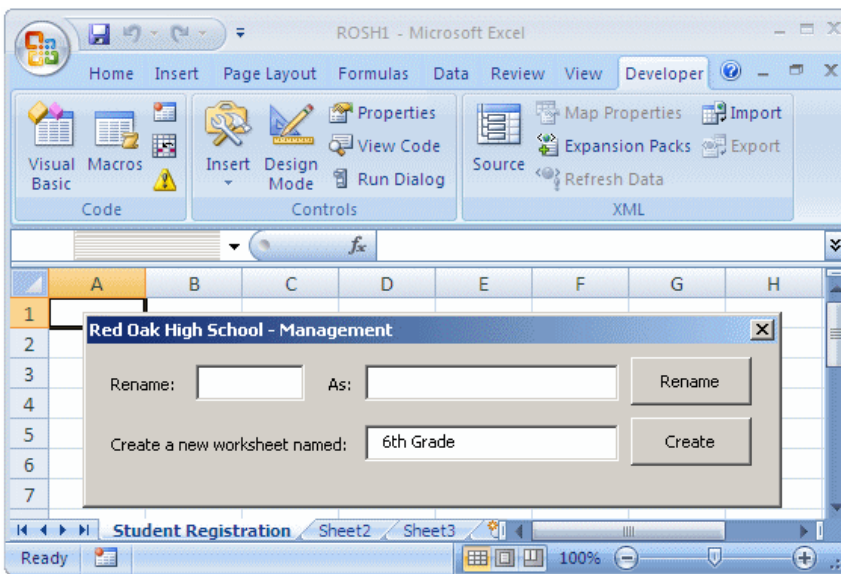


Control	Name	Caption
Label		Create a new worksheet named:
TextBox	txtNewWorksheet	
CommandButton	cmdNewWorksheet	

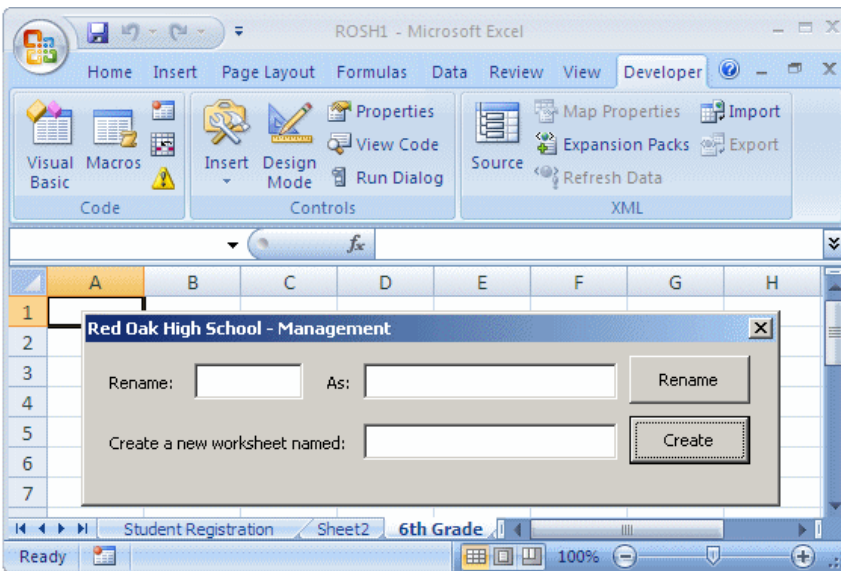
2. Double-click the Create button and implement its Click event as follows:

```
Private Sub cmdNewWorksheet_Click()
    Worksheets.Add Before:=Worksheets("Sheet3")
    Worksheets(Worksheets.Count - 1).Name = txtNewWorksheet.Text
    txtNewWorksheet.Text = ""
End Sub
```

3. To test the code, on the Standard toolbar of Microsoft Visual Basic, click the Run Sub/UserForm button
4. Click the Create A New Worksheet Named text box and type **6th Grade**



- Click Create



- In the same way, create new worksheets named **5th Grade**, **4th Grade**, **3rd Grade**, **2nd Grade**, and **1st Grade**
- Close the form

Removing Worksheets

To remove a worksheet, call the **Delete()** method of its collection. When calling this method, pass the name of the worksheet you want to remove to the collection.

❖ Practical Learning: Deleting Worksheets

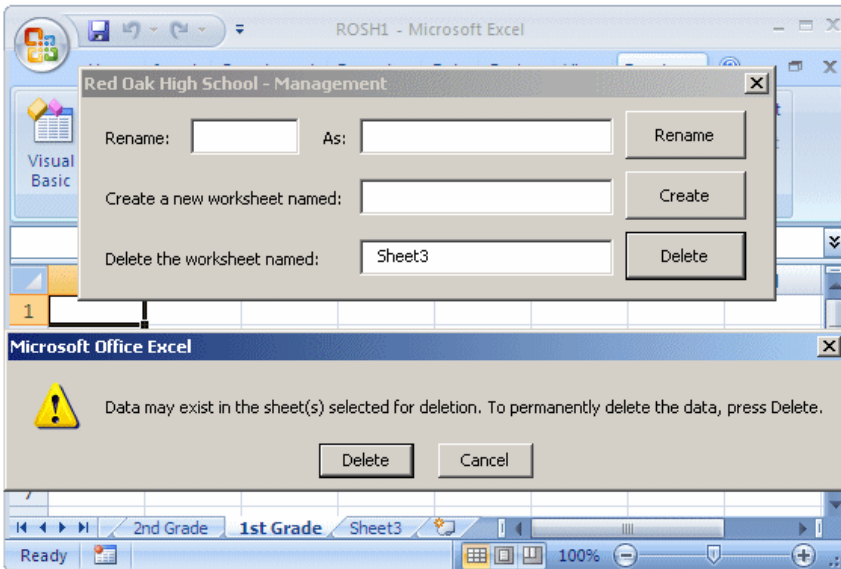
- Change the design of the form as follows:

Control	Name	Caption
Label		Delete the worksheet named:
TextBox	txtRemoveSheet	
CommandButton	cmdDelete	Delete

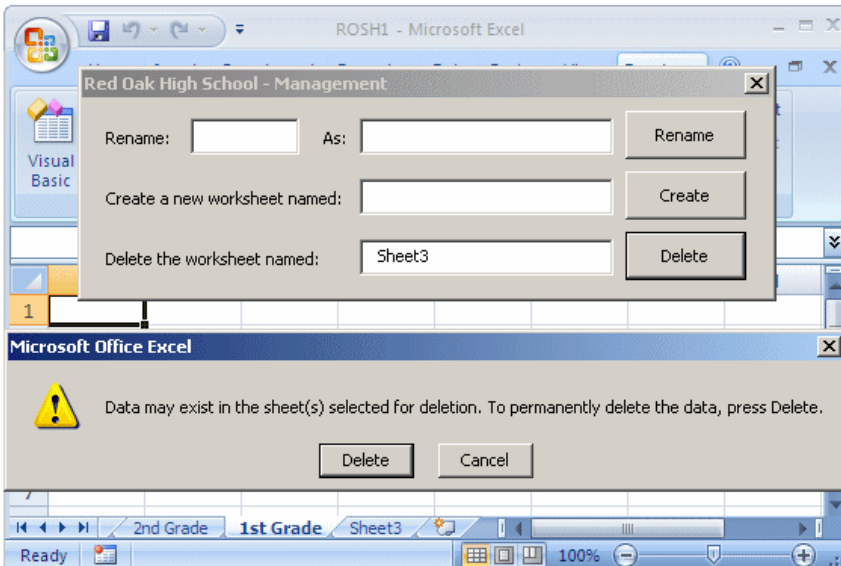
- Double-click the Create button and implement its Click event as follows:

```
Private Sub cmdRemoveSheet_Click()
    Worksheets("Sheet3").Delete
    txtRemoveSheet.Text = ""
End Sub
```

- To test the code, on the Standard toolbar of Microsoft Visual Basic, click the Run Sub/UserForm button
- Click the Delete The Worksheet Named text box, type Sheet3



- Click Delete



- After reading the warning, click Delete
- In the same way, delete the worksheet named Sheet2
- Close the form
- Save the document

Accessing a Worksheet

To access a worksheet, the **Worksheet** class is equipped with a method named **Activate**. Its syntax is:

```
Worksheet.Activate()
```

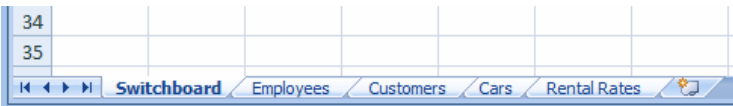
This method takes no argument. To call it, get a reference to the worksheet you want to access and call the **Activate()** method. You can also do this with less code by applying the index directly to the **Worksheets** collection. Here is an example:

```
Private Sub cmdSelectWorkbook_Click()  
    Worksheets(2).Activate  
End Sub
```

❖ Practical Learning: Introducing File Processing

- To create a new workbook, press Ctrl + N
- Double-click Sheet1, type **Switchboard**
- Double-click Sheet2 and type **Employees**
- Double-click Sheet3 and type **Customers**
- Click the next sheet tab (the Insert Worksheet)
- Double-click the new sheet tab and type **Cars**
- Click the next sheet tab (the Insert Worksheet)

8. Double-click the new sheet tab and type **Rental Rates**
9. Click the Switchboard tab
10. Press and hold Shift
11. Click the Rental Rates tab
12. Release Shift



13. Click Cell B2 and type **Bethesda Car Rental**
14. Click the Enter button
15. Click the Employees sheet tab
16. Click Cell B6 and type **Employee #**
17. **Create a few employees**
18. Click the Customers sheet tab
19. Click Cell B6 and type **Driver's Lic. #**
20. **Create a few customers**
21. Click the Cars sheet tab
22. Click Cell B6 and type **Tag Number**
23. **Create a few cars**
24. On the Ribbon, click Developer



25. In the Code section of the Ribbon, click Visual Basic
26. On the main menu of Microsoft Visual Basic, click Insert -> UserForm
27. If the Properties window is not available, on the main menu, click View -> Properties Window. In the Properties window, click (Name) and type **frmRentalOrder**
28. Click Caption and type Bethesda **Car Rental - Order Processing - Rental Order**
29. Add a Command Button to the form and change its properties as follows:
(Name): **cmdEmployees**
Caption: **Employees**
30. Right-click Employees button and click View Code
31. Implement the event as follows:

```
Private Sub cmdEmployees_Click()  
    Worksheets(2).Activate  
End Sub
```

32. Display the form again
33. Add another Command Button and change its characteristics as follows:
(Name): **cmdCustomers**
Caption: **Customers**
34. Double-click the Customers button and implement the event as follows:

```
Private Sub cmdCustomers_Click()  
    Worksheets(3).Activate  
End Sub
```

35. Return to the form
36. Add another Command Button and change its characteristics as follows:
(Name): **cmdCars**
Caption: **Cars**
37. Double-click the Cars button and implement the event as follows:

```
Private Sub txtCars_Click()  
    Worksheets(4).Activate  
End Sub
```

38. Show the form one more time
39. Add another Command Button and change its characteristics as follows:
(Name): **cmdRentalRates**
Caption: **Rental Rates**
40. Double-click the new button and implement its Click event as follows:

```
Private Sub cmdRentalRates_Click()
```

41. On the Standard toolbar, click the Run Sub/UserForm button
42. Click each button and notice that the corresponding worksheet displays
43. Close the form
44. Close Microsoft Visual Basic
45. Close Microsoft Excel
46. If asked whether you want to save, click No

[Previous](#)

Copyright © 2009-2010 FunctionX, Inc.

[Next](#)



The Columns of a Worksheet

Columns Fundamentals

Introduction

A worksheet is arranged in columns. In VBA for Microsoft Excel, to programmatically refer to a column, you will use a collection. In your code, one of the classes you can use to access a column is named **Range**. As we will see in various examples, you can directly access this class.

If you want to get a reference to a column or a group of columns, declare a variable of type **Range**:

```
Sub Exercise()  
    Dim Series As Range  
End Sub
```

To initialize the variable, you will identify the workbooks and the worksheets you are using. We will see various examples later on.

The Columns Collection

When Microsoft Excel starts, it displays the columns that have already been created. To support its group of columns, the **Worksheet** class is equipped with a property named **Columns**. There are various ways you can identify a column: using its index or using its label.

Identifying a Column

A Column by its Index

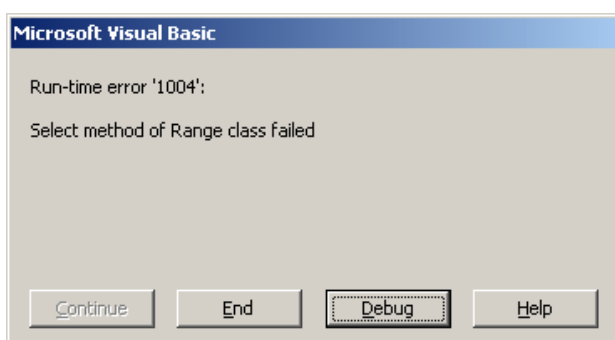
The columns on a worksheet are arranged by positions. A position is in fact referred to as the index of the column. The first column on the left has the index 1, the second from left has the index 2, and so on. Based on this, to refer to its column, pass its index to the parentheses of the Columns collection. Here are two examples:

```
Sub Exercise()  
    REM This refers to the first column  
    Workbooks(1).Worksheets(2).Columns(1)  
    ' This refers to the 12th column  
    Workbooks(1).Worksheets(2).Columns(12)  
End Sub
```

In the previous lesson, we saw that you can omit calling the **Workbooks(1)** property to identify the first workbook if you know you are referring to the default workbook. Therefore, the above code can be written as follows:

```
Sub Exercise()  
    REM This refers to the fourth column  
    Worksheets(2).Columns(4)  
End Sub
```

This code now indicates that you are referring to the fourth column in the second worksheet. When this code runs, Microsoft Excel must be displaying the second worksheet. If you run this code while Microsoft Excel is displaying a worksheet other than the second, you would receive an error:



This means that if you try accessing a column from a worksheet other than the one indicated in your code, the code would fail. If you want to access a specific column in any worksheet from the workbook that Microsoft Excel is currently showing, you can omit indicating the worksheet from the **Worksheets** collection. Here is an example:

```
Sub Exercise()
    REM This refers to the fourth column
    Columns(4)
End Sub
```

This time, the code indicates that you are referring to the fourth column of whatever worksheet is currently active.

A Column by its Name

To programmatically refer to a column using its name, pass its letter or combination of letters as a string in the parentheses of the Columns collection. Here are two examples:

```
Sub Exercise()
    Rem This refers to the column named/labeled A
    Columns("A")
    ' This refers to the column named DR
    Columns("DR")
End Sub
```

Adjacent Columns

To programmatically refer to adjacent columns, you can use the **Columns** collection. In its parentheses, type the name of a column that would be on one end of the range, followed by a colon ":", followed by the name of the column that would be on the other end. Here is an example that refers to columns in the range D to G:

```
Sub ColumnReference()
    Rem Refer to the range of columns D to G
    Columns("D:G")
End Sub
```

You can also select columns using the Range class. To do this, in the parentheses type the name of the first column, followed by a colon, followed by the name of the column on the other end. Here is an example:

```
Sub ColumnReference()
    Rem This refers to the columns in the range D to G
    Range("D:H")
End Sub
```

Non-Adjacent Columns

Columns are referred to as non-adjacent when they do not follow each other. For example, columns B, D, and G are non-adjacent. To programmatically refer to non-adjacent columns, use the **Range** collection. In its parentheses, type each name of a column, followed by a colon, followed by the same name of column, then separate these combinations with commas. Here is an example:

```
Sub Exercise()
    Rem This refers to Column H, D, and B
    Range("H:H, D:D, B:B")
End Sub
```

To refer to all columns of a worksheet, use the **Columns** name. Here is an example:

```
Sub Exercise()
    Columns
End Sub
```

Columns Selection

Selecting a Column

To support column selection, the **Column** class is equipped with a method named **Select**. This method does not take any argument. Based on this, to select the fourth column using its indexed, you would use code as follows:

```
Sub Exercise()
    Rem This selects the fourth column
    Columns(4).Select
End Sub
```

To select a column using its name, you would use code as follows:

```
Sub Exercise()
    Rem This selects the column labeled ADH
```

```
Columns("ADH").Select
End Sub
```

When a column has been selected, it is stored in an object called **Selection**. You can then use that object to take an action to apply to the column.

Selecting a Range of Adjacent Columns

To programmatically select a range of columns, in the parentheses of the **Columns** collection, enter the name of the first column on one end, followed by a colon ":", followed the name of the column that will be at the other end. Here is an example:

```
Sub Exercise()
    Rem This selects the range of columns from Column D to Column G
    Columns("D:G").Select
End Sub
```

You can use this same notation to select one column. To do this, use the **Range** collection. In the parentheses of the collection, enter the name of the column, followed by a colon, followed by the same column name. Here is an example:

```
Sub Exercise()
    Rem This selects Column G
    Range("G:G").Select
End Sub
```

Selecting Non-Adjacent Columns

To programmatically select non-adjacent columns, use the technique we saw earlier to refer to non-adjacent columns, then call the **Select** method. Here is an example:

```
Sub Exercise()
    Rem This selects Columns B, D, and H
    Range("H:H, D:D, B:B").Select
End Sub
```

When many columns have been selected (whether adjacent or not), their selection is stored in an object named **Selection**. You can access that object to apply a common action to all selected columns.

Creating Columns

Adding a New Column

To support the creation of columns, the **Column** class is equipped with a method named **Insert**. This method takes no argument. When calling it, you must specify the column that will succeed the new one. Here is an example that will create a new column in the third position and move the columns from 3 to 16384 to the right:

```
Sub CreateColumn()
    Columns(3).Insert
End Sub
```

Adding New Columns

To programmatically add a new column, specify its successor using the **Range** class. Then call the **Insert** method of the **Column** class. Here is an example that creates new columns in places of Columns B, D, and H that are pushed to the right to make place for the new ones:

```
Sub CreateColumns()
    Range("H:H, D:D, B:B").Insert
End Sub
```

Deleting Columns

Deleting a Column

To provide the ability to delete a column, the **Column** class is equipped with a method named **Delete**. This method does not take an argument. To delete a column, use the **Columns** collection to specify the index or the name of the column that will be deleted. Then call the **Delete** method. Here is an example that removes the fourth column. Here is an example:

```
Sub DeleteColumn()
    Columns("D:D").Delete
End Sub
```

Deleting Many Columns

To programmatically delete many adjacent columns, specify their range using the **Columns** collection and call the **Delete** method. Here is an example:

```
Sub DeleteColumns()
    Columns("D:F").Delete
End Sub
```

To delete many non-adjacent columns, use the **Range** class then call the **Delete** method of the **Column** class. Here is an example that deletes Columns C, E, and P:

```
Sub DeleteColumns()
    Range("C:C, E:E, P:P").Delete
End Sub
```

The Width of Columns

Introduction

To support column sizes, the **Column** class is equipped with a property named **ColumnWidth**. Therefore, to programmatically specify the width of a column, access it, then access its **ColumnWidth** property and assign the desired value to it. Here is an example that sets Column C's width to 4.50:

```
Sub Exercise()
    Columns("C").ColumnWidth = 4.5
End Sub
```

Automatically Resizing the Columns

To use **AutoFit Selection**, first select the column(s) and store it (them) in a **Selection** object, access its **Columns** property, then call the **AutoFit** method of the **Columns** property. This can be done as follows:

```
Private Sub Exercise()
    Selection.Columns.AutoFit
End Sub
```

Setting the Width Value of Columns

To specify the widths of many columns, access them using the **Range** class, then access the **ColumnWidth** property, and assign the desired value. Here is an example that sets the widths of Columns C, E, and H to 5 each:

```
Sub Exercise()
    Range("C:C, E:E, H:H").ColumnWidth = 5#
End Sub
```

Hiding, Freezing, and Splitting Columns

Hiding and Revealing Columns

To programmatically hide a column, first select it, then assign **True** to the **Hidden** property of the **EntireColumn** object of **Selection**. Consider the following code:

```
Private Sub Exercise()
    Columns("F:F").Select
    Selection.EntireColumn.Hidden = True
End Sub
```

To unhide a hidden column, assign a **False** value to the **Hidden** property:

```
Private Sub Exercise()
    Columns("F:F").Select
    Selection.EntireColumn.Hidden = False
End Sub
```

Splitting the Columns

To split the columns, call the **ActiveWindow** object, access its **SplitColumn** and assign it the column number. Here is an example:

```
Sub Exercise()
    ActiveWindow.SplitColumn = 4
End Sub
```

To remove the splitting, access the same property of the **ActiveWindow** object and assign 0 to it. Here is an example:

```
Sub Exercise()
    ActiveWindow.SplitColumn = 0
End Sub
```




The Rows of a Worksheet

Rows Fundamentals

Introduction

We already know that a worksheet organizes its information in columns. To show the values in a worksheet, each column holds a particular value that corresponds to another value in the same horizontal range. The group of values that correspond to the same horizontal arrangement is called a row.

Identifying a Row

To support the rows of a worksheet, the **Worksheet** class is equipped with a property named **Rows**. Therefore, to refer to a row, you can use the **Worksheets** collection or the **Worksheet** object to access the **Rows** property. Another way you can refer to rows is by using the **Range** object.

To identify a row, indicate its worksheet and you can pass its number to the parentheses of the **Rows** collection. Here is an example that refers to the 5th row of the second worksheet of the current workbook:

```
Sub Exercise()  
    Worksheets.Item(1).Worksheets.Item(2).Rows(5)  
End Sub
```

As reviewed for the columns, this code would work only if the second worksheet of the current workbook is displaying. If you run it while a worksheet other than the second is active, you would receive an error. To access any row, omit the **Workbooks** and the **Worksheets** collections.

As mentioned already, you can refer to a row using the **Range** object. To do that, pass a string to the **Range** object. In the parentheses, type the number of the row, followed by a colon, followed by the number of the row. Here is an example that refers to Row 4:

```
Sub Exercise()  
    Range("4:4")  
End Sub
```

If you want to refer to the rows more than once, you can declare a variable of type **Range** and initialize it using the **Set** operator and assign it the range you want. Here is an example:

```
Sub Exercise()  
    Dim SeriesOfRows As Range  
  
    Set SeriesOfRows = Workbooks.Item(1).Worksheets.Item("Sheet1").Range("4:4")  
  
    SeriesOfRows.Whatever  
End Sub
```

Identifying a Group of Rows

A group of rows is said to be in a range if they are next to each other. To refer to rows in a range, in the parentheses of the **Rows** collection, pass a string that is made of the number of the row from one end, followed by a colon, followed by the row number of the other end. Here is an example that refers to rows from 2 to 6:

```
Sub Exercise()  
    Rows("2:6")  
End Sub
```

The rows of a group qualify as non-adjacent if they are or they are not positioned next to each other. To refer to non-adjacent rows, pass a string to the **Range** collection. In the parentheses, type the number of each row followed by a colon, followed by the same number. These combinations are separated by commas. Here is an example that refers to Rows 3, 5, and 8:

```
Sub Exercise()  
    Range("3:3, 5:5, 8:8")  
End Sub
```

To refer to all rows of a worksheet, use the **Rows** name. Here is an example:


```
Sub Exercise()  
    Rows  
End Sub
```

Rows Selection

Selecting a Row

To support row selection, the **Row** class is equipped with a method named **Select**. Therefore, to programmatically select a row, access a row from the **Rows** collection using the references we saw earlier. Then call the **Select** method. Here is an example that selects Row 6:

```
Sub Exercise()  
    Rows(6).Select  
End Sub
```

We also saw that you could refer to a row using the Range object. After accessing the row, call the Select method. Here is an example that selects Row 4:

```
Sub Exercise()  
    Range("4:4").Select  
End Sub
```

When a row has been selected, it is stored in an object called **Selection**. You can then use that object to apply an action to the row.

Selecting a Group of Rows

To programmatically select a range of rows, refer to the range using the techniques we saw earlier, then call the **Select** method. Here is an example that selects rows from 2 to 6:

```
Sub Exercise()  
    Rows("2:6").Select  
End Sub
```

To programmatically select non-adjacent rows, refer to them as we saw earlier and call the **Select** method. Here is an example that selects Rows 3, 5, and 8:

```
Sub Exercise()  
    Range("3:3, 5:5, 8:8").Select  
End Sub
```

To programmatically select all rows of a worksheet, call the **Select** method on the Rows collection. Here is an example:

```
Sub Exercise()  
    Rows.Select  
End Sub
```

When many rows have been selected (whether adjacent or not), their selection is stored in an object named **Selection**. You can access that object to apply a common action to all selected rows.

Managing Rows

The Height of a Row

To support the height of a row, the Row object is equipped with a property named **RowHeight**. Therefore, to programmatically specify the height of a row, access the row using a reference as we saw earlier, access its **RowHeight** property and assign the desired value to it. Here is an example that sets the height of Row 6 to 2.50

```
Sub Exercise()  
    Rows(6).RowHeight = 2.5  
End Sub
```

Adding a New Row

To provide the ability to add a new row, the **Row** class is equipped with a method named **Insert**. Therefore, to programmatically add a row, refer to the row that will be positioned below the new one and call the **Insert** method. Here is an example:

```
Sub Exercise()  
    Rows(3).Insert  
End Sub
```

Adding New Rows

To programmatically add new rows, refer to the rows that would be below the new ones, and call the Insert method. Here is an example that will add new rows in positions 3, 6, and 10:

```
Sub Exercise()
    Range("3:3, 6:6, 10:10").Insert
End Sub
```

Removing Rows

Deleting a Row

To support row removal, the **Row** class is equipped with a method named **Delete** that takes no argument. Based on this, to delete a row, access it using a reference as we saw earlier, and call the **Delete** method. Here is an example:

```
Sub Exercise()
    Rows(3).Delete
End Sub
```

Of course, you can use either the Rows collection or the Range object to refer to the row.

Deleting Rows

To delete a group of rows, identify them using the **Range** collection. Then call the **Delete** method. Here is an example:

```
Sub Exercise()
    Range("3:3, 6:6, 10:10").Delete
End Sub
```

Using Rows

Moving Rows

To move a group of rows, access the **Range** collection and identify them. Call the **Cut** method. Access its **Destination** argument to which you will assign the rows where you are moving. Here is an example:

```
Sub Exercise()
    Rows("11:12").Cut Destination:=Rows("16:17")
End Sub
```

Copying and Pasting Rows

To copy a row (or a group of rows), use the **Rows** collection to identify the row(s). Call the Copy method on it. Access the Destination argument and assign the destination row(s) to it. Here is an example:

```
Sub Exercise()
    Rows("10:15").Copy Destination:=Rows("22:27")
End Sub
```

Hiding and Revealing Rows

To programmatically hide a row, first select. Then, access the **Hidden** property of the **EntireRow** object of **Selection**. Here is an example:

```
Private Sub Exercise()
    Rows("6:6").Select
    Selection.EntireRow.Hidden = True
End Sub
```

This code example will hide row 6. In the same way, to hide a group of rows, first select their range, then write **Selection.EntireRow.Hidden = True**.

Splitting the Rows

To split the rows, call the **ActiveWindow** object, access its **SplitRow** and assign it the row number. Here is an example:

```
Sub Exercise()
    ActiveWindow.SplitRow = 4
End Sub
```

To remove the splitting, access the same property of the **ActiveWindow** object and assign 0 to it. Here is an example:

```
Sub Exercise()
    ActiveWindow.SplitRow = 0
End Sub
```

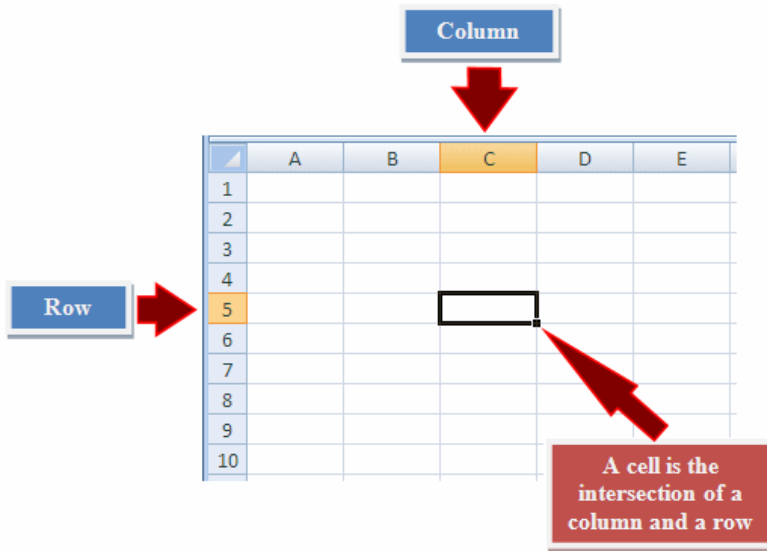



Introduction to Cells

A Cell in a Worksheet

Introduction

A spreadsheet is a series of columns and rows. These columns and rows intersect to create cells:



When Microsoft Excel starts, it creates 16,384 columns and 1,048,576 rows. As a result, a Microsoft Excel spreadsheet has $16,384 * 1,048,576 = 17,179,869,184$ cells available.

The Active Cell

To access a cell, you can click it. This becomes the active cell. In VBA, the active cell is represented by an object named **ActiveCell**.

Referencing Cells

To identify a cell, you can use the **Range** object. In the parentheses of the **Range** object, pass a string that contains the name of the cell. Here is an example that refers to the cell located as D6:

```
Sub Exercise()  
    Workbooks.Item(1).Worksheets.Item("Sheet1").Range("D6")  
End Sub
```

To get a reference to a cell, declare a variable of type Range. To initialize the variable, identify the cell and assign it to the variable using the Set operator. Here is an example:

```
Sub Exercise()  
    Dim Cell As Range  
    Set Cell = Workbooks.Item(1).Worksheets.Item("Sheet1").Range("D6")  
End Sub
```

Cells are referred to as adjacent when they touch each other. To refer to a group of adjacent cells, in the parentheses of the Range object, pass a string that is made of the address of the cell that will be on one corner, followed by a colon, followed by the address of the cell that will be on the other corner. Here is an example:

```
Sub Exercise()  
    Range("B2:H6")  
End Sub
```

You can use this same technique to refer to one cell. To do this, use the same cell address on both sides of the colon. Here is an example:

```
Sub Exercise()  
    Range("D4:D4")  
End Sub
```

Instead of referring to one group of adjacent cells, you can refer to more than one group of non-adjacent cells. To do this, pass a string to the **Range** object. In the string, create each range as you want but separate them with commas. Here is an example:

```
Sub Exercise()
```

```
Range("D2:B5, F8:I14")
End Sub
```

Selecting Cells

Introduction

Before doing anything on a cell or a group of cells, you must first select it. To support cell selection, the **Range** object is equipped with a method named **Select**. Therefore, to programmatically select a cell, after referencing it, call the **Select** method. Here is an example:

```
Sub Exercise()
    Range("D6").Select
End Sub
```

When you have selected a cell, it is stored in an object named **Selection**. You can use this object to take an action on the cell that is currently selected.

Selecting Cells

To programmatically select a group of adjacent cells, refer to the group using the techniques we saw earlier, then call the **Select** method.

To programmatically select all cells of a column, access the **Columns** collection and pass the column name as a string, then call the **Select** method. Here is an example we saw in Lesson 9:

```
Sub Exercise()
    Rem This selects all cells from the fourth column
    Columns(4).Select
End Sub
```

To perform this operation using the name of a column, pass that name as argument. Here is an example that selects all cells from Column ADH:

```
Sub Exercise()
    Rem This selects all cells from the column labeled ADH
    Columns("ADH").Select
End Sub
```

You can also perform this operation using the **Range** object. To do this, use the **Range** collection. In the parentheses of the collection, enter the name of the column, followed by a colon, followed by the same column name. Here is an example:

```
Sub Exercise()
    Rem This selects all cells from Column G
    Range("G:G").Select
End Sub
```

To programmatically select all cells that belong to a group of adjacent columns, in the parentheses of the **Columns** collection, enter the name of the first column on one end, followed by a colon ":", followed the name of the column that will be at the other end. Here is an example:

```
Sub Exercise()
    Rem This selects all cells in the range of columns from Column D to Column G
    Columns("D:G").Select
End Sub
```

To select the cells that belong to a group of non-adjacent columns, use the technique we saw earlier to refer to non-adjacent columns, then call the **Select** method. Here is an example:

```
Sub Exercise()
    Rem This selects the cells from columns B, D, and H
    Range("H:H, D:D, B:B").Select
End Sub
```

To programmatically select all cells that belong to a row, access a row from the **Rows** collection, then call the **Select** method. Here is an example that all cells from Row 6:

```
Sub Exercise()
    Rows(6).Select
End Sub
```

You can also use the **Range** object. After accessing the row, call the **Select** method. Here is an example that selects all cells from Row 4:

```
Sub Exercise()
    Range("4:4").Select
End Sub
```

To select all cells that belong to a range of rows, refer to the range and call the **Select** method. Here is an example that selects all cells that belong to the rows from 2 to 6:

```
Sub Exercise()
    Rows("2:6").Select
End Sub
```

To select all cells that belong to non-adjacent rows, refer to the rows and call the **Select** method. Here is an example that selects all cells belonging to Rows 3, 5, and 8:

```
Sub Exercise()
    Range("3:3, 5:5, 8:8").Select
End Sub
```

To programmatically select cells in the same region, enter their range as a string to the **Range** object, then call the **Select** method. Here is an example:

```
Sub Exercise()  
    Range("B2:H6").Select  
End Sub
```

Remember that you can use the same technique to refer to one cell, thus to select a cell. Here is an example:

```
Sub Exercise()  
    Range("D4:D4").Select  
End Sub
```

To select more than one group of non-adjacent cells, refer to the combination as we saw earlier and call the **Select** method. Here is an example:

```
Sub Exercise()  
    Range("D2:B5, F8:I14").Select  
End Sub
```

To select all cells of a spreadsheet, you can call the **Select** method on the **Rows** collection. Here is an example:

```
Sub Exercise()  
    Rows.Select  
End Sub
```

Instead of the **Rows** collection, you can use the **Columns** collection instead and you would get the same result.

When you have selected a group of cells, the group is stored in an object named **Selection**. You can use this object to take a common action on all of the cells that are currently selected.

The Name of a Cell

We already saw that, to refer to a cell using its name, you can pass that name as a string to the **Range** object.

After creating a name for a group of cells, to refer to those cells using the name, call the **Range** object and pass the name as a string.

The Gridlines and Headings of a Worksheet

Showing the Gridlines of Cells

To show or hide the gridlines, call the **ActiveWindow** and access its **DisplayGridlines** property. This is a Boolean property. If you set its value to **True**, the gridlines appear. If you set it to **False**, the gridlines disappear. Here is an example of using it:

```
Sub Exercise()  
    ActiveWindow.DisplayGridlines = False  
End Sub
```

Showing the Headings of a Worksheet

To show or hide the headers of columns, get the **ActiveWindow** object and access its **DisplayHeadings** Boolean property. To show the headers, set this property to **True**. To hide the headers, set the property to **False**. Here is an example:

```
Sub ShowHeadings()  
    ActiveWindow.DisplayHeadings = False  
End Sub
```

Operations on Cells

Adding Cells

We know that, to insert a column made of (vertical) cells, you can access the **Columns** collection, specify an index in its parentheses, and call the **Insert** method. Here is an example:

```
Sub CreateColumn()  
    Columns(3).Insert  
End Sub
```

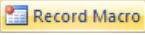
We also know how to create a series of rows made of cells horizontally.

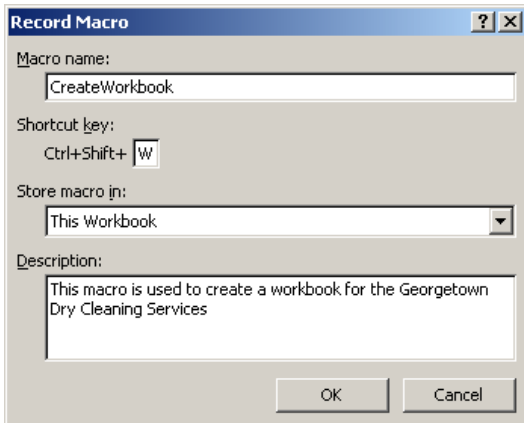
Cell Data Entry

Data entry consists of adding one or more values into one or more cells. This can be done manually, automatically, or programmatically. We already know that a cell that is currently selected in a worksheet is called **ActiveCell**. Therefore, to programmatically add a value to the active cell, assign that value to this object.

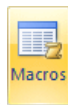
❖ Practical Learning: Introducing Data Entry

1. Start Microsoft Excel

2. On the [Ribbon](#), click Developer
3. In the Code section, click Record Macro 
4. Set the Macro Name to **CreateWorkbook**
5. In the Shortcut Key text box, type **W** to get Ctrl + Shift + W



6. Click OK
7. On the Ribbon, click Stop Recording



8. In the Code section of the Ribbon, click Macros
9. In the Macro dialog box, make sure CreateWorkbook is selected and click Edit
10. Change the code as follows:

Option Explicit

```
Sub CreateWorkbook()
' CreateWorkbook Macro
' This macro is used to create a workbook for the
' Georgetown Dry Cleaning Services

' Keyboard Shortcut: Ctrl+Shift+W

Rem Just in case there is anything on the
Rem worksheet, delete everything
Range("A:K").Delete
Range("1:20").Delete

Rem Create the sections and headings of the worksheet
Range("B2") = "Georgetown Dry Cleaning Services"
Range("B5") = "Order Identification"
Range("B6") = "Receipt #:"
Range("G6") = "Order Status:"
Range("B7") = "Customer Name:"
Range("G7") = "Customer Phone:"

Range("B9") = "Date Left:"
Range("G9") = "Time Left:"
Range("B10") = "Date Expected:"
Range("G10") = "Time Expected:"
Range("B11") = "Date Picked Up:"
Range("G11") = "Time Picked Up:"


Range("B13") = "Items to Clean"
Range("B14") = "Item"
Range("D14") = "Unit Price"
Range("E14") = "Qty"
Range("F14") = "Sub-Total"

Range("B15") = "Shirts"
Range("H15") = "Order Summary"
Range("B16") = "Pants"
Range("B17") = "None"
Range("H17") = "Cleaning Total:"
Range("B18") = "None"
Range("H18") = "Tax Rate:"
Range("I18") = "5.75"
Range("J18") = "%"
Range("B19") = "None"
Range("H19") = "Tax Amount:"
Range("B20") = "None"
Range("H20") = "Order Total:"

Rem Change the widths and heights of some columns and rows
Rem In previous lessons, we learned all these things
Range("E:E, G:G").ColumnWidth = 4
Columns("H").ColumnWidth = 14
Columns("J").ColumnWidth = 1.75

Rows("3").RowHeight = 2
```

```
Rem Hide the gridlines  
ActiveWindow.DisplayGridlines = False  
End Sub
```

11. To return to Microsoft Excel, click the View Microsoft Excel button 
12. To fill the worksheet, press Ctrl + Shift + W

	A	B	C	D	E	F	G	H	I	J
1										
2		Georgetown Dry Cleaning Services								
4										
5		Order Identification								
6		Receipt #:					Order Status:			
7		Customer Name:					Customer Phone:			
8										
9		Date Left:					Time Left:			
10		Date Expected:					Time Expected:			
11		Date Picked Up:					Time Picked Up:			
12										
13		Items to Clean								
14		Item	Unit Price	Qty	Sub-Total					
15		Shirts					Order Summary			
16		Pants								
17		None					Cleaning Total:			
18		None					Tax Rate:	5.75 %		
19		None					Tax Amount:			
20		None					Order Total:			

13. Close Microsoft Excel
14. When asked whether you want to save, click No



Cells Aesthetic Formatting

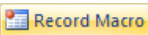
Cell Formatting With a Font

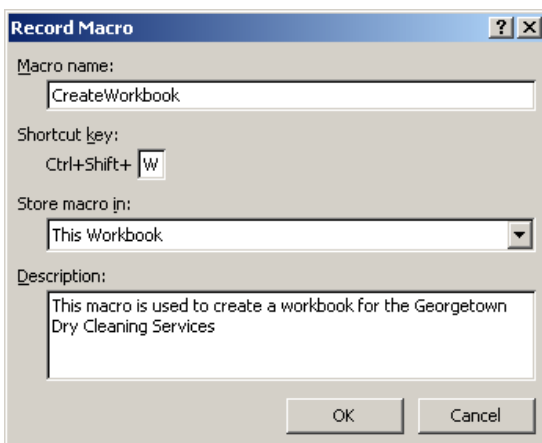
Introduction

A font is a description of characters designs to represent meaningful (or graphical) characters. A font is an object made of various characteristics, including a name, a size, and a style.

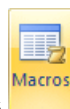
To define a font, the VBA library provides a class named Font. This class is equipped with the necessary characteristics.

❖ Practical Learning: Introducing Cell Formatting

1. Start Microsoft Excel
2. On the [Ribbon](#), click Developer
3. In the Code section, click Record Macro 
4. Set the Macro Name to **CreateWorkbook**
5. In the Shortcut Key text box, type **W** to get Ctrl + Shift + W



6. Click OK
7. On the Ribbon, click Stop Recording



8. In the Code section of the Ribbon, click Macros
9. In the Macro dialog box, make sure CreateWorkbook is selected and click Edit
10. Change the code as follows:

Option Explicit

```
Sub CreateWorkbook()
    ' CreateWorkbook Macro
    ' This macro is used to create a workbook for the
    ' Georgetown Dry Cleaning Services

    ' Keyboard Shortcut: Ctrl+Shift+W

    Rem Just in case there is anything on the
    Rem worksheet, delete everything
    Range("A:K").Delete
    Range("1:20").Delete

    Rem Create the sections and headings of the worksheet
    Range("B2") = "Georgetown Dry Cleaning Services"
    Range("B5") = "Order Identification"
    Range("B6") = "Receipt #:"
    Range("G6") = "Order Status:"
    Range("B7") = "Customer Name:"
    Range("G7") = "Customer Phone:"

    Range("B9") = "Date Left:"

```

```

Range("G9") = "Time Left:"
Range("B10") = "Date Expected:"
Range("G10") = "Time Expected:"
Range("B11") = "Date Picked Up:"
Range("G11") = "Time Picked Up:"

Range("B13") = "Items to Clean"
Range("B14") = "Item"
Range("D14") = "Unit Price"
Range("E14") = "Qty"
Range("F14") = "Sub-Total"


Range("B15") = "Shirts"
Range("H15") = "Order Summary"
Range("B16") = "Pants"
Range("B17") = "None"
Range("H17") = "Cleaning Total:"
Range("B18") = "None"
Range("H18") = "Tax Rate:"
Range("I18") = "5.75"
Range("J18") = "%"
Range("B19") = "None"
Range("H19") = "Tax Amount:"
Range("B20") = "None"
Range("H20") = "Order Total:"

Rem Change the widths and heights of some columns and rows
Rem In previous lessons, we learned all these things
Range("E:E, G:G").ColumnWidth = 4
Columns("H").ColumnWidth = 14
Columns("J").ColumnWidth = 1.75

Rows("3").RowHeight = 2
Range("8:8, 12:12").RowHeight = 8

Rem Hide the gridlines
ActiveWindow.DisplayGridlines = False
End Sub

```

11. To return to Microsoft Excel, click the View Microsoft Excel button 
12. To fill the worksheet, press Ctrl + Shift + W

	A	B	C	D	E	F	G	H	I	J
1										
2		Georgetown Dry Cleaning Services								
4										
5		Order Identification								
6		Receipt #:					Order Status:			
7		Customer Name:					Customer Phone:			
9		Date Left:					Time Left:			
10		Date Expected:					Time Expected:			
11		Date Picked Up:					Time Picked Up:			
13		Items to Clean								
14		Item	Unit Price	Qty	Sub-Total					
15		Shirts					Order Summary			
16		Pants								
17		None					Cleaning Total:			
18		None					Tax Rate:	5.75	%	
19		None					Tax Amount:			
20		None					Order Total:			

13. Close Microsoft Excel
14. When asked whether you want to save, click No

The Name of a Font

To programmatically specify the name of a font, refer to the cell or the group of cells on which you want to apply the font, access its **Font** object, followed by its **Name** property. Then assign the name of the font to the cell or group of cells.

❖ Practical Learning: Selecting a Font

1. Change the code as follows (if you do not have the Rockwell Condensed font, use Times New Roman):

```

Sub CreateWorkbook()
' CreateWorkbook Macro
' This macro is used to create a workbook for the
' Georgetown Dry Cleaning Services

```

```

' Keyboard Shortcut: Ctrl+Shift+W

Rem Just in case there is anything on the
Rem worksheet, delete everything
Range("A:K").Delete
Range("1:20").Delete

Rem Create the sections and headings of the worksheet
Range("B2") = "Georgetown Dry Cleaning Services"
Range("B2").Font.Name = "Rockwell Condensed"

Range("B5") = "Order Identification"
Range("B5").Font.Name = "Cambria"

Range("B6") = "Receipt #:"
Range("G6") = "Order Status:"
Range("B7") = "Customer Name:"
Range("G7") = "Customer Phone:"

Range("B9") = "Date Left:"
Range("G9") = "Time Left:"
Range("B10") = "Date Expected:"
Range("G10") = "Time Expected:"
Range("B11") = "Date Picked Up:"
Range("G11") = "Time Picked Up:"

Range("B13") = "Items to Clean"
Range("B13").Font.Name = "Cambria"

Range("B14") = "Item"
Range("D14") = "Unit Price"
Range("E14") = "Qty"
Range("F14") = "Sub-Total"

Range("B15") = "Shirts"

Range("H15") = "Order Summary"
Range("H15").Font.Name = "Cambria"

. . . No Change
End Sub

```

2. Return to Microsoft Excel and press Ctrl + Shift + W to see the result

	A	B	C	D	E	F	G	H	I	J
1										
2		Georgetown Dry Cleaning Services								
4		Order Identification								
6		Receipt #:				Order Status:				
7		Customer Name:				Customer Phone:				
9		Date Left:				Time Left:				
10		Date Expected:				Time Expected:				
11		Date Picked Up:				Time Picked Up:				
13		Items to Clean								
14		Item	Unit Price	Qty	Sub-Total	Order Summary				
15		Shirts								
16		Pants								
17		None				Cleaning Total:				
18		None				Tax Rate:				5.75 %
19		None				Tax Amount:				
20		None				Order Total:				

3. Return to Microsoft Visual Basic

The Size of a Font

Besides its name, a font is also known for its size. To programmatically specify the font size of a cell or a group of cells, refer to that cell or the group of cells, access its **Font** object, followed by its **Size** property, and assign the desired value to it.

❖ Practical Learning: Setting the Font Size of a Cell

1. Change the code as follows:

```

Sub CreateWorkbook()
' CreateWorkbook Macro
' This macro is used to create a workbook for the
' Georgetown Dry Cleaning Services

' Keyboard Shortcut: Ctrl+Shift+W

```

```

Rem Just in case there is anything on the
Rem worksheet, delete everything
Range("A:K").Delete
Range("1:20").Delete

Rem Create the sections and headings of the worksheet
Range("B2") = "Georgetown Dry Cleaning Services"
Range("B2").Font.Name = "Rockwell Condensed"
Range("B2").Font.Size = 24

Range("B5") = "Order Identification"
Range("B5").Font.Name = "Cambria"
Range("B5").Font.Size = 14

. . . No Change

Range("B13") = "Items to Clean"
Range("B13").Font.Name = "Cambria"
Range("B13").Font.Size = 14

Range("B14") = "Item"
Range("D14") = "Unit Price"
Range("E14") = "Qty"
Range("F14") = "Sub-Total"

Range("B15") = "Shirts"

Range("H15") = "Order Summary"
Range("H15").Font.Name = "Cambria"
Range("H15").Font.Size = 14

. . . No Change
End Sub

```

2. Return to Microsoft Excel and press Ctrl + Shift + W to see the result

	A	B	C	D	E	F	G	H	I	J
1										
2		Georgetown Dry Cleaning Services								
4		Order Identification								
6		Receipt #:				Order Status:				
7		Customer Name:				Customer Phone:				
9		Date Left:				Time Left:				
10		Date Expected:				Time Expected:				
11		Date Picked Up:				Time Picked Up:				
13		Items to Clean								
14		Item	Unit Price	Qty	Sub-Total					
15		Shirts				Order Summary				
16		Pants								
17		None				Cleaning Total:				
18		None				Tax Rate:		5.75 %		
19		None				Tax Amount:				
20		None				Order Total:				

3. Return to Microsoft Visual Basic

The Style of a Font

The style of a font is a technique of drawing the characters of the text

To support font styles, the **Font** object is equipped with various Boolean properties that are **Bold**, **Italic**, **Underline**, and **Strikethrough**. Therefore, to grammatically specify the font style of a cell or a group of cells, access the cell or the group of cells, access its **Font** object, followed by the desired style, and assign the desired Boolean value.

❖ Practical Learning: Formatting With Styles

1. Change the code as follows:

```

Sub CreateWorkbook()
' CreateWorkbook Macro
' This macro is used to create a workbook for the
' Georgetown Dry Cleaning Services

' Keyboard Shortcut: Ctrl+Shift+W

Rem Just in case there is anything on the
Rem worksheet, delete everything
Range("A:K").Delete

```

```

Range("1:20").Delete

Rem Create the sections and headings of the worksheet
Range("B2") = "Georgetown Dry Cleaning Services"
Range("B2").Font.Name = "Rockwell Condensed"
Range("B2").Font.Size = 24
Range("B2").Font.Bold = True

Range("B5") = "Order Identification"
Range("B5").Font.Name = "Cambria"
Range("B5").Font.Size = 14
Range("B5").Font.Bold = True

. . . No Change

Range("B13") = "Items to Clean"
Range("B13").Font.Name = "Cambria"
Range("B13").Font.Size = 14
Range("B13").Font.Bold = True

Range("B14") = "Item"
Range("D14") = "Unit Price"
Range("E14") = "Qty"
Range("F14") = "Sub-Total"

Range("B15") = "Shirts"

Range("H15") = "Order Summary"
Range("H15").Font.Name = "Cambria"
Range("H15").Font.Size = 14
Range("H15").Font.Bold = True

. . . No Change
End Sub

```

2. Return to Microsoft Excel and press Ctrl + Shift + W to see the result
3. Return to Microsoft Visual Basic

Text Color

A character or text can use a color to get a better visual representation. The VBA supports colors at different levels. To support colors, the **Font** object is equipped with a property named **Color**. To specify the color, assign the desired color to the property. The VBA provides a (limited) list of colors that each can be specified using a named constant. They are:

Color Name	Constant	Value	Color
Black	vbBlack	&h00	
Red	vbRed	&hFF	
Green	vbGreen	&hFF00	
Yellow	vbYellow	&hFFFF	
Blue	vbBlue	&hFF0000	
Magenta	vbMagenta	&hFF00FF	
Cyan	vbCyan	&hFFFF00	
White	vbWhite	&hFFFFFF	

Therefore, the available colors are **vbBlack**, **vbRed**, **vbGreen**, **vbYellow**, **vbBlue**, **vbMagenta**, **vbCyan**, and **vbWhite**. These are standard colors. In reality, a color in Microsoft Windows is represented as a value between 0 and 16,581,375 (**in the next lesson**, we will know where that number comes from). This means that you can assign a positive number to the **Font.Color** property and use the equivalent color.

The colors in the Font Color button are represented by a property named **ThemeColor**. Each one of the colors in the Theme Colors section has an equivalent name in the VBA. If you know the name of the color, assign it to the **ThemeColor** property.

As another alternative to specify a color, in the next lesson, we will see that you can use a function named **RGB** to specify a color.

❖ Practical Learning: Specifying the Color of Text

1. Change the code as follows:

```

Sub CreateWorkbook()
' CreateWorkbook Macro
' This macro is used to create a workbook for the
' Georgetown Dry Cleaning Services

' Keyboard Shortcut: Ctrl+Shift+W

Rem Just in case there is anything on the
Rem worksheet, delete everything
Range("A:K").Delete
Range("1:20").Delete

Rem Create the sections and headings of the worksheet
Range("B2") = "Georgetown Dry Cleaning Services"

```

```

Range("B2").Font.Name = "Rockwell Condensed"
Range("B2").Font.Size = 24
Range("B2").Font.Bold = True
Range("B2").Font.Color = vbBlue

Range("B3:J3").Interior.ThemeColor = xlThemeColorLight2

Range("B5") = "Order Identification"
Range("B5").Font.Name = "Cambria"
Range("B5").Font.Size = 14
Range("B5").Font.Bold = True
Range("B5").Font.ThemeColor = 5

. . . No Change

Range("B13") = "Items to Clean"
Range("B13").Font.Name = "Cambria"
Range("B13").Font.Size = 14
Range("B13").Font.Bold = True
Range("B13").Font.ThemeColor = 5

Range("B14") = "Item"
Range("D14") = "Unit Price"
Range("E14") = "Qty"
Range("F14") = "Sub-Total"

. . . No Change

Range("H15") = "Order Summary"
Range("H15").Font.Name = "Cambria"
Range("H15").Font.Size = 14
Range("H15").Font.Bold = True
Range("H15").Font.ThemeColor = 5

. . . No Change

```

```
End Sub
```

2. Return to Microsoft Excel and press Ctrl + Shift + W to see the result

	A	B	C	D	E	F	G	H	I	J
1										
2		Georgetown Dry Cleaning Services								
4		Order Identification								
6		Receipt #:				Order Status:				
7		Customer Name:				Customer Phone:				
9		Date Left:				Time Left:				
10		Date Expected:				Time Expected:				
11		Date Picked Up:				Time Picked Up:				
13		Items to Clean								
14		Item	Unit Price	Qty	Sub-Total			Order Summary		
15		Shirts						Order Summary		
16		Pants						Order Summary		
17		None						Cleaning Total:		
18		None						Tax Rate:	5.75 %	
19		None						Tax Amount:		
20		None						Order Total:		

Cell Alignment

Cells Merging

To programmatically merge some cells, first select them and access the **MergeCells** Boolean property. Then assign **True** or **False** depending on your intentions.

❖ Practical Learning: Merging Cells

- Change the code as follows:

```

Sub CreateWorkbook()
. . . No Change

Rem Merge the cells H15, I15, H16, and I16
Range("H15:I16").MergeCells = True

Rem Hide the gridlines

```

Cells Content Alignment

To programmatically align the text of a cell or a group of cells, access that cell or the group of cells, access either the **HorizontalAlignment** or the **VerticalAlignment** property, and assign the desired value to it.

❖ Practical Learning: Controlling Cells Alignment

- Change the code as follows:

```
Sub CreateWorkbook()
    . . . No Change

    Rem Merge the cells H15, I15, H16, and I16
    Range("H15:I16").MergeCells = True
    Rem Align the merged text to the left
    Range("H15:H16").VerticalAlignment = xlCenter

    Rem Hide the gridlines
    ActiveWindow.DisplayGridlines = False
End Sub
```

Cells Content Indentation

To programmatically indent the content of a cell or the contents of various cells, refer to that cell or to the group of cells and access its **IndentLevel** property. Then assign the desired value. Here is an example:

```
Range("A1").IndentLevel = 5
```

Cells Borders

The Line Style of a Border

A cell appears as a rectangular box with borders and a background. To programmatically control the borders of a cell or a group of cells, refer to the cell or the group of cells and access its **Borders** object. This object is accessed as an indexed property. Here is an example:

```
Range("B2").Borders()
```

In the parentheses of the **Borders** property, specify the border you want to change. The primary available values are: **xlEdgeBottom**, **xlEdgeTop**, **xlEdgeLeft**, and **xlEdgeRight**. Sometimes you may have selected a group of cells and you want to take an action on the line(s) between (among) them. To support this, the **Borders** property can take an index named **xlInsideVertical** for a vertical border between two cells or an index named **xlInsideHorizontal** for a horizontal border between the cells.

After specifying the border you want to work on, you must specify the type of characteristic you want to change. For example, you can specify the type of line you want the border to show. To support this, the **Borders** object is equipped with a property named **LineStyle**. To specify the type of line you want the border to display, you can assign a value to the **LineStyle** property. The available values are **xlContinuous**, **xlDash**, **xlDashDot**, **xlDashDotDot**, **xlDot**, **xlDouble**, **xlSlantDashDot**, and **xlLineStyleNone**. Therefore, you can assign any of these values to the property. To assist you with this, you can type **LineStyle** followed by a period and select the desired value from the list that appears:

```
Sub Exercise()
```

```
    Range("B2").Borders(xlEdgeLeft).LineStyle = XlLineStyle.
```

```
End Sub
```



❖ Practical Learning: Specifying the Styles of Cells Border

- Change the code as follows:

```
Sub CreateWorkbook()
    ' CreateWorkbook Macro
    ' This macro is used to create a workbook for the
    ' Georgetown Dry Cleaning Services
```

```

' Keyboard Shortcut: Ctrl+Shift+W

Rem Just in case there is anything on the
Rem worksheet, delete everything
Range("A:K").Delete
Range("1:20").Delete

Rem Create the sections and headings of the worksheet
Range("B2") = "Georgetown Dry Cleaning Services"
Range("B2").Font.Name = "Rockwell Condensed"
Range("B2").Font.Size = 24
Range("B2").Font.Bold = True
Range("B2").Font.Color = vbBlue

Range("B5") = "Order Identification"
Range("B5").Font.Name = "Cambria"
Range("B5").Font.Size = 14
Range("B5").Font.Bold = True
Range("B5").Font.ThemeColor = 5

Rem To draw a thick line, change the bottom
Rem borders of the cells from B5 to J5
Range("B5:J5").Borders(xlEdgeBottom).LineStyle = xlContinuous

Range("B6") = "Receipt #:"
Range("D6:F6").Borders(xlEdgeBottom).LineStyle = xlContinuous

Range("G6") = "Order Status:"
Range("I6:J6").Borders(xlEdgeBottom).LineStyle = xlContinuous

Range("B7") = "Customer Name:"
Range("D7:F7").Borders(xlEdgeBottom).LineStyle = xlContinuous

Range("G7") = "Customer Phone:"
Range("I7:J7").Borders(xlEdgeBottom).LineStyle = xlContinuous

Rem To draw a thick line, change the bottom
Rem borders of the cells from B5 to J5
Range("B8:J8").Borders(xlEdgeBottom).LineStyle = xlContinuous

Range("B9") = "Date Left:"
Range("D9:F9").Borders(xlEdgeBottom).LineStyle = xlContinuous

Range("G9") = "Time Left:"
Range("I9:J9").Borders(xlEdgeBottom).LineStyle = xlContinuous

Range("B10") = "Date Expected:"
Range("D10:F10").Borders(xlEdgeBottom).LineStyle = xlContinuous

Range("G10") = "Time Expected:"
Range("I10:J10").Borders(xlEdgeBottom).LineStyle = xlContinuous

Range("B11") = "Date Picked Up:"
Range("D11:F11").Borders(xlEdgeBottom).LineStyle = xlContinuous

Range("G11") = "Time Picked Up:"
Range("I11:J11").Borders(xlEdgeBottom).LineStyle = xlContinuous

Rem To draw a thick line, change the bottom
Rem borders of the cells from B5 to J5
Range("B12:J12").Borders(xlEdgeBottom).LineStyle = xlContinuous

Range("B13") = "Items to Clean"
Range("B13").Font.Name = "Cambria"
Range("B13").Font.Size = 14
Range("B13").Font.Bold = True

Range("B14") = "Item"
Range("D14") = "Unit Price"
Range("E14") = "Qty"
Range("F14") = "Sub-Total"

Range("B14:F14").Borders(xlEdgeLeft).LineStyle = xlContinuous
Range("B14:F14").Borders(xlEdgeTop).LineStyle = xlContinuous
Range("B14:F14").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("B14:F14").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("C14").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("D14").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E14").Borders(xlEdgeRight).LineStyle = xlContinuous

Range("B15") = "Shirts"
Range("B15").Borders(xlEdgeLeft).LineStyle = xlContinuous

Range("H15") = "Order Summary"
Range("H15").Font.Name = "Cambria"
Range("H15").Font.Size = 14
Range("H15").Font.Bold = True

Range("B16") = "Pants"
Range("B16").Borders(xlEdgeLeft).LineStyle = xlContinuous

Range("B17") = "None"

```



```

Range("B17").Borders(xlEdgeLeft).LineStyle = xlContinuous

Range("H17") = "Cleaning Total:"
Range("I17").Borders(xlEdgeBottom).LineStyle = xlContinuous

Range("B18") = "None"
Range("B18").Borders(xlEdgeLeft).LineStyle = xlContinuous

Range("H18") = "Tax Rate:"
Range("I18").Borders(xlEdgeBottom).LineStyle = xlContinuous

Range("I18") = "5.75"
Range("J18") = "%"
Range("B19") = "None"
Range("B19").Borders(xlEdgeLeft).LineStyle = xlContinuous

Range("H19") = "Tax Amount:"
Range("I19").Borders(xlEdgeBottom).LineStyle = xlContinuous

Range("B20") = "None"
Range("B20").Borders(xlEdgeLeft).LineStyle = xlContinuous

Range("C15").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C16").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C17").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C18").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C19").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C20").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("B14:C14").Borders(xlEdgeBottom).LineStyle = xlContinuous

Range("B15:C15").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D15:F15").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D15").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E15").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("F15").Borders(xlEdgeRight).LineStyle = xlContinuous

Range("B16:C16").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D16:F16").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D16").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E16").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("F16").Borders(xlEdgeRight).LineStyle = xlContinuous

Range("B17:C17").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D17:F17").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D17").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E17").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("F17").Borders(xlEdgeRight).LineStyle = xlContinuous

Range("B18:C18").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D18:F18").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D18").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E18").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("F18").Borders(xlEdgeRight).LineStyle = xlContinuous

Range("B19:C19").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D19:F19").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D19").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E19").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("F19").Borders(xlEdgeRight).LineStyle = xlContinuous

Range("B20:F20").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D20").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E20").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("F20").Borders(xlEdgeRight).LineStyle = xlContinuous

Range("H20") = "Order Total:"
Range("I20").Borders(xlEdgeBottom).LineStyle = xlContinuous

Rem Change the widths and heights of some columns and rows
Rem In previous lessons, we learned all these things
Range("E:E, G:G").ColumnWidth = 4
Columns("H").ColumnWidth = 14
Columns("J").ColumnWidth = 1.75

Rows("3").RowHeight = 2
Range("8:8, 12:12").RowHeight = 8

Rem Merge the cells H15, I15, H16, and I16
Range("H15:I16").MergeCells = True
Rem Align the merged text to the left
Range("H15:H16").VerticalAlignment = xlCenter

Range("H16").Borders(xlEdgeBottom).LineStyle = xlContinuous

Rem Hide the gridlines
ActiveWindow.DisplayGridlines = False
End Sub

```

The Weight of a Border

After specifying the type of line to apply to a border, you can control the thickness of the line. To support this, the **Borders** object is equipped with a property named **Weight**. The

❖ Practical Learning: Specifying the Weight of Cells Border

1. Change the code as follows:

```

Sub CreateWorkbook()
' CreateWorkbook Macro
' This macro is used to create a workbook for the
' Georgetown Dry Cleaning Services

' Keyboard Shortcut: Ctrl+Shift+W

Rem Just in case there is anything on the
Rem worksheet, delete everything
Range("A:K").Delete
Range("1:20").Delete

Rem Create the sections and headings of the worksheet
Range("B2") = "Georgetown Dry Cleaning Services"
Range("B2").Font.Name = "Rockwell Condensed"
Range("B2").Font.Size = 24
Range("B2").Font.Bold = True
Range("B2").Font.Color = vbBlue

Range("B5") = "Order Identification"
Range("B5").Font.Name = "Cambria"
Range("B5").Font.Size = 14
Range("B5").Font.Bold = True
Range("B5").Font.ThemeColor = 5

Rem To draw a thick line, change the bottom
Rem borders of the cells from B5 to J5
Range("B5:J5").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B5:J5").Borders(xlEdgeBottom).Weight = xlMedium

Range("B6") = "Receipt #:"
Range("D6:F6").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D6:F6").Borders(xlEdgeBottom).Weight = xlHairline

Range("G6") = "Order Status:"
Range("I6:J6").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I6:J6").Borders(xlEdgeBottom).Weight = xlHairline

Range("B7") = "Customer Name:"
Range("D7:F7").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D7:F7").Borders(xlEdgeBottom).Weight = xlHairline

Range("G7") = "Customer Phone:"
Range("I7:J7").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I7:J7").Borders(xlEdgeBottom).Weight = xlHairline

Rem To draw a thick line, change the bottom
Rem borders of the cells from B5 to J5
Range("B8:J8").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B8:J8").Borders(xlEdgeBottom).Weight = xlThin

Range("B9") = "Date Left:"
Range("D9:F9").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D9:F9").Borders(xlEdgeBottom).Weight = xlHairline

Range("G9") = "Time Left:"
Range("I9:J9").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I9:J9").Borders(xlEdgeBottom).Weight = xlHairline

Range("B10") = "Date Expected:"
Range("D10:F10").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D10:F10").Borders(xlEdgeBottom).Weight = xlHairline

Range("G10") = "Time Expected:"
Range("I10:J10").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I10:J10").Borders(xlEdgeBottom).Weight = xlHairline

Range("B11") = "Date Picked Up:"
Range("D11:F11").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D11:F11").Borders(xlEdgeBottom).Weight = xlHairline

Range("G11") = "Time Picked Up:"
Range("I11:J11").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I11:J11").Borders(xlEdgeBottom).Weight = xlHairline

Rem To draw a thick line, change the bottom
Rem borders of the cells from B5 to J5
Range("B12:J12").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B12:J12").Borders(xlEdgeBottom).Weight = xlMedium

Range("B13") = "Items to Clean"
Range("B13").Font.Name = "Cambria"
Range("B13").Font.Size = 14
Range("B13").Font.Bold = True

Range("B14") = "Item"

```

```
Range("D14") = "Unit Price"
Range("E14") = "Qty"
Range("F14") = "Sub-Total"
```

```
Range("B14:F14").Borders(xlEdgeLeft).LineStyle = xlContinuous
Range("B14:F14").Borders(xlEdgeLeft).Weight = xlThin
Range("B14:F14").Borders(xlEdgeTop).LineStyle = xlContinuous
Range("B14:F14").Borders(xlEdgeTop).Weight = xlThin
Range("B14:F14").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("B14:F14").Borders(xlEdgeRight).Weight = xlThin
Range("B14:F14").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B14:F14").Borders(xlEdgeBottom).Weight = xlThin
Range("C14").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C14").Borders(xlEdgeRight).Weight = xlThin
Range("D14").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("D14").Borders(xlEdgeRight).Weight = xlThin
Range("E14").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E14").Borders(xlEdgeRight).Weight = xlThin
```

```
Range("B15") = "Shirts"
Range("B15").Borders(xlEdgeLeft).LineStyle = xlContinuous
Range("B15").Borders(xlEdgeLeft).Weight = xlThin
```

```
Range("H15") = "Order Summary"
Range("H15").Font.Name = "Cambria"
Range("H15").Font.Size = 14
Range("H15").Font.Bold = True
```

```
Range("B16") = "Pants"
Range("B16").Borders(xlEdgeLeft).LineStyle = xlContinuous
Range("B16").Borders(xlEdgeLeft).Weight = xlThin
```

```
Range("B17") = "None"
Range("B17").Borders(xlEdgeLeft).LineStyle = xlContinuous
Range("B17").Borders(xlEdgeLeft).Weight = xlThin
```

```
Range("H17") = "Cleaning Total:"
Range("I17").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I17").Borders(xlEdgeBottom).Weight = xlHairline
```

```
Range("B18") = "None"
Range("B18").Borders(xlEdgeLeft).LineStyle = xlContinuous
Range("B18").Borders(xlEdgeLeft).Weight = xlThin
```

```
Range("H18") = "Tax Rate:"
Range("I18").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I18").Borders(xlEdgeBottom).Weight = xlHairline
```

```
Range("I18") = "5.75"
Range("J18") = "%"
Range("B19") = "None"
Range("B19").Borders(xlEdgeLeft).LineStyle = xlContinuous
Range("B19").Borders(xlEdgeLeft).Weight = xlThin
```

```
Range("H19") = "Tax Amount:"
Range("I19").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I19").Borders(xlEdgeBottom).Weight = xlHairline
```

```
Range("B20") = "None"
Range("B20").Borders(xlEdgeLeft).LineStyle = xlContinuous
Range("B20").Borders(xlEdgeLeft).Weight = xlThin
```

```
Range("C15").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C15").Borders(xlEdgeRight).Weight = xlThin
Range("C16").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C16").Borders(xlEdgeRight).Weight = xlThin
Range("C17").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C17").Borders(xlEdgeRight).Weight = xlThin
Range("C18").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C18").Borders(xlEdgeRight).Weight = xlThin
Range("C19").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C19").Borders(xlEdgeRight).Weight = xlThin
Range("C20").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C20").Borders(xlEdgeRight).Weight = xlThin
Range("B14:C14").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B14:C14").Borders(xlEdgeBottom).Weight = xlThin
```

```
Range("B15:C15").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B15:C15").Borders(xlEdgeBottom).Weight = xlThin
Range("D15:F15").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D15:F15").Borders(xlEdgeBottom).Weight = xlHairline
Range("D15").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("D15").Borders(xlEdgeRight).Weight = xlHairline
Range("E15").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E15").Borders(xlEdgeRight).Weight = xlHairline
Range("F15").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("F15").Borders(xlEdgeRight).Weight = xlThin
```

```
Range("B16:C16").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B16:C16").Borders(xlEdgeBottom).Weight = xlThin
Range("D16:F16").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D16:F16").Borders(xlEdgeBottom).Weight = xlHairline
Range("D16").Borders(xlEdgeRight).LineStyle = xlContinuous
```

```

Range("D16").Borders(xlEdgeRight).Weight = xlHairline
Range("E16").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E16").Borders(xlEdgeRight).Weight = xlHairline
Range("F16").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("F16").Borders(xlEdgeRight).Weight = xlThin

Range("B17:C17").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B17:C17").Borders(xlEdgeBottom).Weight = xlThin
Range("D17:F17").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D17:F17").Borders(xlEdgeBottom).Weight = xlHairline
Range("D17").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("D17").Borders(xlEdgeRight).Weight = xlHairline
Range("E17").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E17").Borders(xlEdgeRight).Weight = xlHairline
Range("F17").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("F17").Borders(xlEdgeRight).Weight = xlThin

Range("B18:C18").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B18:C18").Borders(xlEdgeBottom).Weight = xlThin
Range("D18:F18").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D18:F18").Borders(xlEdgeBottom).Weight = xlHairline
Range("D18").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("D18").Borders(xlEdgeRight).Weight = xlHairline
Range("E18").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E18").Borders(xlEdgeRight).Weight = xlHairline
Range("F18").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("F18").Borders(xlEdgeRight).Weight = xlThin

Range("B19:C19").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B19:C19").Borders(xlEdgeBottom).Weight = xlThin
Range("D19:F19").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D19:F19").Borders(xlEdgeBottom).Weight = xlHairline
Range("D19").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("D19").Borders(xlEdgeRight).Weight = xlHairline
Range("E19").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E19").Borders(xlEdgeRight).Weight = xlHairline
Range("F19").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("F19").Borders(xlEdgeRight).Weight = xlThin

Range("B20:F20").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B20:F20").Borders(xlEdgeBottom).Weight = xlThin
Range("D20").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("D20").Borders(xlEdgeRight).Weight = xlHairline
Range("E20").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E20").Borders(xlEdgeRight).Weight = xlHairline
Range("F20").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("F20").Borders(xlEdgeRight).Weight = xlThin

Range("H20") = "Order Total:"
Range("I20").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I20").Borders(xlEdgeBottom).Weight = xlHairline

Rem Change the widths and heights of some columns and rows
Rem In previous lessons, we learned all these things
Range("E:E, G:G").ColumnWidth = 4
Columns("H").ColumnWidth = 14
Columns("J").ColumnWidth = 1.75

Rows("3").RowHeight = 2
Range("8:8, 12:12").RowHeight = 8

Rem Merge the cells H15, I15, H16, and I16
Range("H15:I16").MergeCells = True
Rem Align the merged text to the left
Range("H15:H16").VerticalAlignment = xlCenter

Range("H16").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("H16:I16").Borders(xlEdgeBottom).Weight = xlMedium

Rem Hide the gridlines
ActiveWindow.DisplayGridlines = False
End Sub

```

2. Return to Microsoft Excel and press Ctrl + Shift + W to see the result

	A	B	C	D	E	F	G	H	I	J
1										
2	Georgetown Dry Cleaning Services									
4										
5	Order Identification									
6	Receipt #:					Order Status:				
7	Customer Name:					Customer Phone:				
8										
9	Date Left:					Time Left:				
10	Date Expected:					Time Expected:				
11	Date Picked Up:					Time Picked Up:				
12										
13	Items to Clean									
14	Item		Unit Price	Qty	Sub-Total					
15	Shirts									
16	Pants									
17	None									
18	None									
19	None									
20	None									
	Order Summary									
	Cleaning Total:									
	Tax Rate: 5.75 %									
	Tax Amount:									
	Order Total:									

3. Return to Microsoft Visual Basic

The Color of a Border

To programmatically specify the color of a border, access the **Borders** indexed property of a cell or a group of cells and specify the border whose color you want to change, as we saw in the previous section. To support colors, the **Borders** object is equipped with a property named **Color**. To specify the color, assign the desired color to the property. The VBA provides a (limited) list of colors such as **vbBlack**, **vbWhite**, **vbRed**, **vbGreen**, and **vbBlue**. In reality, a color in Microsoft Windows is represented as a color between 0 and 16,581,375.

❖ Practical Learning: Controlling the Colors of Cells Borders

1. Change the code as follows:

```
Sub CreateWorkbook()
    ' CreateWorkbook Macro
    ' This macro is used to create a workbook for the
    ' Georgetown Dry Cleaning Services

    ' Keyboard Shortcut: Ctrl+Shift+W

    . . . No Change

    Rem To draw a thick line, change the bottom
    Rem borders of the cells from B5 to J5
    Range("B5:J5").Borders(xlEdgeBottom).LineStyle = xlContinuous
    Range("B5:J5").Borders(xlEdgeBottom).Weight = xlMedium
    Range("B5:J5").Borders(xlEdgeBottom).ThemeColor = 5

    . . . No Change

    Rem To draw a thick line, change the bottom
    Rem borders of the cells from B5 to J5
    Range("B12:J12").Borders(xlEdgeBottom).LineStyle = xlContinuous
    Range("B12:J12").Borders(xlEdgeBottom).Weight = xlMedium
    Range("B12:J12").Borders(xlEdgeBottom).ThemeColor = 5

    . . . No Change

    Range("H16").Borders(xlEdgeBottom).LineStyle = xlContinuous
    Range("H16:I16").Borders(xlEdgeBottom).Weight = xlMedium
    Range("H16:I16").Borders(xlEdgeBottom).ThemeColor = 5

    Rem Hide the gridlines
    ActiveWindow.DisplayGridlines = False
End Sub
```

2. Return to Microsoft Excel and press Ctrl + Shift + W to see the result

3. Return to Microsoft Visual Basic

The Cell's Background

A cell has a background color which, by default, is white. If you want to change a background, specify the cell or group of cells, using the Range class. The **Range** class is equipped with a property named **Interior**. From this property, you can access the **ThemeColor** and assign the desired color.

❖ Practical Learning: Painting the Background of Cells

1. Change the code as follows:

```

Sub CreateWorkbook()
' CreateWorkbook Macro
' This macro is used to create a workbook for the
' Georgetown Dry Cleaning Services

' Keyboard Shortcut: Ctrl+Shift+W

Rem Just in case there is anything on the
Rem worksheet, delete everything
Range("A:K").Delete
Range("1:20").Delete

Rem Create the sections and headings of the worksheet
Range("B2") = "Georgetown Dry Cleaning Services"
Range("B2").Font.Name = "Rockwell Condensed"
Range("B2").Font.Size = 24
Range("B2").Font.Bold = True
Range("B2").Font.Color = vbBlue

Range("B3:J3").Interior.ThemeColor = xlThemeColorLight2

Range("B5") = "Order Identification"
Range("B5").Font.Name = "Cambria"
Range("B5").Font.Size = 14
Range("B5").Font.Bold = True
Range("B5").Font.ThemeColor = 5

Rem To draw a thick line, change the bottom
Rem borders of the cells from B5 to J5
Range("B5:J5").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B5:J5").Borders(xlEdgeBottom).Weight = xlMedium
Range("B5:J5").Borders(xlEdgeBottom).ThemeColor = 5

Range("B6") = "Receipt #:"
Range("D6:F6").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D6:F6").Borders(xlEdgeBottom).Weight = xlHairline

Range("G6") = "Order Status:"
Range("I6:J6").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I6:J6").Borders(xlEdgeBottom).Weight = xlHairline

Range("B7") = "Customer Name:"
Range("D7:F7").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D7:F7").Borders(xlEdgeBottom).Weight = xlHairline

Range("G7") = "Customer Phone:"
Range("I7:J7").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I7:J7").Borders(xlEdgeBottom).Weight = xlHairline

Rem To draw a thick line, change the bottom
Rem borders of the cells from B5 to J5
Range("B8:J8").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B8:J8").Borders(xlEdgeBottom).Weight = xlThin

Range("B9") = "Date Left:"
Range("D9:F9").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D9:F9").Borders(xlEdgeBottom).Weight = xlHairline

Range("G9") = "Time Left:"
Range("I9:J9").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I9:J9").Borders(xlEdgeBottom).Weight = xlHairline

Range("B10") = "Date Expected:"
Range("D10:F10").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D10:F10").Borders(xlEdgeBottom).Weight = xlHairline

Range("G10") = "Time Expected:"
Range("I10:J10").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I10:J10").Borders(xlEdgeBottom).Weight = xlHairline

Range("B11") = "Date Picked Up:"
Range("D11:F11").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D11:F11").Borders(xlEdgeBottom).Weight = xlHairline

Range("G11") = "Time Picked Up:"
Range("I11:J11").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I11:J11").Borders(xlEdgeBottom).Weight = xlHairline

Rem To draw a thick line, change the bottom
Rem borders of the cells from B5 to J5
Range("B12:J12").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B12:J12").Borders(xlEdgeBottom).Weight = xlMedium
Range("B12:J12").Borders(xlEdgeBottom).ThemeColor = 5

Range("B13") = "Items to Clean"
Range("B13").Font.Name = "Cambria"
Range("B13").Font.Size = 14
Range("B13").Font.Bold = True
Range("B13").Font.ThemeColor = 5

```

```

Range("B14") = "Item"
Range("D14") = "Unit Price"
Range("E14") = "Qty"
Range("F14") = "Sub-Total"

Range("B14:F14").Borders(xlEdgeLeft).LineStyle = xlContinuous
Range("B14:F14").Borders(xlEdgeLeft).Weight = xlThin
Range("B14:F14").Borders(xlEdgeTop).LineStyle = xlContinuous
Range("B14:F14").Borders(xlEdgeTop).Weight = xlThin
Range("B14:F14").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("B14:F14").Borders(xlEdgeRight).Weight = xlThin
Range("B14:F14").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B14:F14").Borders(xlEdgeBottom).Weight = xlThin
Range("C14").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C14").Borders(xlEdgeRight).Weight = xlThin
Range("D14").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("D14").Borders(xlEdgeRight).Weight = xlThin
Range("E14").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E14").Borders(xlEdgeRight).Weight = xlThin

Range("B15") = "Shirts"
Range("B15").Borders(xlEdgeLeft).LineStyle = xlContinuous
Range("B15").Borders(xlEdgeLeft).Weight = xlThin

Range("H15") = "Order Summary"
Range("H15").Font.Name = "Cambria"
Range("H15").Font.Size = 14
Range("H15").Font.Bold = True
Range("H15").Font.ThemeColor = 5

Range("B16") = "Pants"
Range("B16").Borders(xlEdgeLeft).LineStyle = xlContinuous
Range("B16").Borders(xlEdgeLeft).Weight = xlThin

Range("B17") = "None"
Range("B17").Borders(xlEdgeLeft).LineStyle = xlContinuous
Range("B17").Borders(xlEdgeLeft).Weight = xlThin

Range("H17") = "Cleaning Total:"
Range("I17").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I17").Borders(xlEdgeBottom).Weight = xlHairline

Range("B18") = "None"
Range("B18").Borders(xlEdgeLeft).LineStyle = xlContinuous
Range("B18").Borders(xlEdgeLeft).Weight = xlThin

Range("H18") = "Tax Rate:"
Range("I18").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I18").Borders(xlEdgeBottom).Weight = xlHairline

Range("I18") = "5.75"
Range("J18") = "%"
Range("B19") = "None"
Range("B19").Borders(xlEdgeLeft).LineStyle = xlContinuous
Range("B19").Borders(xlEdgeLeft).Weight = xlThin

Range("H19") = "Tax Amount:"
Range("I19").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I19").Borders(xlEdgeBottom).Weight = xlHairline

Range("B20") = "None"
Range("B20").Borders(xlEdgeLeft).LineStyle = xlContinuous
Range("B20").Borders(xlEdgeLeft).Weight = xlThin

Range("C15").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C15").Borders(xlEdgeRight).Weight = xlThin
Range("C16").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C16").Borders(xlEdgeRight).Weight = xlThin
Range("C17").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C17").Borders(xlEdgeRight).Weight = xlThin
Range("C18").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C18").Borders(xlEdgeRight).Weight = xlThin
Range("C19").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C19").Borders(xlEdgeRight).Weight = xlThin
Range("C20").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C20").Borders(xlEdgeRight).Weight = xlThin
Range("B14:C14").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B14:C14").Borders(xlEdgeBottom).Weight = xlThin

Range("B15:C15").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B15:C15").Borders(xlEdgeBottom).Weight = xlThin
Range("D15:F15").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D15:F15").Borders(xlEdgeBottom).Weight = xlHairline
Range("D15").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("D15").Borders(xlEdgeRight).Weight = xlHairline
Range("E15").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E15").Borders(xlEdgeRight).Weight = xlHairline
Range("F15").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("F15").Borders(xlEdgeRight).Weight = xlThin

Range("B16:C16").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B16:C16").Borders(xlEdgeBottom).Weight = xlThin
Range("D16:F16").Borders(xlEdgeBottom).LineStyle = xlContinuous

```

```

Range("D16:F16").Borders(xlEdgeBottom).Weight = xlHairline
Range("D16").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("D16").Borders(xlEdgeRight).Weight = xlHairline
Range("E16").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E16").Borders(xlEdgeRight).Weight = xlHairline
Range("F16").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("F16").Borders(xlEdgeRight).Weight = xlThin

Range("B17:C17").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B17:C17").Borders(xlEdgeBottom).Weight = xlThin
Range("D17:F17").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D17:F17").Borders(xlEdgeBottom).Weight = xlHairline
Range("D17").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("D17").Borders(xlEdgeRight).Weight = xlHairline
Range("E17").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E17").Borders(xlEdgeRight).Weight = xlHairline
Range("F17").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("F17").Borders(xlEdgeRight).Weight = xlThin

Range("B18:C18").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B18:C18").Borders(xlEdgeBottom).Weight = xlThin
Range("D18:F18").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D18:F18").Borders(xlEdgeBottom).Weight = xlHairline
Range("D18").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("D18").Borders(xlEdgeRight).Weight = xlHairline
Range("E18").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E18").Borders(xlEdgeRight).Weight = xlHairline
Range("F18").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("F18").Borders(xlEdgeRight).Weight = xlThin

Range("B19:C19").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B19:C19").Borders(xlEdgeBottom).Weight = xlThin
Range("D19:F19").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D19:F19").Borders(xlEdgeBottom).Weight = xlHairline
Range("D19").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("D19").Borders(xlEdgeRight).Weight = xlHairline
Range("E19").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E19").Borders(xlEdgeRight).Weight = xlHairline
Range("F19").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("F19").Borders(xlEdgeRight).Weight = xlThin

Range("B20:F20").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B20:F20").Borders(xlEdgeBottom).Weight = xlThin
Range("D20").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("D20").Borders(xlEdgeRight).Weight = xlHairline
Range("E20").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E20").Borders(xlEdgeRight).Weight = xlHairline
Range("F20").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("F20").Borders(xlEdgeRight).Weight = xlThin

Range("H20") = "Order Total:"
Range("I20").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I20").Borders(xlEdgeBottom).Weight = xlHairline

Rem Change the widths and heights of some columns and rows
Rem In previous lessons, we learned all these things
Range("E:E, G:G").ColumnWidth = 4
Columns("H").ColumnWidth = 14
Columns("J").ColumnWidth = 1.75

Rows("3").RowHeight = 2
Range("8:8, 12:12").RowHeight = 8

Rem Merge the cells H15, I15, H16, and I16
Range("H15:I16").MergeCells = True
Rem Align the merged text to the left
Range("H15:H16").VerticalAlignment = xlBottom

Range("H16").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("H16:I16").Borders(xlEdgeBottom).Weight = xlMedium
Range("H16:I16").Borders(xlEdgeBottom).ThemeColor = 5

Rem Hide the gridlines
ActiveWindow.DisplayGridlines = False
End Sub

```

2. Return to Microsoft Excel and press Ctrl + Shift + W to see the result

	A	B	C	D	E	F	G	H	I	J																												
1																																						
2	Georgetown Dry Cleaning Services																																					
4																																						
5	Order Identification																																					
6	Receipt #:					Order Status:																																
7	Customer Name:					Customer Phone:																																
8																																						
9	Date Left:					Time Left:																																
10	Date Expected:					Time Expected:																																
11	Date Picked Up:					Time Picked Up:																																
12																																						
13	Items to Clean																																					
14	<table border="1"> <thead> <tr> <th>Item</th> <th>Unit Price</th> <th>Qty</th> <th>Sub-Total</th> </tr> </thead> <tbody> <tr> <td>Shirts</td> <td></td> <td></td> <td></td> </tr> <tr> <td>Pants</td> <td></td> <td></td> <td></td> </tr> <tr> <td>None</td> <td></td> <td></td> <td></td> </tr> <tr> <td>None</td> <td></td> <td></td> <td></td> </tr> <tr> <td>None</td> <td></td> <td></td> <td></td> </tr> <tr> <td>None</td> <td></td> <td></td> <td></td> </tr> </tbody> </table>										Item	Unit Price	Qty	Sub-Total	Shirts				Pants				None				None				None				None			
Item	Unit Price	Qty	Sub-Total																																			
Shirts																																						
Pants																																						
None																																						
None																																						
None																																						
None																																						
15																																						
16																																						
17																																						
18																																						
19																																						
20																																						
21																																						

Order Summary

Cleaning Total:

Tax Rate: 5.75 %

Tax Amount:

Order Total:

3. Close the worksheet
4. When asked whether you want to save, click No



Introduction to Built-In Functions

Constants, Expressions and Formulas

Introduction to Constants

A constant is a value that does not change. It can be a number, a string, or an expression. To create a constant, use the **Const** keyword and assign the desired value to it. Here is an example:

```
Private Sub CreateConstant()  
    Const Number6 = 6  
End Sub
```

After creating the constant, you can use its name wherever its value would have been used. Some of the constants you will use in your expressions have already been created. We will mention them when necessary.

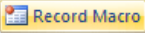
Introduction to Expressions

An expression is one or more symbols combined with one or more values to create another value. For example, +16 is an expression that creates the positive value 16. Most expressions that we know are made of arithmetic calculations. An example is $422.82 * 15.55$.

To add an expression to a selected cell, assign it to the **ActiveCell** object. Here is an example:

```
Sub Exercise()  
    ActiveCell = 422.82 * 15.5  
End Sub
```

❖ Practical Learning: Introducing Expressions

1. Start Microsoft Excel
2. On the [Ribbon](#), click Developer
3. In the Code section, click Record Macro 
4. Set the Macro Name to **CreateWorkbook**
5. In the Shortcut Key text box, type W to get Ctrl + Shift + W and click OK
6. On the Ribbon, click Stop Recording



7. In the Code section of the Ribbon, click Macros
8. In the Macro dialog box, make sure CreateWorkbook is selected and click Edit
9. Change the code as follows:

```
Sub CreateWorkbook()  
    '  
    ' CreateWorkbook Macro  
    ' This macro is used to create a workbook for the  
    ' Georgetown Dry Cleaning Services  
    '  
    ' Keyboard Shortcut: Ctrl+Shift+W  
    '  
    Rem Just in case there is anything on the  
    Rem worksheet, delete everything  
    Range("A:K").Delete  
    Range("1:20").Delete  
    '  
    Rem Create the sections and headings of the worksheet  
    Range("B2") = "Georgetown Dry Cleaning Services"  
    Range("B2").Font.Name = "Rockwell Condensed"  
    Range("B2").Font.Size = 24  
    Range("B2").Font.Bold = True  
    Range("B2").Font.Color = vbBlue  
    '  
    Range("B3:J3").Interior.ThemeColor = xlThemeColorLight2  
    '  
    Range("B5") = "Order Identification"  
    Range("B5").Font.Name = "Cambria"
```

```

Range("B5").Font.Size = 14
Range("B5").Font.Bold = True
Range("B5").Font.ThemeColor = 5

Rem To draw a thick line, change the bottom
Rem borders of the cells from B5 to J5
Range("B5:J5").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B5:J5").Borders(xlEdgeBottom).Weight = xlMedium
Range("B5:J5").Borders(xlEdgeBottom).ThemeColor = 5

Range("B6") = "Receipt #:"
Range("D6:F6").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D6:F6").Borders(xlEdgeBottom).Weight = xlHairline

Range("G6") = "Order Status:"
Range("I6:J6").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I6:J6").Borders(xlEdgeBottom).Weight = xlHairline

Range("B7") = "Customer Name:"
Range("D7:F7").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D7:F7").Borders(xlEdgeBottom).Weight = xlHairline

Range("G7") = "Customer Phone:"
Range("I7:J7").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I7:J7").Borders(xlEdgeBottom).Weight = xlHairline

Rem To draw a thick line, change the bottom
Rem borders of the cells from B5 to J5
Range("B8:J8").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B8:J8").Borders(xlEdgeBottom).Weight = xlThin

Range("B9") = "Date Left:"
Range("D9:F9").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D9:F9").Borders(xlEdgeBottom).Weight = xlHairline

Range("G9") = "Time Left:"
Range("I9:J9").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I9:J9").Borders(xlEdgeBottom).Weight = xlHairline

Range("B10") = "Date Expected:"
Range("D10:F10").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D10:F10").Borders(xlEdgeBottom).Weight = xlHairline

Range("G10") = "Time Expected:"
Range("I10:J10").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I10:J10").Borders(xlEdgeBottom).Weight = xlHairline

Range("B11") = "Date Picked Up:"
Range("D11:F11").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D11:F11").Borders(xlEdgeBottom).Weight = xlHairline

Range("G11") = "Time Picked Up:"
Range("I11:J11").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I11:J11").Borders(xlEdgeBottom).Weight = xlHairline

Rem To draw a thick line, change the bottom
Rem borders of the cells from B5 to J5
Range("B12:J12").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B12:J12").Borders(xlEdgeBottom).Weight = xlMedium
Range("B12:J12").Borders(xlEdgeBottom).ThemeColor = 5

Range("B13") = "Items to Clean"
Range("B13").Font.Name = "Cambria"
Range("B13").Font.Size = 14
Range("B13").Font.Bold = True
Range("B13").Font.ThemeColor = 5

Range("B14") = "Item"
Range("D14") = "Unit Price"
Range("E14") = "Qty"
Range("F14") = "Sub-Total"

Range("B14:F14").Borders(xlEdgeLeft).LineStyle = xlContinuous
Range("B14:F14").Borders(xlEdgeLeft).Weight = xlThin
Range("B14:F14").Borders(xlEdgeTop).LineStyle = xlContinuous
Range("B14:F14").Borders(xlEdgeTop).Weight = xlThin
Range("B14:F14").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("B14:F14").Borders(xlEdgeRight).Weight = xlThin
Range("B14:F14").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B14:F14").Borders(xlEdgeBottom).Weight = xlThin
Range("C14").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C14").Borders(xlEdgeRight).Weight = xlThin
Range("D14").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("D14").Borders(xlEdgeRight).Weight = xlThin
Range("E14").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E14").Borders(xlEdgeRight).Weight = xlThin

Range("B15") = "Shirts"
Range("B15").Borders(xlEdgeLeft).LineStyle = xlContinuous
Range("B15").Borders(xlEdgeLeft).Weight = xlThin

Range("H15") = "Order Summary"
Range("H15").Font.Name = "Cambria"
Range("H15").Font.Size = 14
Range("H15").Font.Bold = True
Range("H15").Font.ThemeColor = 5

```

```

Range("B16") = "Pants"
Range("B16").Borders(xlEdgeLeft).LineStyle = xlContinuous
Range("B16").Borders(xlEdgeLeft).Weight = xlThin

Range("B17") = "None"
Range("B17").Borders(xlEdgeLeft).LineStyle = xlContinuous
Range("B17").Borders(xlEdgeLeft).Weight = xlThin

Range("H17") = "Cleaning Total:"
Range("I17").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I17").Borders(xlEdgeBottom).Weight = xlHairline

Range("B18") = "None"
Range("B18").Borders(xlEdgeLeft).LineStyle = xlContinuous
Range("B18").Borders(xlEdgeLeft).Weight = xlThin

Range("H18") = "Tax Rate:"
Range("I18").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I18").Borders(xlEdgeBottom).Weight = xlHairline

Range("I18") = "5.75"
Range("J18") = "%"
Range("B19") = "None"
Range("B19").Borders(xlEdgeLeft).LineStyle = xlContinuous
Range("B19").Borders(xlEdgeLeft).Weight = xlThin

Range("H19") = "Tax Amount:"
Range("I19").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I19").Borders(xlEdgeBottom).Weight = xlHairline

Range("B20") = "None"
Range("B20").Borders(xlEdgeLeft).LineStyle = xlContinuous
Range("B20").Borders(xlEdgeLeft).Weight = xlThin

Range("C15").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C15").Borders(xlEdgeRight).Weight = xlThin
Range("C16").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C16").Borders(xlEdgeRight).Weight = xlThin
Range("C17").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C17").Borders(xlEdgeRight).Weight = xlThin
Range("C18").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C18").Borders(xlEdgeRight).Weight = xlThin
Range("C19").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C19").Borders(xlEdgeRight).Weight = xlThin
Range("C20").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("C20").Borders(xlEdgeRight).Weight = xlThin
Range("B14:C14").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B14:C14").Borders(xlEdgeBottom).Weight = xlThin

Range("B15:C15").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B15:C15").Borders(xlEdgeBottom).Weight = xlThin
Range("D15:F15").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D15:F15").Borders(xlEdgeBottom).Weight = xlHairline
Range("D15").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("D15").Borders(xlEdgeRight).Weight = xlHairline
Range("E15").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E15").Borders(xlEdgeRight).Weight = xlHairline
Range("F15").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("F15").Borders(xlEdgeRight).Weight = xlThin

Range("B16:C16").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B16:C16").Borders(xlEdgeBottom).Weight = xlThin
Range("D16:F16").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D16:F16").Borders(xlEdgeBottom).Weight = xlHairline
Range("D16").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("D16").Borders(xlEdgeRight).Weight = xlHairline
Range("E16").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E16").Borders(xlEdgeRight).Weight = xlHairline
Range("F16").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("F16").Borders(xlEdgeRight).Weight = xlThin

Range("B17:C17").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B17:C17").Borders(xlEdgeBottom).Weight = xlThin
Range("D17:F17").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D17:F17").Borders(xlEdgeBottom).Weight = xlHairline
Range("D17").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("D17").Borders(xlEdgeRight).Weight = xlHairline
Range("E17").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E17").Borders(xlEdgeRight).Weight = xlHairline
Range("F17").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("F17").Borders(xlEdgeRight).Weight = xlThin

Range("B18:C18").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B18:C18").Borders(xlEdgeBottom).Weight = xlThin
Range("D18:F18").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D18:F18").Borders(xlEdgeBottom).Weight = xlHairline
Range("D18").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("D18").Borders(xlEdgeRight).Weight = xlHairline
Range("E18").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E18").Borders(xlEdgeRight).Weight = xlHairline
Range("F18").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("F18").Borders(xlEdgeRight).Weight = xlThin

Range("B19:C19").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B19:C19").Borders(xlEdgeBottom).Weight = xlThin
Range("D19:F19").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("D19:F19").Borders(xlEdgeBottom).Weight = xlHairline

```

```

Range("D19").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("D19").Borders(xlEdgeRight).Weight = xlHairline
Range("E19").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E19").Borders(xlEdgeRight).Weight = xlHairline
Range("F19").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("F19").Borders(xlEdgeRight).Weight = xlThin

Range("B20:F20").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("B20:F20").Borders(xlEdgeBottom).Weight = xlThin
Range("D20").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("D20").Borders(xlEdgeRight).Weight = xlHairline
Range("E20").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("E20").Borders(xlEdgeRight).Weight = xlHairline
Range("F20").Borders(xlEdgeRight).LineStyle = xlContinuous
Range("F20").Borders(xlEdgeRight).Weight = xlThin

Range("H20") = "Order Total:"
Range("I20").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("I20").Borders(xlEdgeBottom).Weight = xlHairline

Rem Change the widths and heights of some columns and rows
Rem In previous lessons, we learned all these things
Range("E:E, G:G").ColumnWidth = 4
Columns("H").ColumnWidth = 14
Columns("J").ColumnWidth = 1.75

Rows("3").RowHeight = 2
Range("8:8, 12:12").RowHeight = 8

Rem Merge the cells H15, I15, H16, and I16
Range("H15:I16").MergeCells = True
Rem Align the merged text to the left
Range("H15:H16").VerticalAlignment = xlBottom

Range("H16").Borders(xlEdgeBottom).LineStyle = xlContinuous
Range("H16:I16").Borders(xlEdgeBottom).Weight = xlMedium
Range("H16:I16").Borders(xlEdgeBottom).ThemeColor = 5

Rem Hide the gridlines
ActiveWindow.DisplayGridlines = False
End Sub

```

10. To return to Microsoft Excel, click the View Microsoft Excel button 

11. To fill the worksheet, press Ctrl + Shift + W

Introduction to Formulas

A formula is another name for an expression. It combines one or more values, one or more variables, to an operator, to produce a new value. This also means that you use the same approach or building an expression when creating a formula.

To assist you with assigning the result of a formula to a cell or a group of cells, the Range class is equipped with a property named **Formula**. This property is of type Variant, which means its value can be anything, not necessarily a number. After accessing the Formula property, you can assign whatever value, expression, or formula you want to it. Here are examples:

```

Sub Exercise()
Rem Using the Formula property to assign a string to the active cell
ActiveCell.Formula = "Weekly Salary:"

Rem Using the Formula property to assign an expression to cell B2
Range("B2").Formula = 24.5 * 42.5

Rem Using the Formula property to assign
Rem the same string to a group of cells
Range("C2:F5, B8:D12").Formula = "Antoinette"
End Sub

```

If you are creating a worksheet that would be used on computers of different languages, use **FormulaLocal** instead. The **FormulaLocal** property is equipped to adapt to a different language-based version of Microsoft Excel when necessary.

Besides **Formula**, the **Range** class is also equipped with a property named **FormulaR1C1**. Its functionality is primarily the same as **Formula**. Here are examples:

```

Sub Exercise()
Rem Using the Formula property to assign a string to the active cell
ActiveCell.FormulaR1C1 = "Weekly Salary:"

Rem Using the Formula property to assign an expression to cell B2
Range("B2").FormulaR1C1 = 24.5 * 42.5

Rem Using the Formula property to assign
Rem the same string to a group of cells
Range("C2:F5, B8:D12").FormulaR1C1 = "Antoinette"
End Sub

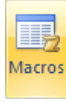
```

If you are creating the worksheet for various languages, use **FormulaR1C1Local** instead.

❖ Practical Learning: Creating Formulas

1. In the Developer tab of the Ribbon and in the Code section, click Record Macro 

2. Set the Macro Name to **CalculateOrder**
3. In the Shortcut Key text box, type **C** to get Ctrl + Shift + C
4. Click OK
5. On the Ribbon, click Stop Recording



6. In the Code section of the Ribbon, click Macros
7. In the Macro dialog box, make sure CalculateOrder is selected and click Edit
8. Change the code as follows:

```
Sub CalculateOrder()
'
' CreateWorkbook Macro
'
' Keyboard Shortcut: Ctrl+Shift+C
'
    Rem Calculate the sub-total of each category of items as:
    Rem SubTotal = Unit Price * Quantity
    Rem And display the total in the equivalent F cell
    Range("F15").Formula = Range("D15") * Range("E15")
    Range("F16").Formula = Range("D16") * Range("E16")
    Range("F17").Formula = Range("D17") * Range("E17")
    Range("F18").Formula = Range("D18") * Range("E18")
    Range("F19").Formula = Range("D19") * Range("E19")
    Range("F20").Formula = Range("D20") * Range("E20")

    Rem Retrieve the values of the cleaning total and the tax rate
    Rem Use them to calculate the amount of tax
    Range("I19").Formula = Range("I17") * Range("I18") / 100

    Rem Calculate the total order by adding
    Rem the cleaning total to the tax amount
    Range("I20").Formula = Range("I17") + Range("I19")
End Sub
```

Fundamentals of Built-In Functions

Introduction

Instead of creating your own function, you can use one of those that ship with the VBA language. This language provides a very extensive library of functions so that, before creating your own, check whether the function exists already. If so, use it instead.

To use a VBA built-in function, simply use as you would an expression. That is, assign its returned value to a cell. Here is an example:

```
Sub Exercise()
    Range("B2:B2") = Len("Paul Bertrand Yamaguchi")
End Sub
```

Microsoft Excel Built-In Functions

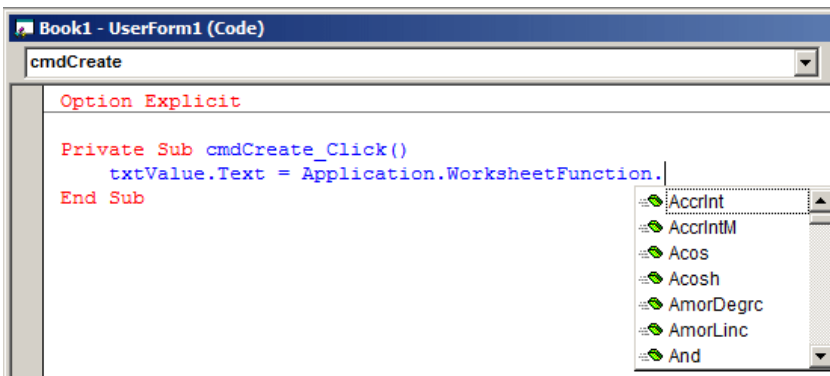
To assist you with developing smart worksheets, Microsoft Excel provides a very large library of functions.

To use a Microsoft Excel built-in function in your code, you have many functions.

In double-quotes, you can include the assignment operator followed by the function's whole expression. Here is an example:

```
Sub Exercise()
    Range("B5:B5") = "=SomeFunction(B2, B3, B4)"
End Sub
```

As an alternative, the Application class is equipped with a property named WorksheetFunction. This property represents all functions of the Microsoft Excel library. Therefore, to access a function, type Application, followed by a period, followed by a period. Then type (or select the name of the function you want to use:



After specifying the function you want to use, because it is a function, you must include the parentheses. In the parentheses, type the necessary argument(s). Here is an example:

```
Private Sub cmdCreate_Click()
    txtValue.Text = Application.WorksheetFunction.Sum(Range("D4:D8"))
End Sub
```

Conversion Functions

You may recall that when studying data types, we saw that each had a corresponding function used to convert a string value or an expression to that type. As a reminder, the general syntax of the conversion functions is:

```
ReturnType = FunctionName(Expression)
```

The *Expression* could be of any kind. For example, it could be a string or expression that would produce a value such as the result of a calculation. The conversion function would take such a value, string, or expression and attempt to convert it. If the conversion is successful, the function would return a new value that is of the type specified by the *ReturnType* in our syntax.

The conversion functions are as follows:

Function		
Name	Return Type	Description
CBool	Boolean	Converts an expression into a Boolean value
CByte	Byte	Converts an expression into Byte number
CDBl	Double	Converts an expression into a floating-point number with double precision
CDec	Decimal	Converts an expression into a decimal number
CInt	Integer	Converts an expression into an integer (natural) number
CLng	Long	Converts an expression into a long integer (a large natural) number
CObj	Object	Converts an expression into an Object type
CSByte	SByte	Converts an expression into a signed byte
CShort	Short	Converts an expression into a short integer
CSng	Single	Converts an expression into a floating-point number with single precision
CUInt	UInt	Converts an expression into an unsigned integer
CULng	ULong	Converts an expression into an unsigned long integer
CUShort	UShort	Converts an expression into an unsigned short integer

These functions allow you to convert a known value to a another type.

❖ Practical Learning: Using Conversion Functions

- Change the code as follows:

```
Sub CalculateOrder()
' CalculateOrder Macro
' Keyboard Shortcut: Ctrl+Shift+C

    Rem Calculate the sub-total of each category of items as:
    Rem SubTotal = Unit Price * Quantity
    Rem And display the total in the equivalent F cell
    Range("F15").Formula = CDBl(Range("D15")) * CInt(Range("E15"))
    Range("F16").Formula = CDBl(Range("D16")) * CInt(Range("E16"))
    Range("F17").Formula = CDBl(Range("D17")) * CInt(Range("E17"))
    Range("F18").Formula = CDBl(Range("D18")) * CInt(Range("E18"))
    Range("F19").Formula = CDBl(Range("D19")) * CInt(Range("E19"))
    Range("F20").Formula = CDBl(Range("D20")) * CInt(Range("E20"))

    Rem Retrieve the values of the cleaning total and the tax rate
    Rem Use them to calculate the amount of tax
    Range("I19").Formula = CDBl(Range("I17")) * CDBl(Range("I18")) / 100

    Rem Calculate the total order by adding the
    Rem cleaning total to the tax amount
    Range("I20").Formula = CDBl(Range("I17")) + CDBl(Range("I19"))
```

Accessory Built-In Functions

Introduction

Both Microsoft Excel and the Visual Basic language provide each an extensive library of functions. We refer to some functions as accessories because you almost cannot anything about them or at least they are very useful.

Specifying a Color

To assist you with specifying the color of anything, the VBA is equipped with a function named RGB. Its syntax is:

```
Function RGB(RedValue As Byte, GreenValue As Byte, BlueValue As Byte) As long
```

This function takes three arguments and each must hold a value between 0 and 255. The first argument represents the ratio of red of the color. The second argument represents the green ratio of the color. The last argument represents the blue of the color. After the function has been called, it produces a number whose maximum value can be $255 * 255 * 255 = 16,581,375$, which represents a color.

❖ Practical Learning: Using the RGB Function

1. Locate the CreateWorkbook procedure and change its code as follows:

```
Sub CreateWorkbook()
' CreateWorkbook Macro
' Keyboard Shortcut: Ctrl+Shift+E
'
' . . . No Change

Rem Change the background color of cells F15 to F20 to a light blue
Range("F15:F20").Interior.Color = RGB(210, 225, 250)
Rem Change the background color of cells I17 to I20 to a dark blue
Range("I17:I20").Interior.Color = RGB(5, 65, 165)
Rem Change the text color of cells I17 to I20 to a dark blue
Range("I17:I20").Font.Color = RGB(255, 255, 195)

Rem Hide the gridlines
ActiveWindow.DisplayGridlines = False
End Sub
```

2. Return to Microsoft Excel and press Ctrl + Shift + W
3. Press Ctrl + Shift + C to see the result

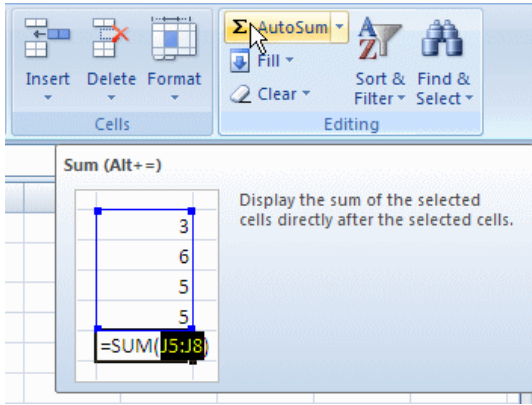
Item	Unit Price	Qty	Sub-Total
Shirts			0
Pants			0
None			0
None			0
None			0
None			0

Order Summary	
Cleaning Total:	
Tax Rate:	5.75 %
Tax Amount:	0
Order Total:	0

4. Return to Microsoft Visual Basic

The Sum Function

The Microsoft Excel's **SUM** function is used to add the numeric values of various cells. The result can be displayed in another cell or used in an expression. Like all functions of the Microsoft Excel library, you can use SUM visually or programmatically.



❖ Practical Learning: Using the SUM Function

1. Locate the CalculateOrder procedure and change its code as follows:

```
Sub CalculateOrder()
'
' CalculateOrder Macro
'
' Keyboard Shortcut: Ctrl+Shift+C
'
Rem Calculate the sub-total of each category of items as:
Rem SubTotal = Unit Price * Quantity
Rem And display the total in the equivalent F cell
Range("F15").Formula = CDb1(Range("D15")) * CInt(Range("E15"))
Range("F16").Formula = CDb1(Range("D16")) * CInt(Range("E16"))
Range("F17").Formula = CDb1(Range("D17")) * CInt(Range("E17"))
Range("F18").Formula = CDb1(Range("D18")) * CInt(Range("E18"))
Range("F19").Formula = CDb1(Range("D19")) * CInt(Range("E19"))
Range("F20").Formula = CDb1(Range("D20")) * CInt(Range("E20"))

Rem Use the SUM() function to calculate the sum of
Rem cells F15 to F20 and display the result in cell J17
Range("I17").Formula = "=SUM(F15:F20)"

Rem Retrieve the values of the cleaning total and the tax rate
Rem Use them to calculate the amount of tax
Range("I19").Formula = CDb1(Range("I17")) * CDb1(Range("I18")) / 100

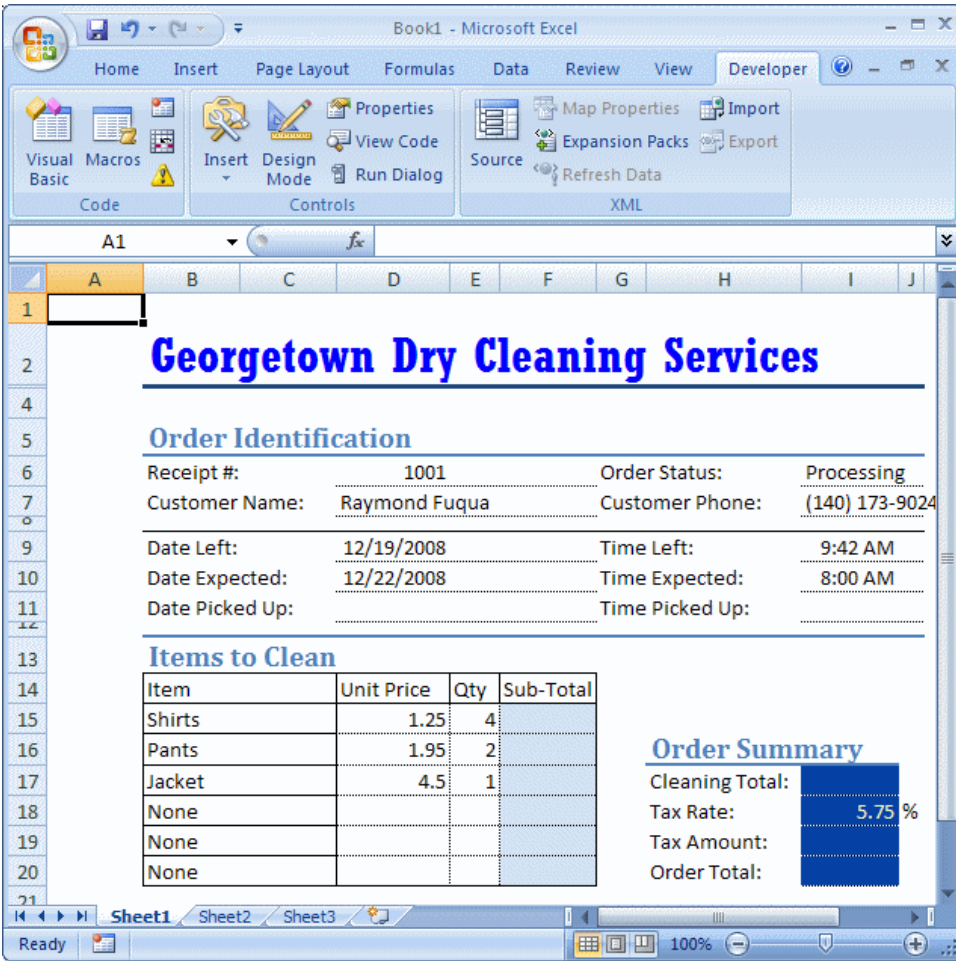
Rem Calculate the total order by adding the
Rem cleaning total to the tax amount
Range("I20").Formula = CDb1(Range("I17")) + CDb1(Range("I19"))
End Sub
```

2. Return to Microsoft Excel and press Ctrl + Shift + C to see the result
3. Enter the following values in the worksheet:

Receipt #:	1001	Order Status:	Processing
Customer Name:	Raymond Fuqua	Customer Phone:	(140) 173-9024
Date Left:	12/19/2008	Time Left:	09:42 AM
Date Expected:	12/22/2008	Time Expected:	08:00 AM

	Unit Price	Qty
Shirts	1.25	4
Pants	1.95	2
Jacket	4.50	1

4. Click cell A1



5. Press Ctrl + Shift + C



6. Return to Microsoft Visual Basic

The Absolute Value

The absolute value of a number x is x if the number is (already) positive. If the number is negative, then its absolute value is its positive equivalent. For example, the absolute value of 12

is 12, while the absolute value of -12 is 12.

To get the absolute value of a number, you can use either the Microsoft Excel's **ABS()** or the VBA's **Abs()** function. Their syntaxes are:

```
Function ABS(number) As Number
Function Abs(number) As Number
```

This function takes one argument. The argument must be a number or an expression convertible to a number:

- If the argument is a positive number, the function returns it
- If the argument is zero, the function returns 0
- If the argument is a negative number, the function is returns its equivalent positive value

Getting the Integral Part of a Number

If you have a decimal number but are interested only in the integral part, to assist you with retrieving that part, the Visual Basic language provides the **Int()** and the **Fix()** functions. In the same way, the Microsoft Excel library provides the **INT()** function to perform a similar operation. Their syntaxes are:

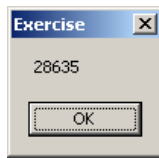
```
Function Int(ByVal Number As { Number | Expression } ) As Integer
Function Fix(ByVal Number As { Number | Expression } ) As Integer
Function ABS(ByVal Number As { Number | Expression } ) As Integer
```

Each function must take one argument. The value of the argument must be number-based. This means it can be an integer or a floating-point number. If the value of the argument is integer-based, the function returns the (whole) number. Here is an example

```
Sub Exercise()
    Dim Number As Integer

    Number = 28635
    ActiveCell = MsgBox(Int(Number), vbOKOnly, "Exercise")
End Sub
```

This would produce:

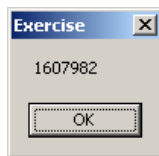


If the value of the argument is a decimal number, the function returns only the integral part. Here is an example

```
Sub Exercise()
    Dim Number As Double

    Number = 7942.225 * 202.46
    ActiveCell = MsgBox(Int(Number), vbOKOnly, "Exercise")
End Sub
```

This would produce:

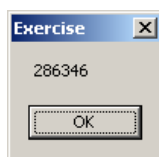


This function always returns the integral part only, even if you ask it to return a floating-point-based value. Here is an example:

```
Sub Exercise()
    Dim Number As Single

    Number = 286345.9924
    ActiveCell = MsgBox(Int(Number), vbOKOnly, "Exercise")
End Sub
```

This would produce:



Cells Content Formatting

Introduction

When it receive values for its cells, by default, Microsoft Excel displays text left aligned and numbers right aligned. In some situations, you will want to treat numbers as text.

Although Microsoft Excel displays all numbers right aligned, as a smart financial and business application, it can distinguish between different types of numbers. It can recognize a date, a currency, or a percentage values, but the computer wants you to specify the way numbers should be displayed, giving you the ability to decide what a particular number represents, not because the software cannot recognize a number, but because a value can represent different things to different people in different scenarios. For example 1.5 might represent a half teaspoon in one spreadsheet while the same 1.5 would represent somebody's age, another spreadsheet's percentage, or etc.

Introduction to Numbers Formatting

When it comes to displaying items, Microsoft Excel uses various default configurations. The computer's Regional Options or Regional Settings govern how dates, numbers, and time, etc get displayed on your computer.

Microsoft Excel recognizes numbers in various formats: accounting, scientific, fractions, and currency. As the software product can recognize a number, you still have the ability to display the number with a format that suits a particular scenario.

To visually control how a cell should display its number, on the Ribbon, click Home and use the Number section.

To assist you with programmatically specifying how a cell should display its number, the **Range** class is equipped with a property named **Style**.

To further assist with number formatting, the Visual Basic language provides a function named **Format**. This function can be used for different types of values. The most basic technique consists of passing it an expression that holds the value to display. The syntax of this function is:

```
Function Format(ByVal Expression As Variant, _
    Optional ByVal Style As String = "" _
) As String
```


The first argument is the value that must be formatted. Here is an example:

```
Sub Exercise()
    Dim Number As Double




    Number = 20502.48
    ActiveCell = Format(Number)
End Sub
```

The second argument is optionally. It specifies the type of format you want to apply. We will see various examples.

Formatting a Number

To visually specify that you want a cell to display its numeric value with the comma delimiter, click the cell to give it focus. Then, in the Home tab of the Ribbon, in the Number section, click the Comma Style button . The thousand numbers would display with a comma sign which makes it easier to read.

To visually control the number of decimal values on the right side of the comma, in the Number section of the Ribbon:

- You can click the Decrease Decimal button  to remove one decimal value. You can continuously click the Decrease Decimal button  to decrease the number of digits.
- You can click the Increase Decimal button  to increase the number of digits

To programmatically specify that you want a cell to display the comma style of number, assign the "Comma" string to the **Style** property of the **Range** class. Here is an example:

```
Sub SpecifyComma()
    ActiveCell.Style = "Comma"
End Sub
```

Alternatively

, to programmatically control how the number should display, you can pass the second argument to the **Format()** function. To produce the number in a general format, you can pass the second argument as "g", "G", "f", or "F".

To display the number with a decimal separator, pass the second argument as "n", "N", or "Standard". Here is an example:

```
Sub Exercise()
    Dim Number As Double

    Number = 20502.48
    ActiveCell = Format(Number, "STANDARD")
End Sub
```

An alternative to get this format is to call a function named **FormatNumber**. Its syntax is:

```
Function FormatNumber(
    ByVal Expression As Variant,
    Optional ByVal NumDigitsAfterDecimal As Integer = -1,
    Optional ByVal IncludeLeadingDigit As Integer,
    Optional ByVal UseParensForNegativeNumbers As Integer,
```

```
Optional ByVal GroupDigits As Integer
) As String
```

Only the first argument is required and it represents the value to display. If you pass only this argument, you get the same format as the **Format()** function called with the **Standard** option. Here is an example:

```
Sub Exercise()
    Dim Number As Double

    Number = 20502.48
    ActiveCell = FormatNumber(Number)
End Sub
```


This would produce the same result as above.

If you call the **Format()** function with the **Standard** option, it would consider only the number of digits on the right side of the decimal separator. If you want to display more digits than the number actually has, call the **FormatNumber()** function and pass a second argument with the desired number. Here is an example:

```
Sub Exercise()
    Dim Number As Double

    Number = 20502.48
    ActiveCell = FormatNumber(Number, 4)
End Sub
```

In the same way, if you want the number to display with less numbers on the right side of the decimal separator, specify that number.

We saw that you could click the Decrease Decimal button  on the Ribbon to visually control the number of decimal values on the right side of the comma and you could continuously click that button to decrease the number of digits. Of course, you can also exercise this control programmatically.

You can call the **Format()** function to format the number with many more options. To represent the integral part of a number, you use the # sign. To specify the number of digits to display on the right side of the decimal separator, type a period on the right side of # followed by the number of 0s representing each decimal place. Here is an example:

```
Sub Exercise()
    Dim Number As Double

    Number = 20502.48
    ActiveCell = Format(Number, "#.00000")
End Sub
```

The five 0s on the right side of the period indicate that you want to display 5 digits on the right side of the period. You can enter as many # signs as you want; it would not change anything. Here is an example:

```
Sub Exercise()
    Dim Number As Double

    Number = 20502.48
    ActiveCell = Format(Number, "#####.00000")
End Sub
```

This would produce the same result as above. To specify that you want to display the decimal separator, include its character between the # signs. Here is an example:

```
Sub Exercise()
    Dim Number As Double

    Number = 20502.48
    ActiveCell = Format(Number, "###,#####.00000")
End Sub
```

You can include any other character or symbol you want in the string to be part of the result, but you should include such a character only at the beginning or the end of the string, otherwise the interpreter might give you an unexpected result.

❖ Practical Learning: Using the SUM Function

1. Locate the CreateWorkbook procedure and change its code as follows:

```
Sub CreateWorkbook()
    '
    ' CreateWorkbook Macro
    '
    ' Keyboard Shortcut: Ctrl+Shift+W
    '
    . . . No Change

    Rem Format the values in the unit prices
    Range("D15").Style = "Comma"
    Range("D16").Style = "Comma"
    Range("D17").Style = "Comma"
    Range("D18").Style = "Comma"
    Range("D19").Style = "Comma"
    Range("D20").Style = "Comma"
```

```

Rem Format the values in the sub totals
Range("F15").Style = "Comma"
Range("F16").Style = "Comma"
Range("F17").Style = "Comma"
Range("F18").Style = "Comma"
Range("F19").Style = "Comma"
Range("F20").Style = "Comma"

Rem Format the values in the Order Summary section
Range("I17").Style = "Comma"
Range("I19").Style = "Comma"
Range("I20").Style = "Comma"

Rem Hide the gridlines
ActiveWindow.DisplayGridlines = False
End Sub

```

- Return to Microsoft Excel and press Ctrl + Shift + W
- Enter the following values in the worksheet:

Receipt #:	1001	Order Status:	Processing
Customer Name:	Raymond Fuqua	Customer Phone:	(140) 173-9024
Date Left:	12/19/2008	Time Left:	09:42 AM
Date Expected:	12/22/2008	Time Expected:	08:00 AM

	Unit Price	Qty
Shirts	1.25	4
Pants	1.95	2
Jacket	4.50	1

- Click cell A1



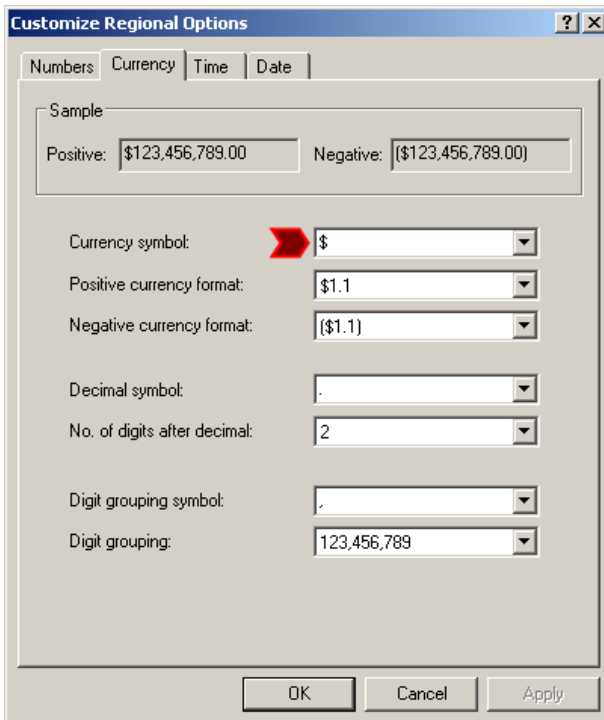
- Press Ctrl + Shift + C




6. Return to Microsoft Visual Basic

Formatting a Currency Value

Another regular type of number used in applications and finances is the currency. A currency value uses a special character specified in the Control Panel. In US English, this character would be the \$ sign:



To visually that a c

ell should display its number as currency, in the Number section of the Ribbon, click the Currency Style button .

To programmatically specify that you want a cell to display its value with the currency style, assign the "Currency" string to the **Style** property of the **Range** class. Here is an example:

```
Sub SpecifyComma()  
ActiveCell.Style = "Currency"
```

End Sub

Alternatively, to programmatically display the currency symbol in the result of a cell or a text box of a form, you can simply add it as part of the second argument to the **Format()** function. Here is an example:

```
Sub Exercise()
    Dim Number As Double

    Number = 205.5

    ActiveCell = Format(Number, "$###,#####.00")
End Sub
```

Fortunately, there are more professional options. Besides the **Format()** function, to support currency formatting of a number, the Visual Basic language provides the **FormatCurrency()** function. Its syntax is:

```
Function FormatCurrency(
    ByVal Expression As Variant,
    Optional ByVal NumDigitsAfterDecimal As Integer = -1,
    Optional ByVal IncludeLeadingDigit As Integer = -2,
    Optional ByVal UseParensForNegativeNumbers As Integer = -2,
    Optional ByVal GroupDigits As Integer = -2
) As String
```

Only the first argument is required. It is the value that needs to be formatted. Here is an example:

```
Sub Exercise()
    Dim UnitPrice As Double

    UnitPrice = 1450.5

    ActiveCell = FormatCurrency(UnitPrice)
End Sub
```

Notice that, by default, the **FormatCurrency()** function is equipped to display the currency symbol (which, in US English is, the \$ sign), the decimal separator (which in US English is the comma), and two decimal digits. If you want to control how many decimal digits are given to the result, pass a second argument as an integer. Here is an example:

```
Sub Exercise()
    Dim UnitPrice As Double

    UnitPrice = 1450.5

    ActiveCell = FormatCurrency(UnitPrice, 4)
End Sub
```


Instead of calling the **FormatCurrency()** function to format a number to currency, you can use the **Format()** function. If you do, pass it a second argument as "Currency", "c", or "C". Here is an example:

```
Sub Exercise()
    Dim CarPrice As Double

    CarPrice = 42790

    ActiveCell = Format(CarPrice, "Currency")
End Sub
```

Formatting a Percentage Value

A percentage of a number represents its rate on a scale, usually of 100 (or more). The number is expressed using digits accompanied by the % sign. To visually specify that a number in a cell should be treated a percentage value, in the Number section of the Ribbon, click the Percent Style button .

To programmatically use a percentage number in a cell or the control of a form, you can use the **Format()** function. Besides the **Format()** function, to support percent values, the Visual Basic language provides a function named **FormatPercent()**. Its syntax is:

```
Function FormatPercent(
    ByVal Expression As Variant,
    Optional ByVal NumDigitsAfterDecimal As Integer = -1,
    Optional ByVal IncludeLeadingDigit As Integer = -2,
    Optional ByVal UseParensForNegativeNumbers As Integer = -2,
    Optional ByVal GroupDigits As Integer = -2
) As String
```

Only the first argument is required and it is the number that needs to be formatted. When calling this function, pay attention to the number you provide as argument. If the number represents a percentage value as a fraction of 0 to 1, make sure you provide it as such. An example would be 0.25. In this case, the Visual Basic interpreter would multiply the value by 100 to give the result. Here is an example:

```
Sub Exercise()
    Dim DiscountRate As Double

    DiscountRate = 0.25
    ActiveCell = FormatPercent(DiscountRate)
End Sub
```


If you pass the value in the hundreds, the interpreter would still multiply it by 100. Although it is not impossible to get a percentage value in the hundreds or thousands, you should make sure that's the type of value you mean to get.

Besides the **FormatPercent()** function, to format a number to its percentage equivalent, you can call the **Format()** function and pass the second argument as **"Percent"**, **"p"**, or **"P"**. Here is an example:

```
Sub Exercise()
    Dim DiscountRate As Double

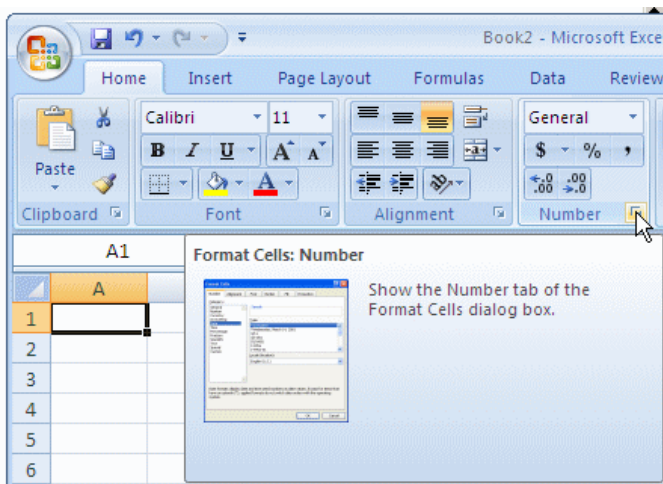
    DiscountRate = 0.25
    ActiveCell = MsgBox("Discount Rate: " & _
        Format(DiscountRate, "Percent"), _
        vbOKOnly, "Exercise")
End Sub
```

Number Format Options

Although you can do most of cells configurations using the Ribbon, Microsoft Excel provides the Format Cells dialog box. This dialog box presents more options and more precision.

To display the Format Cells dialog box:

- On the Ribbon, click Home. In the Number section, click the more options button:



- Right-click the cell or group of cells whose format you want to change and click Format Cells...
- Press Ctrl + 1 as a shortcut



Strings

Introduction to Strings

A String

A string is one or a combination of characters. To declare a variable for it, you can use either **String** or the **Variant** data types. To initialize the variable, put its value in double-quotes and assign it to the variable. Here are examples:

Here is an example:

```
Sub Exercise()
    ActiveCell = "AAA"
End Sub
```

When this code runs, the value AAA would be entered into any cell that is currently selected.

```
Sub Exercise()
    Dim FirstName As Variant
    Dim LastName As String

    FirstName = "William"
    LastName = "Sansen"
End Sub
```

Producing a Beeping Sound

If you want, you can make the computer produce a beeping a sound in response to something, anything. To support this, the Visual Basic language provides a function called **Beep**. Its syntax is:

```
Public Sub Beep()
```

Here is an example of calling it:

```
Sub Exercise()
    Beep
End Sub
```

If this function is called when a program is running, the computer emits a brief sound.

String Concatenation

A string concatenation consists of adding one string to another. to support this operation, you can use either the + or the & operator. Here are examples:

```
Sub Exercise()
    Dim FirstName As Variant
    Dim LastName As String
    Dim FullName As String

    FirstName = "William"
    LastName = "Sansen"
    FullName = LastName + ", " & FirstName

    ActiveCell = "Full Name: " & FullName
End Sub
```

This would produce:

	A	B	C	D	E	F	G
1							
2		Full Name: Sansen, William					
3							

Introduction to Characters

Getting the ASCII Character of a Number

The characters used in the US English and the most common characters of Latin-based languages are created in a list or map of character codes. Each character is represented with a small number between 0 and 255. This means that each character must fit in a byte.

To help you find the equivalent ASCII character of such a number, the Visual Basic language provides a function named **Chr**. Its syntax is:

```
Public Function Chr(ByVal CharCode As Integer) As String
```

When calling this function, pass a small number as argument. Here is an example:

```
Sub Exercise()
    Dim Character As String
    Dim Number As Integer

    Number = 114
    Character = Chr(Number)

    ActiveCell = "The ASCII character of " & Number & " is " & Character
End Sub
```

This would produce:

	A	B	C	D	E	F	G
1							
2							
3		The ASCII character of 114 is r					
4							

Besides finding the ASCII equivalent of a number, the **Chr()** function can be used to apply some behavior in a program. For example, a combination of **Chr(13)** and **Chr(10)** would break a line in an expression, which is equivalent to the **vbCrLf** operator.

Getting the Wide ASCII Character of a Number

If you pass a number lower than 0 or higher than 255 to the **Chr()** function, you would receive an error. The reason you may pass a number higher than 255 is that you may want to get a character beyond those of US English, such as å. To support such numbers, the Visual Basic language provides another version of the function. Its syntax is:

```
Public Function ChrW(ByVal CharCode As Integer) As String
```

The **W** here represents Wide Character. This makes it possible to store the character in the memory equivalent to the **Short** integer data type, which can hold numbers from -32768 to 32767. Normally, you should consider that the character should fit in a **Char** data type, which should be a positive number between 0 and 65535.

Here is an example:

```
Sub Exercise()
    Dim Character As String
    Dim Number As Long

    Number = 358
    Character = ChrW(Number)

    ActiveCell = "The ASCII character of " & Number & " is " & Character
End Sub
```

This would produce:

	A	B	C	D	E	F	G
1							
2		The ASCII character of 358 is 𐀀					
3							

The Length of a String

The length of a string is the number of characters it contains. To assist you with finding the length of a string, the Visual Basic language provides a function named **Len**. Its syntax is:

```
Public Function Len(ByVal Expression As String) As Integer
```

This function expects a string as argument. If the function succeeds in counting the number of characters, which it usually does, it returns the an integer. Here is an example:

```
Sub Exercise()
    Dim Item As String
    Dim Length As Integer

    Item = "Television"
    Length = Len(Item)

    ActiveCell = "The number of characters in "" & Item & "" is " & Length
End Sub
```

This would produce:

	A	B	C	D	E	F	G	H
1								
2		The number of characters in "Television" is 10						
3								

The Microsoft Excel library provides the `LEN()` function that produces the same result.

Characters, Strings, and Procedures

Passing a Character or a String to a Procedure

Like a normal value, a character or a string can be passed to a procedure. When creating the procedure, enter the argument and its name in the parentheses of the procedure. Then, in the body of the procedure, use the argument as you see fit. When calling the procedure, you can pass a value for the argument in double-quotes. In the same way, you can apply any of the features we studied for procedures, including passing as many arguments as you want or passing a mixture of characters, strings, and other types of arguments. You can also create a procedure that receives an optional argument.

Returning a Character or a String From a Function

To create a function that returns a character or a string, create the procedure using the **Function** keyword and, on the right side of the parentheses, include the **String** data type preceded by the **As** keyword or use the `$` character. Here is an example we saw in Lesson 5:

```
Function GetFullName$()
    Dim FirstName$, LastName$

    FirstName = "Raymond"
    LastName = "Kouma"

    GetFullName$ = LastName & ", " & FirstName
End Function
```

When calling the function, you can use it as a normal function or you can retrieve the value it returns and use it as you see fit. Here is an example:

```
Function GetFullName$()
    Dim FirstName$, LastName$

    FirstName = "Raymond"
    LastName = "Kouma"

    GetFullName$ = LastName & ", " & FirstName
End Function

Sub Exercise()
    Range("B2") = GetFullName$
End Sub
```

Character and String Conversions

Introduction

To convert an expression to a string, you can call the VBA's `CStr()` function. Its syntax is:

```
Public Function CStr(ByVal Expression As Variant) As String
```

The argument can be almost any expression that can be converted it to a string, which in most cases it can. If it is successful, the function returns a string. Here is an example:

```
Sub Exercise()
    Dim DateHired As Date

    DateHired = #1/4/2005#
    ActiveCell = CStr(DateHired)
End Sub
```

The `CStr()` function is used to convert any type of value to a string. If the value to be converted is a number, you can use the `Str()` function. Its syntax is:

```
Public Function Str(ByVal Number As Variant) As String
```

This function expects a number as argument. Here is an example:

```
Sub Exercise()
    Dim Number As Double

    Number = 1450.5 / 2
    ActiveCell = Str(Number)
End Sub
```

Numeric Hexadecimal Conversion

In Lesson 3, we saw that the Visual Basic language supports hexadecimal number and we saw how to initialize an integer variable with a hexadecimal number. Now, on the other hand, if you have a decimal number but need it in hexadecimal format, you can convert it. To support this operation, you can call the **Hex()** function. Its syntax is:

```
Function Hex(ByVal Number As { Byte | Integer | Long | Variant} ) As String
```

This function is used to convert either an integer-based or a decimal number to its hexadecimal equivalent. It returns the result as a string. Here is an example:

```
Sub Exercise()
    Dim Number As Integer

    Number = 28645
    ActiveCell = Hex(Number)
End Sub
```

The Microsoft Excel library provides more functions to perform this type of operation.

Numeric Octal Conversion

If you have a decimal number you want to convert to its octal format, you can call the **Oct()** function. Its syntax is:

```
Function Oct(ByVal Number As { Byte | Integer | Long | Variant} ) As String
```

This function takes an integer-based or a decimal number and converts its octal equivalent. It returns the result as a string. Here is an example:

```
Sub Exercise()
    Dim Number As Double

    Number = 28645
    ActiveCell = Oct(Number)
End Sub
```

The Microsoft Excel library provides more functions to perform this type of operation.

Case Conversion

If you are presented with a string or an expression whose cases must be the same, you can convert all of its characters in either uppercase or lowercase.

To convert a character, a string or an expression to uppercase, you can call the VBA's **UCase()** or the Microsoft Excel's **UPPER()** functions. These functions take one argument as the string or expression to be considered. The syntaxes are:

```
Function UCASE(ByVal Value As String) As String
Function UPPER(ByVal Value As String) As String
```

Each function receives a character or string as argument. If a character is already in uppercase, it would be returned the same. If the character is not a readable character, no conversion would happen and the function would return it. If the character is in lowercase, it would be converted to uppercase and the function would then return the uppercase equivalent.

Here is an example:

```
Sub Exercise()
    Dim ProgrammingEnvironment As String

    ProgrammingEnvironment = "Visual Basic for Application for Microsoft Excel"
    ActiveCell = UCASE(ProgrammingEnvironment)
End Sub
```

To convert a character or a string to lowercase, you can call the VBA's **LCASE()** or the Microsoft Excel's **LOWER()** functions. Their syntaxes are:

```
Function LCASE(ByVal Value As String) As String
Function LOWER(ByVal Value As String) As String
```

The function takes a character or a string as argument. If a character is not a readable symbol, it would be kept "as is". If the character is in lowercase, it would not be converted. If the character is in uppercase, it would be converted to lowercase.

The Sub-Strings of a String

Introduction

A sub-string is a character or a group of characters or symbols that are part of an existing string. The Visual Basic language provides functions to create, manipulate, or manage sub-strings. The primary rule to keep in mind is that a sub-string is part of, and depends on, a string. In other words, you cannot have a sub-string if you do not have a string in the first place.

The Left Sub-String of a String

If you have an existing string but want to create a new string using a number of characters from the left side characters of the string, you can use the Microsoft Excel's **LEFT()** or the VBA's **Left()** functions. Their syntaxes are:

Each function takes two arguments and both are required. The first argument is the existing string. The second argument is the number of characters counted from the left side of the string. Here is an example:

```
Sub Exercise()  
    Dim Process As String  
  
    Process = "learning"  
    ActiveCell = "To " & Left(Process, 5) & " is to gain understanding"  
End Sub
```

This would produce:

	A	B	C	D	E
1					
2					
3		To	learn	is	to
4		gain	understanding		

The Right Sub-String of a String

To create a new string using one or more characters from the right side of an existing string, call the Microsoft Excel **RIGHT()** or the VBA's **Right()** functions. Its syntax is:

```
Function RIGHT(ByVal str As String, ByVal Length As Integer) As String  
Function Right(ByVal str As String, ByVal Length As Integer) As String
```

Both arguments are required. The first argument is the original string. The second argument is the number of characters counted from the right side of the string.

The Mid Sub-String of a String

You may want to create a string using some characters either from the left, from the right, or from somewhere inside an existing string. To assist you with this, the Visual Basic language provides a function named **Mid** and the Microsoft Excel library is equipped with a function named **MID**. Here is an example of calling the **Mid()** function:

```
Sub Exercise()  
    Dim ProgrammingEnvironment As String  
  
    ProgrammingEnvironment = "VBA for Microsoft Excel"  
    ActiveCell = "The " & Mid(ProgrammingEnvironment, 10, 13) & " language"  
End Sub
```

Finding a Sub-String

One of the most regular operations you will perform on a string consists of finding out whether it contains a certain character or a certain contiguous group of characters. To help you with this operation, the Visual Basic language provides the **InStr()** function and the Microsoft Excel library equipped with the **FIND()** function. Their syntaxes are:

```
InStr([start, ]string1, string2[, compare])  
FIND([Find_Text, Within_Text, Start_Num])
```

In the first version of the function, the *String1* argument is the string on which the operation will be performed. The *String2* argument is the character or the sub-string to look for. If *String2* is found in *String1* (as part of *String1*), the function return the position of the first character. Here is an example:

The first version of the function asks the interpreter to check *String1* from the left looking for *String2*. If *String1* contains more than one instance of *String2*, the function returns (only) the position of the first instance. Any other subsequent instance would be ignored. If you want to skip the first instance or want the interpreter to start checking from a position other than the left character, use the second version. In this case, the *Start* argument allows you to specify the starting position from where to start looking for *String2* in *String1*.

The **InStr()** function is used to start checking a string from the left side. If you want to start checking from the right side, call the **InStrRev()** function. Its syntax is:

```
InstrRev(stringcheck, stringmatch[, start[, compare]])
```

Replacing a Character or a Sub-String in a String

After finding a character or a sub-string inside of a string, you can take action on it. One of the operations you can perform consists of replacing that character or that sub-string with another character or a sub-string. To do this, the Visual Basic language provides the **Replace()** function and Microsoft Excel provides the **REPLACE()** function. Its syntax is:

```
Replace(expression, find, replace[, start[, count[, compare]])  
REPLACE(Old_Text, Find_Text, Start_Num, Num_Characters, New_Text)
```

The first argument is the string on which the operation will be performed. The second argument is the character or string to look for in the *Expression*. If that character or string is found, the third argument is the character or string to replace it with.

Other Operations on Strings

Reversing a String

Once a string has been initialized, one of the operations you can perform on it consists of reversing it. To do this, you can call the **StrReverse()** function. Its syntax is:

```
Function StrReverse(ByVal Expression As String) As String
```

This function takes as argument the string that needs to be reversed. After performing its operation, the function returns a new string made of characters in reverse order. Here is an example:

```
Sub Exercise()
    Dim StrValue As String
    Dim StrRev As String

    StrValue = "République d'Afrique du Sud"
    StrRev = StrReverse(StrValue)

    ActiveCell = StrValue & vbCrLf & StrRev
End Sub
```

Because the **StrReverse()** function returns a string, you can write it as **StrReverse\$**.

Strings and Empty Spaces

The simplest string is probably one that you declared and initialized. In some other cases, you may work with a string that you must first examine. For example, for some reason, a string may contain an empty space to its left or to its right. If you simply start performing a certain operation on it, the operation may fail. One of the first actions you can take on a string would consist of deleting the empty space(s), if any on its sides.

To remove all empty spaces from the left side of a string, you can call the **LTrim()** function. Its syntax is:

```
Function LTrim(ByVal str As String) As String
```

To remove all empty spaces from the right side of a string, you can call the **RTrim()** function. Its syntax is:

```
Function RTrim(ByVal str As String) As String
```

To remove the empty spaces from both sides of a string, you can call the **Trim()** function. Its syntax is:

```
Function Trim(ByVal str As String) As String
```

Creating an Empty Spaced String

If you want to create a string made of one or more empty spaces, you can call the **Space()** function. Its syntax is:

```
Function Space(ByVal Number As Integer) As String
```

This function is the programmatic equivalent to pressing the Space bar when typing a string to insert an empty space between two characters.

The Message Box

Introduction

A message box is a special dialog box used to display a piece of information to the user. The user cannot type anything in the message box. There are usually two kinds of message boxes you will create: one that simply displays information and one that expects the user to make a decision.

A message box is created using the **MsgBox** function. Its syntax is:

```
Function MsgBox(Prompt[, Buttons] [, Title] [, Helpfile, Context]) As String
```

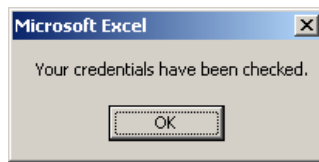
The **MsgBox()** function takes five arguments and only the first one is required.

The Message of a Message Box

The *Prompt* argument is the string that the user will see displaying on the message box. As a string, you can display it in double quotes, like this "Your credentials have been checked.". Here is an example:

```
Sub Exercise()
    MsgBox ("Your credentials have been checked.")
End Sub
```

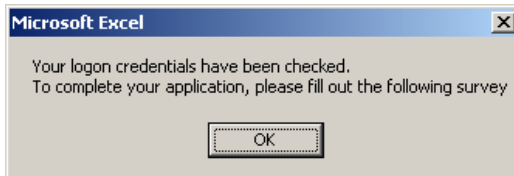
This would produce:



You can also create the message from other pieces of strings. The *Prompt* argument can be made of up to 1024 characters. To display the *Prompt* on multiple lines, you can use either the constant `vbCrLf` or the combination `Chr(10) & Chr(13)` between any two strings. Here is an example:

```
Sub Exercise()
    MsgBox ("Your logon credentials have been checked." & _
        vbCrLf & "To complete your application, please " & _
        "fill out the following survey")
End Sub
```

This would produce:



If you call the `MsgBox()` function with only the first argument, it is referred to as a method (a method is a member function of a class; the class in this case is the `Application` on which you are working). If you want to use the other arguments, you must treat `MsgBox` as a function. That is, you must assign it to a variable or to an object.

The Buttons of a Message Box

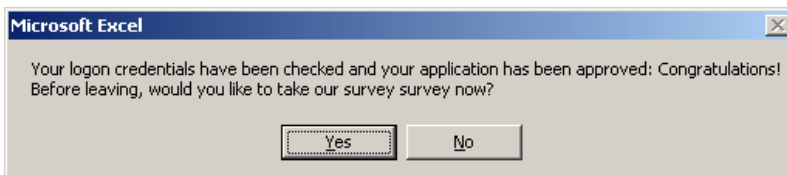
The *Buttons* argument specifies what button(s) should display on the message box. There are different kinds of buttons available and the VBA language. Each button uses a constant integer as follows:

Constant	Numeric Value	Display
<code>vbOKOnly</code>	0	
<code>vbOKCancel</code>	1	
<code>vbAbortRetryIgnore</code>	2	
<code>vbYesNoCancel</code>	3	
<code>vbYesNo</code>	4	
<code>vbRetryCancel</code>	5	

When calling the `MsgBox()` function and specifying the button, you can use one of the above constant numeric values. Here is an example that displays the Yes and the No buttons on the message box:

```
Sub Exercise()
    ActiveCell = MsgBox("Your logon credentials have been checked " & _
        "and your application has been approved: " & _
        "Congratulations!" & vbCrLf & _
        "Before leaving, would you like " & _
        "to take our survey survey now?", vbYesNo)
End Sub
```

This would produce:



The Icon on a Message Box

Besides the buttons, to enhance your message box, you can display an icon in the left section of the message box. To display an icon, you can use or add a member of the `MsgBoxStyle` enumeration. The members that are meant to display an icon are:

Icon Constant	Numeric Value	Description
<code>vbCritical</code>	16	
<code>vbQuestion</code>	32	
<code>vbExclamation</code>	48	

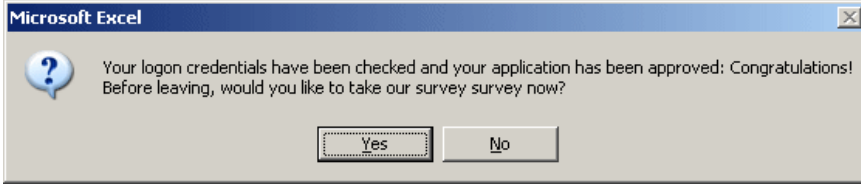
vbInformation	64		
---------------	----	--	--

To use one of these icons, you must combine the value of the button to the desired value of the icon. To perform this combination, you use the **OR** operator. Here is an example:

```
Sub Exercise()
    Dim iAnswer As Integer

    iAnswer = MsgBox("Your logon credentials have been checked " & _
        "and your application has been approved: Congratulations!" & _
        vbCrLf & "Before leaving, would you like " & _
        "to take our survey survey now?", vbYesNo Or vbQuestion)
End Sub
```

This would produce:



When calling the **MsgBox()** function, if you want to show one or more buttons and to show an icon, you can use either two members of the **MsgBoxStyle** enumeration using the **OR** operator, or you can add one of the constant values of the buttons to another constant values for an icon. For example, $3 + 48 = 51$ would result in displaying the buttons Yes, No, and Cancel, and the exclamation icon.

The Default Button of a Message Box

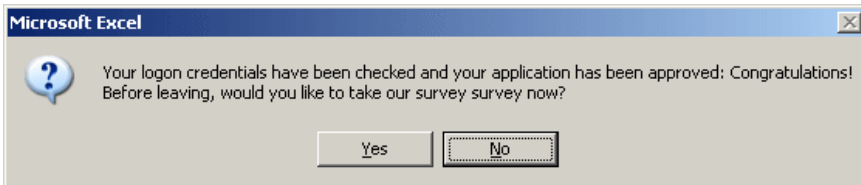
If you create a message box with more than one button, the most left button usually has a thick border, indicating that it is the default. If the user presses Enter after viewing the button, the effect would be the same as if he had clicked the default button. If you want, you can designate another button as the default. To do this, you can use or add another member of the **MsgBoxStyle** enumeration. The members used to specify the default button are:

Default Button Constant	Numeric Value	If the message box contains more than one button, the default would be
vbDefaultButton1	0	The first button
vbDefaultButton2	256	The second button
vbDefaultButton3	512	The third button

Once again, to specify a default value, use the **OR** operator to combine a Default Button Constant with any other combination. Here is an example:

```
Sub Exercise
    ActiveCell = MsgBox("Your logon credentials have been checked " & _
        "and your application has been approved: Congratulations!" & _
        vbCrLf & "Before leaving, would you like " & _
        "to take our survey survey now?", _
        vbYesNo Or _
        vbQuestion Or vbDefaultButton2)
End Sub
```

This would produce:



These additional buttons can be used to further control what the user can do:

Constant	Value	Effect
vbApplicationModal	0	The user must dismiss the message box before proceeding with the current database
vbSystemModal	4096	The user must dismiss this message before using any other open application of the computer

The Title of a Message Box

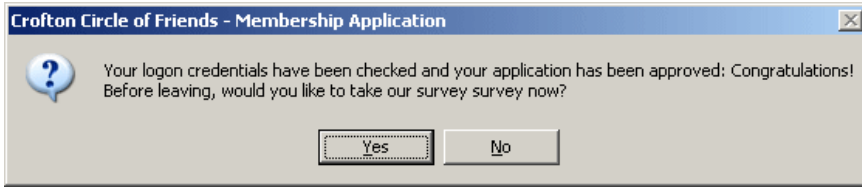
The *Title* argument is the caption that would display on the title bar of the message box. It is a string whose word or words you can enclose between parentheses or that you can get from a created string. The *Title* argument is optional. As you have seen so far, if you omit, the message box would display the name of the application on the title bar. Otherwise, if you want a custom title, you can provide it as the third argument to the **MsgBox()** function. The caption can be a simple string. Here is an example:

```

Sub Exercise()
    ActiveCell = MsgBox("Your logon credentials have been checked " & _
        "and your application has been approved: Congratulations!" & _
        vbCrLf & "Before leaving, would you like " & _
        "to take our survey survey now?", _
        vbYesNo Or vbQuestion, _
        "Crofton Circle of Friends - Membership Application")
End Sub

```

This would produce:



Notice that the caption is now customized instead of the name of the application. The caption can also be a string created from an expression or emanating from a variable or value.

The Returned Value of a Message Box

The **MsgBox()** function can be used to return a value. This value corresponds to the button the user clicked on the message box. Depending on the buttons the message box is displaying, after the user has clicked, the **MsgBox()** function can return a value. The value can be a member of the **MsgBoxResult** enumeration or a constant numeric value recognized by the Visual Basic language. The value returned can be one of the following values:

If the user click	The function returns	Numeric Value
	vbOK	1
	vbCancel	2
	vbAbort	3
	vbRetry	4
	vbIgnore	5
	vbYes	6
	vbNo	7

The Input Box

Introduction

The Visual Basic language provides a function that allows you to request information from the user who can type it in a text field of a dialog box. The function used to accomplish this is called **InputBox** and its basic syntax is:

```
InputBox(prompt[, title] [, default] [, xpos] [, ypos] [, helpfile, context])
```

Presenting the Message

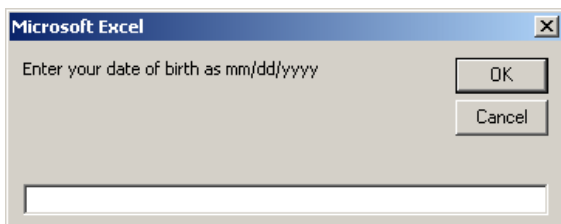
The most basic piece of information you can provide to the **InputBox()** function is referred to as the prompt. It should be a string that the user will read and know what you are expecting. Here is an example:

```

Sub Exercise()
    InputBox("Enter your date of birth as mm/dd/yyyy")
End Sub

```

This would produce



Upon reading the message on the input box, the user is asked to enter a piece of information. The type of information the user is supposed to provide depends on you, the programmer. Therefore, there are two important things you should always do. First you should let the user know what type of information is requested. Is it a number (what type of number)? Is it a string (such as the name of a country or a customer's name)? Is it the location of a file (such as C:\Program Files\Homework)? Are you expecting a Yes/No True/False type of answer (if so, how should the user provide it)? Is it a date (if it is a date, what format is the user supposed to enter)? These questions mean that you should state a clear request to the user and specify what kind of value you are expecting. A solution, also explicit enough, consists of providing an

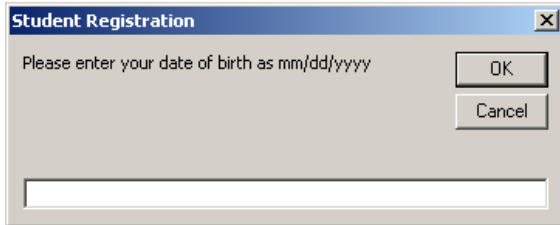
The Title of an Input Box

The second argument to the `InputBox()` function allows you to optionally specify the title of the input box. This is the string that would appear on the title bar. Since this is an optional argument, if you don't pass it, the input box would display the name of the application. Otherwise, to display your own title bar, pass the *Title* argument.

The title is passed as a string. Here is an example:

```
Sub Exercise()
    ActiveCell = InputBox("Please enter your date of birth as mm/dd/yyyy", _
        "Student Registration")
End Sub
```

This would produce:



Notice that the caption is now customized instead of the name of the application. The caption can also be a string created from an expression or emanating from a variable or value.

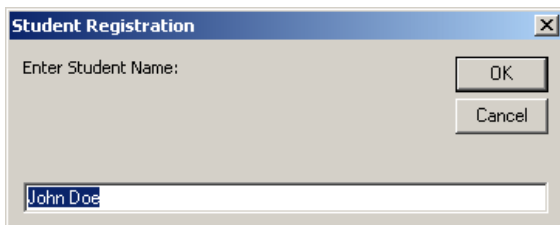
The Default Value of an Input Box

Sometimes, even if you provide an explicit request, the user might not provide a new value but click OK. The problem is that you would still need to get the value of the text box and you might want to involve it in an expression. You can solve this problem and that of providing an example to the user by filling the text box with a default value. To support this, the `InputBox()` function provides the third argument.

To present an example or default value to the user, pass a third argument to the `InputBox()` function. If you want to use this argument to provide an example the user can follow, provide it with the right format. Here is an example:

```
Sub Exercise()
    ActiveCell = InputBox("Enter Student Name:", _
        "Student Registration", "John Doe")
End Sub
```

Here is an example of running the program:



Notice that, when the input box displays with a default value, the value is in the text box and the value is selected. Therefore, if the value is fine, the user can accept it and click OK. Another way you can use the default value is to provide a value the user can accept; that is, the most common or most likely value the user would enter. Here is an example:

```
Sub Exercise()
    ActiveCell = InputBox("Enter Birth State:", _
        "Student Registration", "VA")
End Sub
```

Here is an example of running the program:



Once again, notice that the user can just accept the value and click OK or press Enter.

The Location of the Input Box

By default, when the input box comes up, it displays in the middle of the screen. If you want, you can specify where the input box should be positioned when it comes up. To assist you with this, the `InputBox()` function is equipped with a fourth and a fifth arguments. The fourth

argument specifies the x coordinate of the input box; that is, the distance from its left border to the left border of the monitor. The fifth argument specifies the distance from the top border of the input box to the top border of the monitor.

The Return Value of an Input Box

When the input box displays, after typing a value, the user would click one of the buttons: OK or Cancel. If the user clicks OK, you should retrieve the value the user would have typed. It is also your responsibility to find out whether the user typed a valid value. Because the **InputBox()** function can return any type of value, it has no mechanism of validating the user's entry. To retrieve the value of the input box dialog when the user clicks OK, you can get the returned value of the **InputBox()** function.

After being used, the **InputBox()** function returns a string. Here is an example of getting it:

```
Sub Exercise()
    Dim StudentName As String

    StudentName = InputBox("Enter Student Name:", _
        "Student Registration")
    MsgBox ("Student Name: " & StudentName)
End Sub
```

You can also get any type of value from an input box. That is, when the **InputBox()** function exits, thanks to the flexibility of the Visual Basic language, the compiler can directly cast the returned value for you. Here is an example:

```
Sub Exercise()
    Dim DateOfBirth As Date

    DateOfBirth = InputBox("Please enter your date of birth as mm/dd/yyyy", _
        "Student Registration")
    MsgBox("Date of Birth: " & DateOfBirth)
End Sub
```



Introduction to Conditions

Introduction to Boolean Values

Introduction

A value is referred to as Boolean if it can be either true or false. As you may imagine, the essence of a Boolean value is to check that a condition is true or false, valid or invalid.

The Boolean Data Type

Like a number or a string, a Boolean value can be stored in a variable. To declare such a variable, use the **Boolean** keyword. Here is an example:

```
Sub Exercise()
    Dim EmployeeIsMarried As Boolean
End Sub
```

To actually use a Boolean variable, you can assign a value to it. By default, if you declare a Boolean variable but do not initialize it, it receives a value of **False**:

```
Sub Exercise()
    Dim EmployeeIsMarried As Boolean

    Range("B2").FormulaR1C1 = "Employee Is Married? " & EmployeeIsMarried
End Sub
```

This would produce:

	A	B	C	D	E
1					
2		Employee Is Married? False			

To initialize a Boolean variable, assign it a **True** or a **False** value. In the Visual Basic language, a Boolean variable can also [deal](#) with numeric values. The **False** value is equivalent to 0. For example, instead of **False**, you can initialize a Boolean variable with 0. Any other numeric value, whether positive or negative, corresponds to True:

```
Sub Exercise()
    Dim EmployeeIsMarried As Boolean

    EmployeeIsMarried = -792730
    Range("B2").FormulaR1C1 = "Employee Is Married? " & EmployeeIsMarried
End Sub
```

This would produce:

	A	B	C	D	E
1					
2		Employee Is Married? True			

The number can be decimal or hexadecimal:

```
Sub Exercise()
    Dim EmployeeIsMarried As Boolean

    EmployeeIsMarried = &HFA26B5
    Range("B2").FormulaR1C1 = "Employee Is Married? " & EmployeeIsMarried
End Sub
```

Boolean Values and Procedures

Introduction

As done with the other data types that we have used so far, Boolean values can be involved with procedures. This means that a Boolean variable can be passed to a procedure and/or a function can be made to return a Boolean value. Some of the issues involved with procedures require conditional statements that we will study in the next lesson. Still, the basic functionality is possible with what we have learned so far.

Passing a Boolean Variable as Argument

To pass an argument as a Boolean value, in the parentheses of the procedure, type the name of the argument followed by the **As Boolean** expression. Here is an example:

```
Private Sub CheckingEmployee(ByVal IsFullTime As Boolean)
End Sub
```

In the same way, you can pass as many Boolean arguments as you need, and you can combine Boolean and non-Boolean arguments as you judge necessary. Then, in the body of the procedure, use (or do not use) the Boolean argument as you wish.

Returning a Boolean Value

Just as done for the other data types, you can create a function that returns a Boolean value. When declaring the function, specify its name and the **As Boolean** expression on the right side of the parentheses. Here is an example:

```
Public Function IsDifferent() As Boolean
```

Of course, the function can take arguments of any kind you judge necessary:

```
Public Function IsDifferent(ByVal Value1 As Integer, _
                          ByVal Value2 As Integer) As Boolean
```

In the body of the function, do whatever you judge necessary. Before exiting the function, you must return a value that evaluates to True or False. We will see **an example** below.

Boolean Built-In Functions

Converting a Value to Boolean

To assist you with validating some values or variables to true or false, the Visual Basic language provides many functions. First, to convert a value to Boolean, you can use the **CBool()** function. Its syntax is:

```
Function CBool(ByVal Expression As Variant) As Boolean
```

Like all conversion functions, **CBool** takes one argument, the expression to be evaluated. It should produce a valid Boolean value. If it does, the function returns **True** or **False**.

Checking Whether a Variable Has Been Initialized

After declaring a variable, memory is reserved for but you should assign value to it before using it. At any time, to check whether a variable has been initialized, you can call the **IsEmpty()** function. Its syntax is:

```
Public Function IsEmpty(ByVal Expression As Variant) As Boolean
```

When calling this function, pass the name of a variable to it. If the variable was already initialized, the function would return **True**. Otherwise, it would return **False**.

Checking Whether a Value is Numeric

One of the most valuable operations you will perform on a value consists of finding out whether it is numeric or not. To assist you with this, the Visual Basic language provides a function named **IsNumeric**. Its syntax is:

```
Public Function IsNumeric(ByVal Expression As Variant) As Boolean
```

This function takes as argument the value or expression to be evaluated. If the argument holds or can produce a valid integer or a decimal value, the function returns True. Here is an example:

```
Sub Exercise()
    Dim Value As Variant

    Value = 258.08 * 9920.3479

    Range("B2").FormulaR1C1 = "Is Numeric? " & IsNumeric(Value)
End Sub
```

This would produce:

	A	B	C	D
1				
2		Is Numeric? True		

If the argument is holding any other value that cannot be identified as a number, the function produces **False**. Here is an example:

```
Sub Exercise()
    Dim Value As Variant

    Value = #12/4/1770#

    Range("B2").FormulaR1C1 = "Is Numeric? " & IsNumeric(Value)
End Sub
```

This would produce:

	A	B	C	D
1				
2		Is Numeric? False		
3				

Checking for Valid Date/Time

To find out whether an expression holds a valid date, a valid, or not, you can call the **IsDate()** function. Its syntax is:

```
Public Function IsDate(ByVal Expression As Variant) As Boolean
```

This function takes an argument as the expression to be evaluated. If the argument holds a valid date and/or time, the function returns True. Here is an example:

```
Sub Exercise()
    Dim DateHired As Variant

    DateHired = "9/16/2001"
    Range("B2").FormulaR1C1 = "Is 9/16/2001 a valid date? " & IsDate(DateHired)
End Sub
```

This would produce:

	A	B	C	D	E
1					
2		Is 9/16/2001 a valid date? True			
3					

If the value of the argument cannot be evaluated to a valid date or time, the function returns **False**. Here is an example:

```
Sub Exercise()
    Dim DateHired As Variant

    DateHired = "Who Knows?"
    Range("B2").FormulaR1C1 = "Is it a valid date? " & IsDate(DateHired)
End Sub
```

This would produce:

	A	B	C	D	E
1					
2		Is it a valid date? False			
3					

Checking for Nullity

After declaring a variable, you should initialize it with a valid value. Sometimes you will not. In some other cases, you may be using a variable without knowing with certainty whether it is holding a valid value. To assist you with checking whether a variable is currently holding a valid value, you can call the **IsNull()** function. Its syntax is:

```
Public Function IsNull(ByVal Expression As Variant) As Boolean
```

When calling this function, pass the name of a variable to it. If the variable is currently holding a valid value, this function would return **True**. Otherwise, it would return **False**.

Logical Operators

Introduction

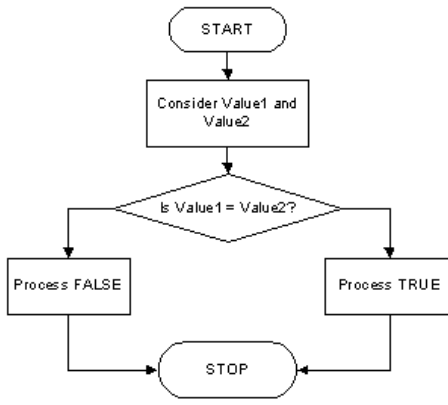
A comparison is an operation used to get the boolean result of two values one checked against the other. Such a comparison is performed between two values of the same type.

Equality

To compare two variables for equality, use the = operator. Its syntax is:

```
Value1 = Value2
```

The equality operation is used to find out whether two variables (or one variable and a constant) hold the same value. From our syntax, the value of Value1 would be compared with the value of Value2. If Value1 and Value2 hold the same value, the comparison produces a **True** result. If they are different, the comparison renders **false** or 0.



Here is an example:

```

Sub Exercise()
  Dim IsFullTime As Boolean

  Range("B2").FormulaR1C1 = "Is Employee Full Time? " & IsFullTime

  IsFullTime = True
  Range("B4").FormulaR1C1 = "Is Employee Full Time? " & IsFullTime
End Sub
  
```

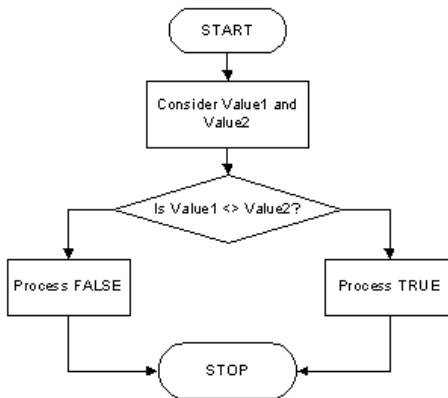
This would produce:

	A	B	C	D	E
1					
2		Is Employee Full Time? False			
3					
4		Is Employee Full Time? True			

Inequality <>

As opposed to checking for equality, you may instead want to know whether two values are different. The operator used to perform this comparison is <> and its formula is:

```
Variable1 <> Variable2
```



If the operands on both sides of the operator are the same, the comparison renders false. If both operands hold different values, then the comparison produces a true result. This also shows that the equality = and the inequality <> operators are opposite.

Here is an example:

```

Public Function IsDifferent(ByVal Value1 As Integer, _
  ByVal Value2 As Integer) As Boolean
  IsDifferent = Value1 <> Value2
End Function

Sub Exercise()
  Dim a%, b%
  Dim Result As Boolean

  a% = 12: b% = 48
  Result = IsDifferent(a%, b%)

  Range("B2").FormulaR1C1 = "The resulting comparison of 12 <> 48 is " & Result
End Sub
  
```

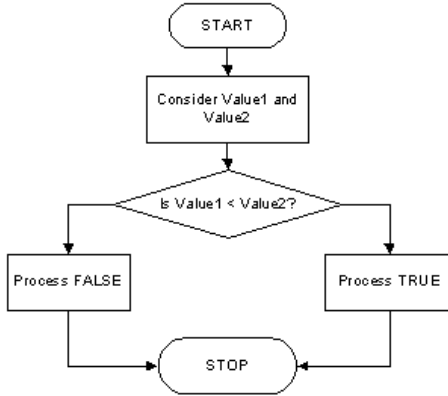
This would produce:

	A	B	C	D	E	F
1						
2		The resulting comparison of 12 <> 48 is True				
3						

A Lower Value <

Value1 < *Value2*

The value held by *Value1* is compared to that of *Value2*. As it would be done with other operations, the comparison can be made between two variables, as in *Variable1* < *Variable2*. If the value held by *Variable1* is lower than that of *Variable2*, the comparison produces a **True**.



Here is an example:

```

Sub Exercise()
  Dim PartTimeSalary, ContractorSalary As Double
  Dim IsLower As Boolean

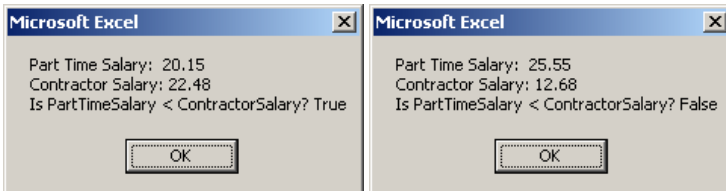
  PartTimeSalary = 20.15
  ContractorSalary = 22.48
  IsLower = PartTimeSalary < ContractorSalary

  MsgBox ("Part Time Salary: " & PartTimeSalary & vbCrLf & _
    "Contractor Salary: " & ContractorSalary & vbCrLf & _
    "Is PartTimeSalary < ContractorSalary? " & IsLower)

  PartTimeSalary = 25.55
  ContractorSalary = 12.68
  IsLower = PartTimeSalary < ContractorSalary

  MsgBox ("Part Time Salary: " & PartTimeSalary & vbCrLf & _
    "Contractor Salary: " & ContractorSalary & vbCrLf & _
    "Is PartTimeSalary < ContractorSalary? " & IsLower)
End Sub
  
```

This would produce:

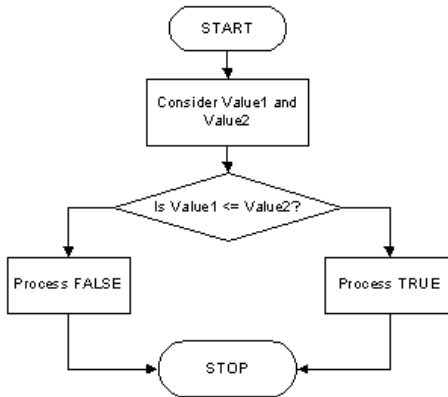


Equality and Lower Value <=

The previous two operations can be combined to compare two values. This allows you to know if two values are the same or if the first is less than the second. The operator used is <= and its syntax is:

Value1 <= *Value2*

The <= operation performs a comparison as any of the last two. If both *Value1* and *Value2* hold the same value, result is true or positive. If the left operand, in this case *Value1*, holds a value lower than the second operand, in this case *Value2*, the result is still true:



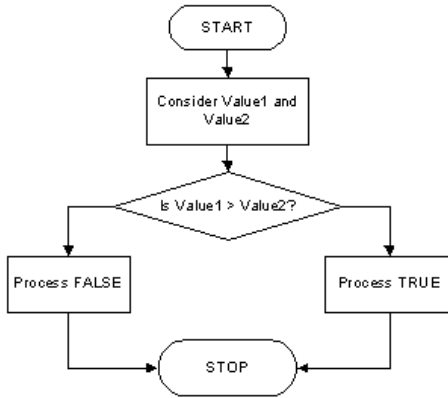
Greater Value >

When two values of the same type are distinct, one of them is usually higher than the other.

VBasic provides a logical operator that allows you to find out if one of two values is greater than the other. The operator used for this operation uses the > symbol. Its syntax is:

Value1 > Value2

Both operands, in this case Value1 and Value2, can be variables or the left operand can be a variable while the right operand is a constant. If the value on the left of the > operator is greater than the value on the right side or a constant, the comparison produces a **True** value. Otherwise, the comparison renders False or null:



Here is an example:

```

Sub Exercise()
  Dim PartTimeSalary, ContractorSalary As Double
  Dim IsLower As Boolean

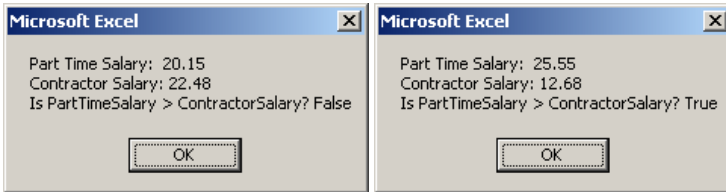
  PartTimeSalary = 20.15
  ContractorSalary = 22.48
  IsLower = PartTimeSalary > ContractorSalary

  MsgBox ("Part Time Salary: " & PartTimeSalary & vbCrLf & _
    "Contractor Salary: " & ContractorSalary & vbCrLf & _
    "Is PartTimeSalary > ContractorSalary? " & IsLower)

  PartTimeSalary = 25.55
  ContractorSalary = 12.68
  IsLower = PartTimeSalary > ContractorSalary

  MsgBox ("Part Time Salary: " & PartTimeSalary & vbCrLf & _
    "Contractor Salary: " & ContractorSalary & vbCrLf & _
    "Is PartTimeSalary > ContractorSalary? " & IsLower)
End Sub
  
```

This would produce:

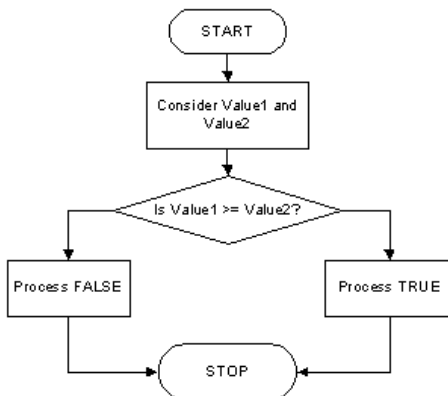


Greater or Equal Value >=

The greater than or the equality operators can be combined to produce an operator as follows: >=. This is the "greater than or equal to" operator. Its syntax is:

Value1 >= Value2

A comparison is performed on both operands: Value1 and Value2. If the value of Value1 and that of Value2 are the same, the comparison produces a **True** value. If the value of the left operand is greater than that of the right operand, the comparison still produces **True**. If the value of the left operand is strictly less than the value of the right operand, the comparison produces a **False** result:



Here is a summary table of the logical operators we have studied:

Operator	Meaning	Example	Opposite
=	Equality to	$a = b$	$< >$
$< >$	Not equal to	$12 < > 7$	=
<	Less than	$25 < 84$	$> =$
$< =$	Less than or equal to	$Cab < = Tab$	>
>	Greater than	$248 > 55$	$< =$
$> =$	Greater than or equal to	$Val1 > = Val2$	<

[Previous](#)

Copyright © 2008-2010 FunctionX, Inc.



Introduction to Conditional Statements

Checking Whether a Condition is True/False

Introduction

In some programming assignments, you must find out whether a given situation bears a valid value. This is done by checking a condition. To support this, the Visual Basic language provides a series of keywords and operators that can be combined to perform this checking. Checking a condition usually produces a True or a False result.

Once the condition has been checked, you can use the result (as True or False) to take action. Because there are different ways to check a condition, there are also different types of keywords to check different things. To use them, you must be aware of what each does or cannot do so you would select the right one.

❖ Practical Learning: Introducing Conditional Statements

1. Start Microsoft Excel
2. On the Ribbon, click Developer and, in the Code section, click Visual Basic
3. On the main menu, click Insert -> UserForm
4. Design the form as follows:

Control	Name	Caption
Label		First Name:
TextBox	txtFirstName	
Label		Last Name:
TextBox	txtLastName	
Label		Full Name:
TextBox	txtFullName	

5. Return to Microsoft Excel
6. Save the file with the name **Conditions1** as a Macro-Enabled Workbook
7. Return to Microsoft Visual Basic
8. Right-click the form and click View Code

If a Condition is True/False, Then What?

The **If...Then** statement examines the truthfulness of an expression. Structurally, its formula is:

```
If ConditionToCheck Then Statement
```

Therefore, the program examines a condition, in this case *ConditionToCheck*. This *ConditionToCheck* can be a simple expression or a combination of expressions. If the *ConditionToCheck* is true, then the program will execute the *Statement*.

There are two ways you can use the **If...Then** statement. If the conditional formula is short enough, you can write it on one line, like this:

```
If ConditionToCheck Then Statement
```

Here is an example:

```
Sub Exercise()
    Dim IsMarried As Boolean
    Dim TaxRate As Double
```

```

TaxRate = 33.0
MsgBox("Tax Rate: " & TaxRate & "%")

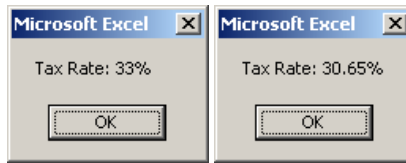
IsMarried = True

If IsMarried = True Then TaxRate = 30.65

MsgBox("Tax Rate: " & TaxRate & "%")
End Sub

```

This would produce:



If there are many statements to execute as a truthful result of the condition, you should write the statements on alternate lines. Of course, you can use this technique even if the condition you are examining is short. If you write the conditional statement in more than one line, you must end it with **End If** on its own line. The formula used is:

```

If ConditionToCheck Then
    Statement
End If

```

Here is an example:

```

Sub Exercise()
    Dim IsMarried As Boolean
    Dim TaxRate As Double

    TaxRate = 33#

    MsgBox ("Tax Rate: " & TaxRate & "%")

    IsMarried = True

    If IsMarried = True Then
        TaxRate = 30.65

        MsgBox ("Tax Rate: " & TaxRate & "%")
    End If
End Sub

```

❖ Practical Learning: Using If...Then

1. In the Variant combo box, select txtFirstName and change its Change event as follows:

```

Private Sub txtFirstName_Change()
    Dim FirstName As String
    Dim LastName As String
    Dim FullName As String

    FirstName = txtFirstName.Text
    LastName = txtLastName.Text

    FullName = LastName & ", " & FirstName

    txtFullName.Text = FullName
    If LastName = "" Then txtFullName.Text = FirstName
End Sub

```

2. In the Variant combo box, select txtLastName and change its **Change** event as follows:

```

Private Sub txtLastName_Change()
    Dim FirstName As String
    Dim LastName As String
    Dim FullName As String

    FirstName = txtFirstName.Text
    LastName = txtLastName.Text

    FullName = LastName & ", " & FirstName

    txtFullName.Text = FullName
    If LastName = "" Then txtFullName.Text = FirstName
End Sub

```

3. To test the form, on the main menu of Visual Basic, click Run -> Run Sub/UserForm
4. Click the top text box and type Julienne. Notice that only the first name displays in the Full Name text box

5. Press Tab
6. In the other text box, start typing Pal and notice that the Full Name text box is changing
7. Complete it with Palace
8. Close the form and return to Microsoft Visual Basic

Using the Default Value of a Boolean Expression

In the previous lesson, we saw that when you declare a Boolean variable, by default, it is initialized with the False value. Here is an example:

Module Exercise

```
Sub Exercise
    Dim IsMarried As Boolean

    MsgBox("Employee Is Married? " & IsMarried)

    Return 0
End Function
```

End Module

This would produce:



Based on this, if you want to check whether a newly declared and uninitialized Boolean variable is false, you can omit the = False expression applied to it. Here is an example:

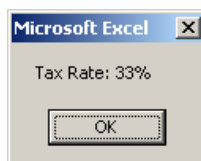
```
Sub Exercise()
    Dim IsMarried As Boolean
    Dim TaxRate As Double

    TaxRate = 33#

    If IsMarried Then TaxRate = 30.65

    MsgBox ("Tax Rate: " & TaxRate & "%")
End Sub
```

This would produce:



Notice that there is no = after the If IsMarried expression. In this case, the value of the variable is **False**. On the other hand, if you want to check whether the variable is **True**, make sure you include the = **True** expression. Overall, whenever in doubt, it is safer to always initialize your variable and it is safer to include the = **True** or = **False** expression when evaluating the variable:

```
Sub Exercise()
    Dim IsMarried As Boolean
    Dim TaxRate As Double

    TaxRate = 36.45 ' %

    IsMarried = True

    If IsMarried = False Then TaxRate = 33.15

    MsgBox ("Tax Rate: " & TaxRate & "%")
End Sub
```

In the previous lesson, we introduced some Boolean-based functions such as **IsNumeric** and **IsDate**. The default value of these functions is True. This means that when you call them, you can omit the = True expression.

What Else When a Condition is True/False?

The **If...Then** statement offers only one alternative: to act if the condition is true. Whenever you would like to apply an alternate expression in case the condition is false, you can use the **If...Then...Else** statement. The formula of this statement is:

```
If ConditionToCheck Then
    Statement1
Else
    Statement2
End If
```

When this section of code is executed, if the *ConditionToCheck* is true, then the first statement, *Statement1*, is executed. If the *ConditionToCheck* is false, the second statement, in this case *Statement2*, is executed.

Here is an example:

```
Sub Exercise()
    Dim MemberAge As Integer
    Dim MemberCategory As String

    MemberAge = 16

    If MemberAge <= 18 Then
        MemberCategory = "Teen"
    Else
        MemberCategory = "Adult"
    End If

    MsgBox ("Membership: " & MemberCategory)
End Sub
```

This would produce:



❖ Practical Learning: Using If...Then...Else

1. Change the codes of both events as follows:

```
Private Sub txtFirstName_Change()
    Dim FirstName As String
    Dim LastName As String
    Dim FullName As String

    FirstName = txtFirstName.Text
    LastName = txtLastName.Text

    If LastName = "" Then
        FullName = FirstName
    Else
        FullName = LastName & ", " & FirstName
    End If

    txtFullName.Text = FullName
End Sub

Private Sub txtLastName_Change()
    Dim FirstName As String
    Dim LastName As String
    Dim FullName As String

    FirstName = txtFirstName.Text
    LastName = txtLastName.Text

    If FirstName = "" Then
        FullName = LastName
    Else
        FullName = LastName & ", " & FirstName
    End If

    txtFullName.Text = FullName
End Sub
```

2. Press F5 to test the form
3. After using the form, close it and return to Visual Basic

To assist you with checking a condition and its alternative, the Visual Basic language provides a function named **IIf**. Its syntax is:

```
Public Function IIf( _
    ByVal Expression As Boolean, _
    ByVal TruePart As Variant, _
    ByVal FalsePart As Variant _
) As Variant
```

This function operates like an **If...Then...Else** condition. It takes three required arguments and returns a result of type **Variant**. This returned value will hold the result of the function.

The condition to check is passed as the first argument:

- If that condition is true, the function returns the value of the *TruePart* argument and the last argument is ignored
- If the condition is false, the first argument is ignored and the function returns the value of the second argument

As mentioned already, you can retrieve the value of the right argument and assign it to the result of the function. The expression we saw early can be written as follows:

```
Sub Exercise()
    Dim MemberAge As Integer
    Dim MemberCategory As String

    MemberAge = 16

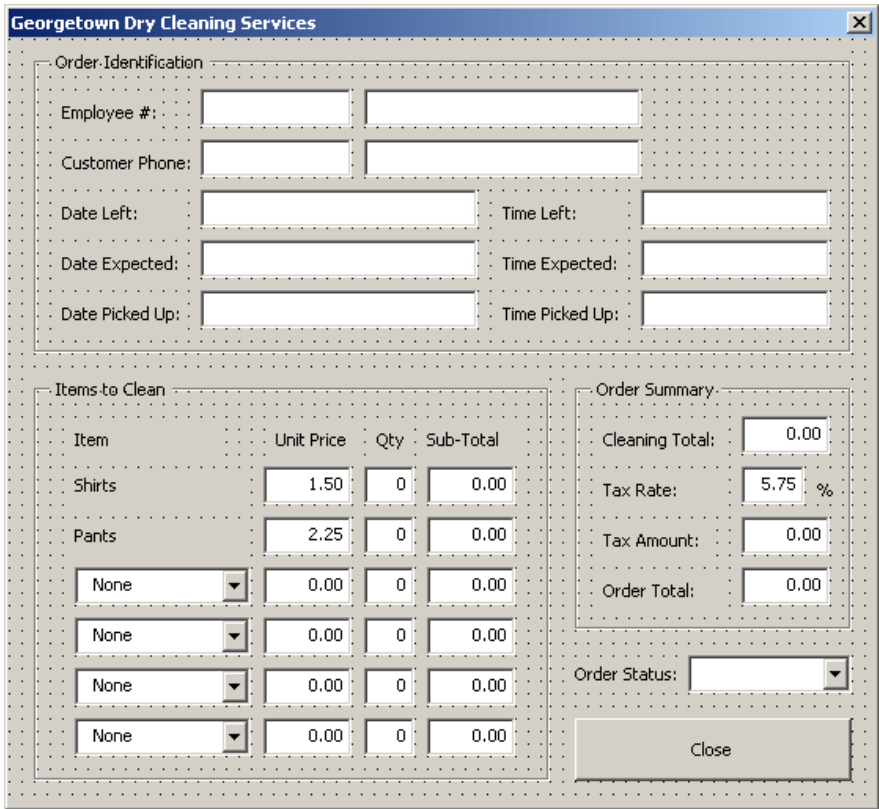
    MemberCategory = IIf(MemberAge <= 18, "Teen", "Adult")

    MsgBox ("Membership: " & MemberCategory)
End Sub
```

This would produce the same result we saw earlier.

❖ Practical Learning: Introducing Select Cases

1. From the resources that accompany these lessons, open the **gdcs1** (or **gdcs2**) workbook
2. To open Microsoft Visual Basic, on the [Ribbon](#), click Developer and, in the Code section, click Visual Basic:



3. Right-click the form and click View Code
4. Click under any code and type the following:

```
Private Sub CalucateOrder()
    Dim UnitPriceShirts As Double, UnitPricePants As Double
    Dim UnitPriceItem1 As Double, UnitPriceItem2 As Double
    Dim UnitPriceItem3 As Double, UnitPriceItem4 As Double
    Dim QuantityShirts As Integer, QuantityPants As Integer
    Dim QuantityItem1 As Integer, QuantityItem2 As Integer
```



```

Dim QuantityItem3 As Integer, QuantityItem4 As Integer
Dim SubTotalShirts As Double, SubTotalPants As Double
Dim SubTotalItem1 As Double, SubTotalItem2 As Double
Dim SubTotalItem3 As Double, SubTotalItem4 As Double
Dim CleaningTotal As Double, TaxRate As Double
Dim TaxAmount As Double, OrderTotal As Double

```

```

UnitPriceShirts = 0#: UnitPricePants = 0#
UnitPriceItem1 = 0#: UnitPriceItem2 = 0#
UnitPriceItem3 = 0#: UnitPriceItem4 = 0#

```

```

QuantityShirts = 0: QuantityPants = 0
QuantityItem1 = 0: QuantityItem2 = 0
QuantityItem3 = 0: QuantityItem4 = 0

```

```
TaxRate = 0
```

```

UnitPriceShirts = IIf(IsNumeric(txtUnitPriceShirts), _
    CDbI(txtUnitPriceShirts), 0)

```

```

UnitPricePants = IIf(IsNumeric(txtUnitPricePants), _
    CDbI(txtUnitPricePants), 0)

```

```

UnitPriceItem1 = IIf(IsNumeric(txtUnitPriceItem1), _
    CDbI(txtUnitPriceItem1), 0)

```

```

UnitPriceItem2 = IIf(IsNumeric(txtUnitPriceShirts), _
    CDbI(txtUnitPriceItem2), 0)

```

```

UnitPriceItem3 = IIf(IsNumeric(txtUnitPriceShirts), _
    CDbI(txtUnitPriceItem3), 0)

```

```

UnitPriceItem4 = IIf(IsNumeric(txtUnitPriceShirts), _
    CDbI(txtUnitPriceItem4), 0)

```

```

QuantityShirts = IIf(IsNumeric(txtUnitPriceShirts), _
    CInt(txtQuantityShirts), 0)

```

```

QuantityPants = IIf(IsNumeric(txtQuantityPants), _
    CInt(txtQuantityPants), 0)

```

```

QuantityItem1 = IIf(IsNumeric(txtQuantityItem1), _
    Int(txtQuantityItem1), 0)

```

```

QuantityItem2 = IIf(IsNumeric(txtQuantityItem2), _
    CInt(txtQuantityItem2), 0)

```

```

QuantityItem3 = IIf(IsNumeric(txtQuantityItem3), _
    CInt(txtQuantityItem3), 0)

```

```

QuantityItem4 = IIf(IsNumeric(txtQuantityItem4), _
    CInt(txtQuantityItem4), 0)

```

```

TaxRate = IIf(IsNumeric(txtTaxRate), _
    CDbI(txtTaxRate), 0)

```

```

SubTotalShirts = UnitPriceShirts * QuantityShirts
SubTotalPants = UnitPricePants * QuantityPants
SubTotalItem1 = UnitPriceItem1 * QuantityItem1
SubTotalItem2 = UnitPriceItem2 * QuantityItem2
SubTotalItem3 = UnitPriceItem3 * QuantityItem3
SubTotalItem4 = UnitPriceItem4 * QuantityItem4

```

```

txtSubTotalShirts = FormatNumber(SubTotalShirts)
txtSubTotalPants = FormatNumber(SubTotalPants)
txtSubTotalItem1 = FormatNumber(SubTotalItem1)
txtSubTotalItem2 = FormatNumber(SubTotalItem2)
txtSubTotalItem3 = FormatNumber(SubTotalItem3)
txtSubTotalItem4 = FormatNumber(SubTotalItem4)

```

```

CleaningTotal = SubTotalShirts + SubTotalPants + _
    SubTotalItem1 + SubTotalItem2 + _
    SubTotalItem3 + SubTotalItem4

```

```

TaxAmount = CleaningTotal * TaxRate / 100
OrderTotal = CleaningTotal + TaxAmount

```

```

txtCleaningTotal = FormatNumber(CleaningTotal)
txtTaxAmount = FormatNumber(TaxAmount)
txtOrderTotal = FormatNumber(OrderTotal)

```

```
End Sub
```

```

Private Sub txtUnitPriceShirts_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    CalucateOrder
End Sub

```

```

Private Sub txtQuantityShirts_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    CalucateOrder
End Sub

```

```

Private Sub txtUnitPricePants_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    CalucateOrder

```

```

End Sub

Private Sub txtQuantityPants_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    CalucateOrder
End Sub

Private Sub txtUnitPriceItem1_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    CalucateOrder
End Sub

Private Sub txtQuantityItem1_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    CalucateOrder
End Sub

Private Sub txtUnitPriceItem2_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    CalucateOrder
End Sub

Private Sub txtQuantityItem2_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    CalucateOrder
End Sub

Private Sub txtUnitPriceItem3_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    CalucateOrder
End Sub

Private Sub txtQuantityItem3_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    CalucateOrder
End Sub

Private Sub txtUnitPriceItem4_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    CalucateOrder
End Sub

Private Sub txtQuantityItem4_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    CalucateOrder
End Sub

Private Sub txtTaxRate_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    CalucateOrder
End Sub

```

5. Close Microsoft Visual Basic
6. Save the workbook

Choosing a Value

We have learned how to check whether a condition is True or False and take an action. Here is an example:

```

Sub Exercise()
    Dim Status As Integer, EmploymentStatus As String

    Status = 1
    EmploymentStatus = "Unknown"

    If Status = 1 Then
        EmploymentStatus = "Full Time"
    End If

    MsgBox ("Employment Status: " & EmploymentStatus)
End Sub

```

To provide an alternative to this operation, the Visual Basic language provides a function named **Choose**. Its syntax is:

```

Public Function Choose( _
    ByVal Index As Double, _
    ByVal ParamArray Choice() As Variant _
) As Variant

```

This function takes two required arguments. The first argument is equivalent to the *ConditionToCheck* of our **If...Then** formula. For the **Choose()** function, this first argument must be a number. This is the value against which the second argument will be compared. Before calling the function, you must know the value of the first argument. To take care of this, you can first declare a variable and initialize it with the desired value. Here is an example:

```

Sub Exercise()
    Dim Status As Byte, EmploymentStatus As String

    Status = 1

    EmploymentStatus = Choose(Status, ...)

    MsgBox ("Employment Status: " & EmploymentStatus)
End Sub

```

The second argument can be the *Statement* of our formula. Here is an example:

We will see in the next sections that the second argument is actually a list of values and each value has a specific position referred to as its index. To use the function in an **If...Then** scenario, you pass only one value as the second argument. This value/argument has an index of 1. When the **Choose()** function is called in an **If...Then** implementation, if the first argument holds a value of 1, the second argument is validated.

When the **Choose()** function has been called, it returns a value of type **Variant**. You can retrieve that value, [store](#) it in a variable and use it as you see fit. Here is an example:

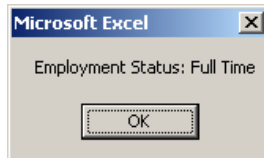
```
Sub Exercise()
    Dim Status As Byte, EmploymentStatus As String

    Status = 1

    EmploymentStatus = Choose(Status, "Full Time")

    MsgBox ("Employment Status: " & EmploymentStatus)
End Sub
```

This would produce:



In some cases, the **Choose()** function can produce a null result. Consider the same program we used earlier but with a different value:

```
Module Exercise

    Sub Exercise
        Dim Status As Integer, EmploymentStatus As String

        Status = 2

        EmploymentStatus = Choose(Status, "Full Time")

        MsgBox(EmploymentStatus)

        Return 0
    End Function
End Module
```

This would produce an error because there is no value in index 2 after the Status variable has been initialized with 2. To use this function as an alternative to the **If...Then...Else** operation, you can pass two values for the second argument. The second argument is actually passed as a list of values. Each value has a specific position as its index. To use the function in an **If...Then...Else** implementation, pass two values for the second argument. Here is an example:

```
Choose(Status, "Full Time", "Part Time")
```

The second argument to the function, which is the first value of the *Choose* argument, has an index of 1. The third argument to the function, which is the second value of the *Choose* argument, has an index of 2.

When the **Choose()** function is called, if the first argument has a value of 1, then the second argument is validated. If the first argument has a value of 2, then the third argument is validated. As mentioned already, you can retrieve the returned value of the function and use it however you want. Here is an example:

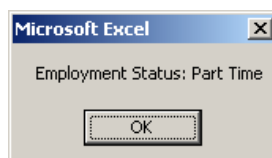
```
Sub Exercise()
    Dim Status As Integer, EmploymentStatus As String

    Status = 2

    EmploymentStatus = Choose(Status, "Full Time", "Part Time")

    MsgBox ("Employment Status: " & EmploymentStatus)
End Sub
```

This would produce:



Switching to a Value

As another alternative to an **If...Then** condition, the Visual Basic language provides a function named **Switch**. Its syntax is:

```
Public Function Switch( _
    ByVal ParamArray VarExpr() As Variant _
) As Variant
```

This function takes one required argument. To use it in an **If...Then** scenario, pass the argument as follows:

```
Switch(ConditionToCheck, Statement)
```

In the *ConditionToCheck* placeholder, pass a Boolean expression that can be evaluated to **True** or **False**. If that condition is true, the second argument would be executed.

When the **Switch()** function has been called, it produces a value of type **Variant** (such as a string) that you can use as you see fit. For example, you can store it in a variable. Here is an example:

```
Sub Exercise()
    Dim Status As Integer, EmploymentStatus As String

    Status = 1
    EmploymentStatus = "Unknown"

    EmploymentStatus = Switch(Status = 1, "Full Time")

    MsgBox ("Employment Status: " & EmploymentStatus)
End Sub
```

In this example, we used a number as argument. You can also use another type of value, such as an enumeration. Here is an example:

```
Private Enum EmploymentStatus
    FullTime
    PartTime
    Contractor
    Seasonal
    Unknown
End Enum

Sub Exercise()
    Dim Status As EmploymentStatus
    Dim Result As String

    Status = EmploymentStatus.FullTime
    Result = "Unknown"

    Result = Switch(Status = EmploymentStatus.FullTime, "Full Time")

    MsgBox ("Employment Status: " & Result)
End Sub
```

When using the **Switch** function, if you call it with a value that is not checked by the first argument, the function produces an error. To apply this function to an **If...Then...Else** scenario, you can call it using the following formula:

```
Switch(Condition1ToCheck, Statement1, Condition2ToCheck, Statement2)
```

In the *Condition1ToCheck* placeholder, pass a Boolean expression that can be evaluated to **True** or **False**. If that condition is true, the second argument would be executed. To provide an alternative to the first condition, pass another condition as *Condition2ToCheck*. If the *Condition2ToCheck* is true, then *Statement2* would be executed. Once gain, remember that you can get the value returned by the Switch function and use it. Here is an example:

```
Private Enum EmploymentStatus
    FullTime
    PartTime
    Contractor
    Seasonal
    Unknown
End Enum

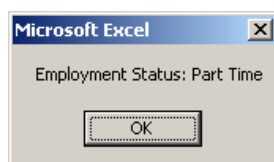
Sub Exercise()
    Dim Status As EmploymentStatus
    Dim Result As String

    Status = EmploymentStatus.PartTime
    Result = "Unknown"

    Result = Switch(Status = EmploymentStatus.FullTime, "Full Time", _
        Status = EmploymentStatus.PartTime, "Part Time")

    MsgBox ("Employment Status: " & Result)
End Sub
```

This would produce:





Functional Conditions

Alternatives to a Condition Being True/False?

The If...Then...ElseIf Condition

The **If...Then...ElseIf** statement acts like the **If...Then...Else** expression, except that it offers as many choices as necessary. The formula is:

```
If Condition1 Then
    Statement1
ElseIf Condition2 Then
    Statement2
ElseIf Conditionk Then
    Statementk
End If
```

The program will first examine *Condition1*. If *Condition1* is true, the program will execute *Statement1* and stop examining conditions. If *Condition1* is false, the program will examine *Condition2* and act accordingly. Whenever a condition is false, the program will continue examining the conditions until it finds one that is true. Once a true condition has been found and its statement executed, the program will terminate the conditional examination at **End If**. Here is an example:

```
Sub Exercise()
    Dim MemberAge As Byte

    MemberAge = 32

    If MemberAge <= 18 Then
        MsgBox ("Membership: " & "Teen")
    ElseIf MemberAge < 55 Then
        MsgBox ("Membership: " & "Adult")
    End If
End Sub
```

This would produce:



❖ Practical Learning: Introducing Data Entry

1. Start Microsoft Excel and, on the [Ribbon](#), click Developer
2. In the Code section, click Visual Basic
3. To add a new form, on the Standard toolbar, click the Insert UserForm button
4. Design the form as follows:

Control	Name	Caption/Text	Other Properties
Label		Number of CDs:	
TextBox	txtQuantity	0	TextAlign: 3 - frmTextAlignRight
CommandButton	cmdEvaluate	Evaluate	
Frame		Based on the Specified Quantity	
Label		Each CD will cost:	
TextBox	txtUnitPrice	0.00	TextAlign: 3 - frmTextAlignRight
Label		And the total price is:	

TextBox	txtTotalPrice	0.00	TextAlign: 3 - frmTextAlignRight
---------	---------------	------	----------------------------------

What If No Alternative is Valid?

There is still a possibility that none of the stated conditions be true. In this case, you should provide a "catch all" condition. This is done with a last **Else** section. The **Else** section must be the last in the list of conditions and would act if none of the primary conditions is true. The formula to use would be:

```
If Condition1 Then
    Statement1
ElseIf Condition2 Then
    Statement2
ElseIf Conditionk Then
    Statementk
Else
    CatchAllStatement
End If
```

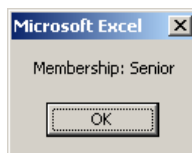
Here is an example:

```
Sub Exercise()
    Dim MemberAge As Byte

    MemberAge = 65

    If MemberAge <= 18 Then
        MsgBox ("Membership: " & "Teen")
    ElseIf MemberAge < 55 Then
        MsgBox ("Membership: " & "Adult")
    Else
        MsgBox ("Membership: " & "Senior")
    End If
End Sub
```

This would produce:



❖ Practical Learning: Using If...Then...ElseIf

1. Double-click the Evaluate button and implement its Click event as follows:

```
Private Sub cmdEvaluate_Click()
    Dim Quantity As Integer
    Dim UnitPrice As Currency
    Dim TotalPrice As Currency

    Quantity = CInt(txtQuantity.Text)

    ' The price of one CD will depend on the number ordered
    ' The more the customer orders, the lower value each
    If Quantity < 20 Then
        UnitPrice = 20
    ElseIf Quantity < 50 Then
        UnitPrice = 15
    ElseIf Quantity < 100 Then
        UnitPrice = 12
    ElseIf Quantity < 500 Then
        UnitPrice = 8
    Else
        UnitPrice = 5
    End If

    TotalPrice = Quantity * UnitPrice

    txtUnitPrice.Text = CStr(UnitPrice)
    txtTotalPrice.Text = CStr(TotalPrice)
End Sub
```

2. Press F5 to test the form
3. Perform the calculations with different quantities. For example, in the top text box, type **1250** and click Evaluate

4. After testing various quantities, close the form

Conditional Statements and Functions

Introduction

As introduced in previous lessons, we know that a function is used to perform a specific assignment and produce a result. Here is an example:

```
Private Function SetMembershipLevel$()
    Dim MemberAge%

    MemberAge% = InputBox("Enter the Member's Age")

    SetMembershipLevel$ = ""
End Function
```

When performing its assignment, a function can encounter different situations, some of which would need to be checked for truthfulness or negation. This means that conditional statements can assist a procedure with its assignment.

❖ Practical Learning: Introducing Condition Functions

1. Start another workbook
2. In cell B2, type **Bethesda Car Rental**
3. In cell B3, type **Order Processing**
4. In cell B4, type **Processed by:**
5. In cell B5, type **Processed for:**
6. In cell B6, type **Car Selected:**
7. In cell B7, type **Tag #:**
8. Enlarge column B so that Processed by: can fit in the allocated width
9. Right-align cell B7

	A1			
	A	B	C	D
1				
2		Bethesda Car Rental		
3		Order Processing		
4		Processed by:		
5		Processed for:		
6		Car Selected:		
7		Tag #:		
8				

Conditional Returns

A function is meant to return a value. Sometimes, it will perform some tasks whose results would lead to different results. A function can return only one value (we saw that, by passing arguments by reference, you can make a procedure return more than one value) but you can make it render a result depending on a particular behavior. If a function is requesting an answer from the user, since the user can provide different answers, you can treat each result differently. Consider the following function:

```
Private Function SetMembershipLevel$()
    Dim MemberAge%

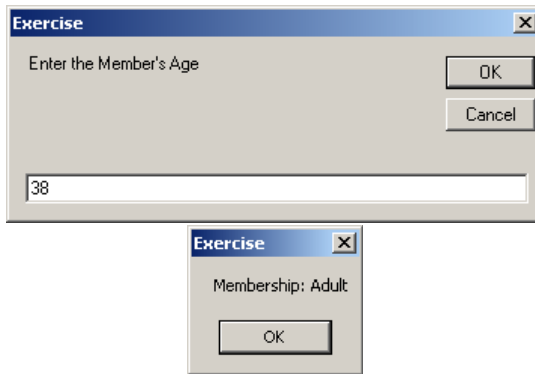
    MemberAge% = InputBox("Enter the Member's Age")

    If MemberAge% < 18 Then
        SetMembershipLevel$ = "Teen"
    ElseIf MemberAge% < 55 Then
        SetMembershipLevel$ = "Adult"
    End If
End Function

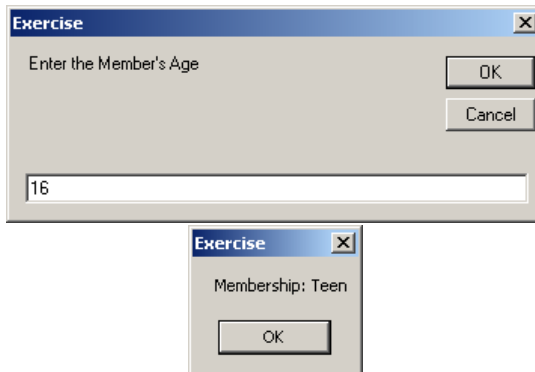
Sub Exercise()
    Dim Membership$

    MsgBox ("Membership: " & Membership$)
```

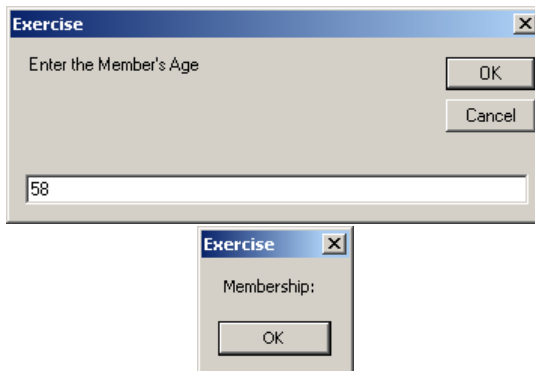

At first glance, this function looks fine. The user is asked to provide a number. If the user enters a number less than 18 (excluded), the function returns Teen. Here is an example of running the program:



If the user provides a number between 18 (included) and 55, the function returns the Adult. Here is another example of running the program:



What if there is an answer that does not fit those we are expecting? The values that we have returned in the function conform only to the conditional statements and not to the function. Remember that in *If Condition Statement*, the *Statement* executes only if the *Condition* is true. Here is what will happen. If the user enters a number higher than 55 (excluded), the function will not execute any of the returned statements. This means that the execution will reach the **End Function** line without encountering a return value. This also indicates to the compiler that you wrote a function that is supposed to return a value, but by the end of the method, it didn't return a value. Here is another example of running the program:



To solve this problem, you have various alternatives. If the function uses an If...Then condition, you can create an Else section that embraces any value other than those validated previously. Here is an example:

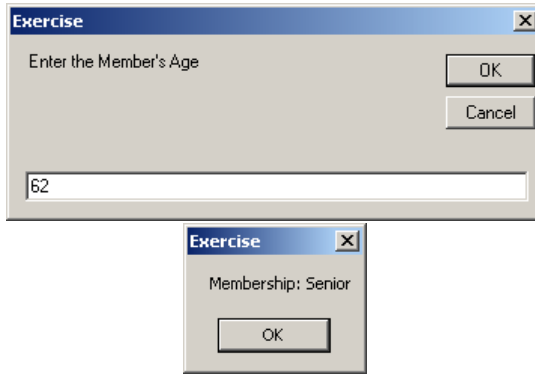
```
Private Function SetMembershipLevel$()
    Dim MemberAge%

    MemberAge% = InputBox("Enter the Member's Age")

    If MemberAge% < 18 Then
        SetMembershipLevel$ = "Teen"
    ElseIf MemberAge% < 55 Then
        SetMembershipLevel$ = "Adult"
    Else
        SetMembershipLevel$ = "Senior"
    End If
End Function

Sub Exercise()
    Dim Membership$

    Membership$ = SetMembershipLevel$()
    MsgBox ("Membership: " & Membership$)
End Sub
```



An alternative is to provide a last return value just before the **End Function** line. In this case, if the execution reaches the end of the function, it would still return something but you would know what it returns. This would be done as follows:

```
Private Function SetMembershipLevel$()
    Dim MemberAge%

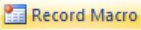
    MemberAge% = InputBox("Enter the Member's Age")

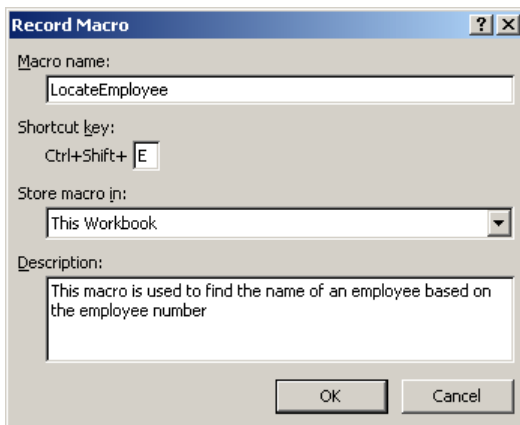
    If MemberAge% < 18 Then
        SetMembershipLevel$ = "Teen"
    ElseIf MemberAge% < 55 Then
        SetMembershipLevel$ = "Adult"
    End If

    SetMembershipLevel$ = "Senior"
End Function
```

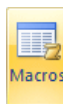
If the function uses an If condition, both implementations would produce the same result.

❖ Practical Learning: Using a Conditional Statement

1. On the Ribbon, click Developer
2. In the Code section, click Record Macro 
3. Set the Macro Name to **LocateEmployee**
4. In the Shortcut Key text box, type **E** to get Ctrl + Shift + E



5. Click OK
6. On the Ribbon, click Stop Recording



7. In the Code section of the Ribbon, click Macros
8. In the Macro dialog box, make sure LocateEmployee is selected and click Edit
9. Change the code as follows:

```
Private Function GetEmployeeName(ByVal EmplNbr As Long) As String
    Dim Name As String

    If EmplNbr = 22804 Then
        Name = "Helene Mukoko"
    ElseIf EmplNbr = 92746 Then
        Name = "Raymond Kouma"
    ElseIf EmplNbr = 54080 Then
        Name = "Henry Larson"
    ElseIf EmplNbr = 86285 Then
        Name = "Gertrude Monay"
    Else
        Name = ""
    End If
End Function
```

```

End If

GetEmployeeName = Name
End Function

Public Sub LocateEmployee()
'
' Macro Name: LocateEmployee
' This macro is used to find the name of an employee
' based on the employee number
'
' Keyboard Shortcut: Ctrl+Shift+E
'
    Dim EmployeeNumber As Long, EmployeeName As String

    If IsEmpty(Range("C4")) Then
        MsgBox "You must enter the employee number in cell C4"
        Range("D4").FormulaR1C1 = ""
        EmployeeNumber = 0
    Else
        EmployeeNumber = CLng(Range("C4"))
    End If

    EmployeeName = GetEmployeeName(EmployeeNumber)
    Range("D4").FormulaR1C1 = EmployeeName
End Sub

```

10. Return to Microsoft Excel
11. In cell C4, type **54080** and press Enter
12. Press Ctrl + Shift + E to see the result

	A	B	C	D	E
1					
2		Bethesda Car Rental			
3		Order Processing			
4		Processed by:	54080	Henry Larson	
5		Processed for:	<input type="text"/>		
6		Car Selected			
7		Tag #:			

If-Condition Built-In Functions

Using the Immediate If Function

The **IIf()** function can also be used in place of an **If...Then...ElseIf** scenario. When the function is called, the *Expression* is checked. As we saw already, if the expression is true, the function returns the value of the *TruePart* argument and ignores the last argument. To use this function as an alternative to **If...Then...ElseIf** statement, if the expression is false, instead of immediately returning the value of the *FalsePart* argument, you can translate that part into a new **IIf** function. The pseudo-syntax would become:

```

Public Function IIf( _
    ByVal Expression As Boolean, _
    ByVal TruePart As Object, _
        Public Function IIf( _
            ByVal Expression As Boolean, _
            ByVal TruePart As Object, _
            ByVal FalsePart As Object _
        ) As Object
) As Object

```

In this case, if the expression is false, the function returns the *TruePart* and stops. If the expression is false, the compiler accesses the internal **IIf** function and applies the same scenario. Here is example:

```

Sub Exercise()
    Dim MemberAge As Byte
    Dim MembershipCategory As String

    MemberAge = 74

    MembershipCategory = _
        IIf(MemberAge <= 18, "Teen", IIf(MemberAge < 55, "Adult", "Senior"))

    MsgBox ("Membership: " & MembershipCategory)
End Sub

```

We saw that in an **If...Then...ElseIf** statement you can add as many **ElseIf** conditions as you want. In the same, you can call as many **IIf** functions in the subsequent *FalsePart* sections as you judge necessary:

```

Public Function IIf( _
    ByVal Expression As Boolean, _
    ByVal TruePart As Object, _
        Public Function IIf( _
            ByVal Expression As Boolean, _

```

```

    ByVal TruePart As Object, _
    Public Function IIf( _
        ByVal Expression As Boolean, _
        ByVal TruePart As Object, _
        Public Function IIf( _
            ByVal Expression As Boolean, _
            ByVal TruePart As Object, _
            ByVal FalsePart As Object _
        ) As Object
    ) As Object
) As Object
) As Object

```

Choose an Alternate Value

As we have seen so far, the **Choose** function takes a list of arguments. To use it as an alternative to the **If...Then...Elseif...Elseif** condition, you can pass as many values as you judge necessary for the second argument. The index of the first member of the second argument would be 1. The index of the second member of the second argument would be 2, and so on. When the function is called, it would first get the value of the first argument, then it would check the indexes of the available members of the second argument. The member whose index matches the first argument would be executed. Here is an example:

```

Sub Exercise()
    Dim Status As Byte, EmploymentStatus As String

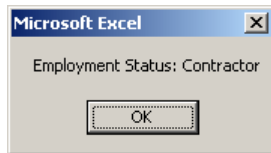
    Status = 3

    EmploymentStatus = Choose(Status, _
        "Full Time", _
        "Part Time", _
        "Contractor", _
        "Seasonal")

    MsgBox ("Employment Status: " & EmploymentStatus)
End Sub

```

This would produce:



So far, we have used only strings for the values of the second argument of the **Choose()** function. In reality, the values of the second argument can be almost anything. One value can be a constant. Another value can be a string. Yet another value can come from calling a function. Here is an example:

```

Private Function ShowContractors$()
    ShowContractors$ = "==" List of Contractors ==" & vbCrLf & _
        "Martin Samson" & vbCrLf & _
        "Geneviève Lam" & vbCrLf & _
        "Frank Viel" & vbCrLf & _
        "Henry Rickson" & vbCrLf & _
        "Samuel Lott"
End Function

Sub Exercise()
    Dim Status As Byte, Result$

    Status = 3

    Result = Choose(Status, _
        "Employment Status: Full Time", _
        "Employment Status: Part Time", _
        ShowContractors, _
        "Seasonal Employment")

    MsgBox (Result)
End Sub

```

This would produce:



The values of the second argument can even be of different types.

Switching to an Alternate Value

The **Switch()** function is a prime alternative to the **If...Then...Elseif...Elseif** condition. The argument to this function is passed as a list of values. As seen previously, each value is passed as a combination of two values:

ConditionXToCheck, StatementX

As the function is accessed, the compiler checks each condition. If a condition X is true, its statement is executed. If a condition Y is false, the compiler skips it. You can provide as many of these combinations as you want. Here is an example:

```
Private Enum EmploymentStatus
    FullTime
    PartTime
    Contractor
    Seasonal
End Enum

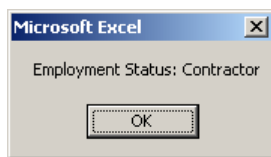
Sub Exercise()
    Dim Status As EmploymentStatus
    Dim Result As String

    Status = EmploymentStatus.Contractor
    Result = "Unknown"

    Result = Switch(Status = EmploymentStatus.FullTime, "Full Time", _
        Status = EmploymentStatus.PartTime, "Part Time", _
        Status = EmploymentStatus.Contractor, "Contractor", _
        Status = EmploymentStatus.Seasonal, "Seasonal")

    MsgBox ("Employment Status: " & Result)
End Sub
```

This would produce:



In a true **If...Then...Elseif...Elseif** condition, we saw that there is a possibility that none of the conditions would fit, in which case you can add a last **Else** statement. The **Switch()** function also supports this situation if you are using a number, a character, or a string. To provide this last alternative, instead of a *ConditionXToCheck* expression, enter **True**, and include the necessary statement. Here is an example:

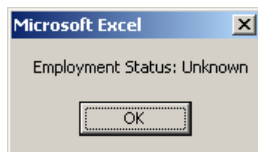
```
Sub Exercise()
    Dim Status As Byte
    Dim Result As String

    Status = 12

    Result = Switch(Status = 1, "Full Time", _
        Status = 2, "Part Time", _
        Status = 3, "Contractor", _
        Status = 4, "Seasonal", _
        True, "Unknown")

    MsgBox ("Employment Status: " & Result)
End Sub
```

This would produce:



Remember that you can also use True with a character. Here is an example:

```
Sub Exercise()
    Dim Gender As String
    Dim Result As String

    Gender = "H"


    Result = Switch(Gender = "f", "Female", _
        Gender = "F", "Female", _
        Gender = "m", "Male", _
        Gender = "M", "Male", _
        True, "Unknown")

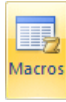
    MsgBox ("Gender: " & Result)
End Sub
```

This would produce:



❖ Practical Learning: Using the Switch() Function

1. In the Code section, click Record Macro 
2. Set the Macro Name to **SelectCar**
3. In the Shortcut Key text box, type **S** to get Ctrl + Shift + S, and click OK
4. On the Ribbon, click Stop Recording



5. In the Code section of the Ribbon, click Macros
6. In the Macro dialog box, make sure FindEmployee is selected and click Edit
7. To use the **Switch()** function, change the document as follows:

```
Public Sub SelectCar()
'
' Macro Name: SelectCar
' This macro is used to locate a car given its tag number
' Keyboard Shortcut: Ctrl+Shift+S
'
    Dim TagNumber As String, CarSelected As String

    If IsEmpty(Range("C7")) Then
        MsgBox "You must enter the tag number of the car the customer will rent"
        TagNumber = 0
    Else
        TagNumber = Range("C7")

        CarSelected = Switch(TagNumber = "297419", "BMW 335i", _
            TagNumber = "485M270", "Chevrolet Avalanche", _
            TagNumber = "247597", "Honda Accord LX", _
            TagNumber = "924095", "Mazda Miata", _
            TagNumber = "772475", "Chevrolet Aveo", _
            TagNumber = "M931429", "Ford E150XL", _
            TagNumber = "240759", "Buick Lacrosse", _
            True, "Unidentified Car")

        Range("D7").FormulaR1C1 = CarSelected
    End If
End Sub
```

8. Return to Microsoft Excel
9. In cell C7, type **924095** and press Enter
10. Press Ctrl + Shift + S to see the result

	A1				
	A	B	C	D	E
1					
2		Bethesda Car Rental			
3		Order Processing			
4		Processed by:	54080	Henry Larson	
5		Processed for:			
6		Car Selected			
7		Tag #:	924095	Mazda Miata	
8					



Conditional Selections

The Select...Case Statement

Introduction

If you have a large number of conditions to examine, the **If...Then...Else** statement will go through each one of them. The Visual Basic language offers the alternative of jumping to the statement that applies to the state of a condition. This is referred to as a select case condition and it uses the keywords **Select** and **Case**.

The formula of the **Select Case** statement is:

```
Select Case Expression
  Case Expression1
    Statement1
  Case Expression2
    Statement2
  Case Expression_X
    Statement_X
End Select
```

The statement starts with **Select Case** and ends with **End Select**. On the right side of **Select Case**, enter a value, the *Expression* factor, that will be used as a tag. The value of *Expression* can be a Boolean value (a **Boolean** type), a character or a string (a **String** type), a natural number (a **Byte**, an **Integer**, or a **Long** type), a decimal number (a **Single** or a **Double** type), a date or time value (a **Date** type), an enumeration (an **Enum** type), or else (a **Variant** type).

Inside the **Select Case** and the **End Select** lines, you provide one or more sections that each contains a **Case** keyword followed by a value. The value on the right side of a **Case**, *Expression1*, *Expression2*, or *Expression_X*, must be the same type as the value of *Expression* or it can be implied from it. After the case and its expression, you can write a statement.

When this section of code is accessed, the value of *Expression* is considered. Then the value of *Expression* is compared to each *Expression_X* of each case:

- If the value of *Expression1* is equal to that of *Expression*, then *Statement1* is executed. If the value of *Expression1* is not equal to that of *Expression*, then the interpreter moves to *Expression2*
- If the value of *Expression2* is equal to that of *Expression*, then *Statement2* is executed
- This will continue down to the last *Expression_X*

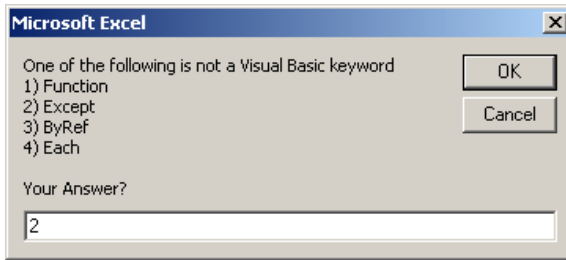
Here is an example:

```
Sub Exercise
  Dim Answer As Byte

  Answer = CByte(InputBox( _
    "One of the following is not a Visual Basic keyword" & vbCrLf & _
    "1) Function" & vbCrLf & _
    "2) Except" & vbCrLf & _
    "3) ByRef" & vbCrLf & _
    "4) Each" & vbCrLf & vbCrLf & _
    "Your Answer? "))

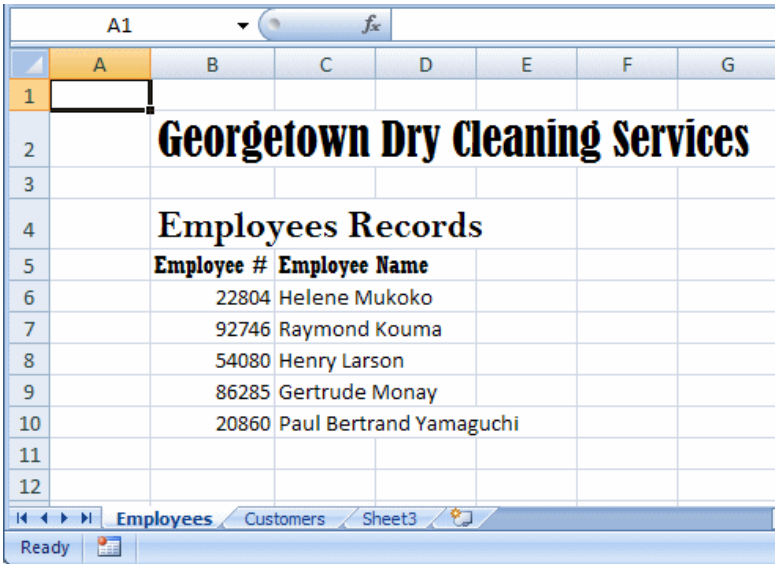
  Select Case Answer
    Case 1
      MsgBox("Wrong: Function is a Visual Basic keyword." & vbCrLf & _
        "It is used to create a procedure of a function type")
    Case 2
      MsgBox("Correct: Except is not a keyword in " & vbCrLf & _
        "Visual Basic but __except is a C++ " & vbCrLf & _
        "keyword used in Exception Handling")
    Case 3
      MsgBox("Wrong: ByRef is a Visual Basic keyword used " & vbCrLf & _
        "to pass an argument by reference to a procedure")
    Case 4
      MsgBox("Wrong: The ""Each"" keyword is used in " & vbCrLf & _
        "Visual Basic in a type of looping " & vbCrLf & _
        "used to ""scan"" a list of item.")
  End Select
End Sub
```

Here is an example of running the program:



❖ Practical Learning: Introducing Select Cases

1. Start Microsoft Excel
2. From the resources that accompany these lessons, open the **gdcs1** (or gdcs2) workbook you
3. Change the names of the first and the second worksheets to Employees and Customers respectively
4. Add a few records in the Employees worksheet



5. Add a few records in the Customers worksheet



6. Save the workbook

7. To open Microsoft Visual Basic, on the [Ribbon](#), click Developer and, in the Code section, click Visual Basic:

8. Right-click the Time Left text box (the text box on the right side of Time Left) and click View Code
9. In the Objects combo box, make sure txtTimeLeft is selected. In the Procedure combo box, select Exit and implement the event as follows:

```
Private Sub txtTimeLeft_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    Dim DateLeft As Date, TimeLeft As Date
    Dim DateExpected As Date, TimeExpected As Date

    If IsDate(txtTimeLeft) Then
        TimeLeft = CDate(txtTimeLeft)
    Else
        MsgBox "The value you entered is not a valid time"
        txtTimeLeft = Time
    End If
End Sub
```

What Case Else?

The above code supposes that one of the cases will match the value of the *Expression* factor. This is not always so. If you anticipate that there could be no match between the *Expression* and one of the *Expressions*, you can use a **Case Else** statement at the end of the list. The statement would then look like this:

```
Select Case Expression
Case Expression1
    Statement1
Case Expression2
    Statement2
Case Expressionk
    Statementk
Case Else
    Statementk
End Select
```

In this case, the statement after the **Case Else** will execute if none of the previous expressions matches the *Expression* factor. Here is an example:

```
Sub Exercise
    Dim Answer As Byte

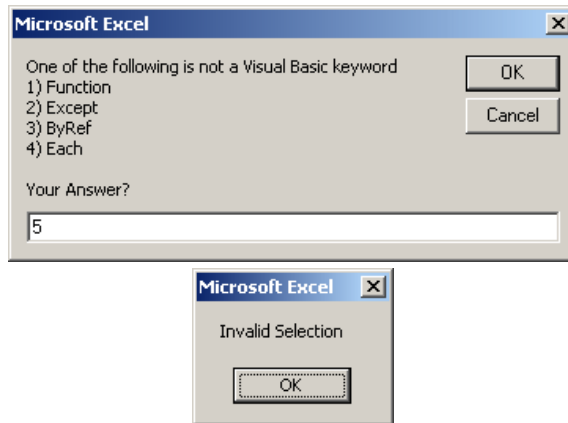
    Answer = CByte(InputBox( _
        "One of the following is not a Visual Basic keyword" & vbCrLf & _
        "1) Function" & vbCrLf & _
        "2) Except" & vbCrLf & _
        "3) ByRef" & vbCrLf & _
        "4) Each" & vbCrLf & vbCrLf & _
        "Your Answer? "))
```

```

Select Case Answer
Case 1
    MsgBox("Wrong: Function is a Visual Basic keyword." & vbCrLf & _
        "It is used to create a procedure of a function type")
Case 2
    MsgBox("Correct: Except is not a keyword in " & vbCrLf & _
        "Visual Basic but __except is a C++ " & vbCrLf & _
        "keyword used in Exception Handling")
Case 3
    MsgBox("Wrong: ByRef is a Visual Basic keyword used " & vbCrLf & _
        "to pass an argument by reference to a procedure")
Case 4
    MsgBox("Wrong: The "Each" keyword is used in " & vbCrLf & _
        "Visual Basic in a type of looping " & vbCrLf & _
        "used to "scan" a list of item.")
Case Else
    MsgBox("Invalid Selection")
End Select
End Sub

```

Here is an example of running the program:



❖ Practical Learning: Using Select Case

1. In the Objects combo box, select txtEmployeeNumber
2. In the Procedure combo box, select Exit
3. Implement the event as follows:

```


Private Sub txtEmployeeNumber_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    Dim EmployeeNumber As Long
    Dim EmployeeName As String

    EmployeeNumber = CLng(txtEmployeeNumber)

    Select Case EmployeeNumber
        Case 22804
            EmployeeName = "Helene Mukoko"
        Case 92746
            EmployeeName = "Raymond Kouma"
        Case 54080
            EmployeeName = "Henry Larson"
        Case 86285
            EmployeeName = "Gertrude Monay"
        Case 20860
            EmployeeName = "Paul Bertrand Yamaguchi"
        Case Else
            EmployeeName = "Unidentified Employee"
    End Select

    txtEmployeeName = EmployeeName
End Sub

```

4. On the Standard toolbar, click the Run Sub/UserForm button 
5. In the Employee # of the form, enter one of the numbers such as 54080 and press Tab
6. Close the form and return to Microsoft Visual Basic

Combining Cases

As mentioned in our introduction, the **Select Case** can use a value other than an integer. For example you can use a character:

```

Sub Exercise
    Dim Gender As String

    Gender = "M"

    Select Case Gender
        Case "F"

```

```

        MsgBox("Female")
    Case "M"
        MsgBox("Male")
    Case Else
        MsgBox("Unknown")
End Select

Return 0
End Function

End Sub

```

This would produce:



Notice that in this case we are using only upper case characters. If want to validate lower case characters also, we may have to create additional case sections for each. Here is an example:

```

Sub Exercise
    Dim Gender As String

    Gender = "f"

    Select Case Gender
        Case "f"
            MsgBox("Female")
        Case "F"
            MsgBox("Female")
        Case "m"
            MsgBox("Male")
        Case "M"
            MsgBox("Male")
        Case Else
            MsgBox("Unknown")
    End Select
End Sub

```

This would produce:



Instead of using one value for a case, you can apply more than one. To do this, on the right side of the **Case** keyword, you can separate the expressions with commas. Here are examples:

```

Sub Exercise
    Dim Gender As String

    Gender = "F"

    Select Case Gender
        Case "f", "F"
            MsgBox("Female")
        Case "m", "M"
            MsgBox("Male")
        Case Else
            MsgBox("Unknown")
    End Select
End Sub

```

Validating a Range of Cases

You can use a range of values for a case. To do this, on the right side of **Case**, enter the lower value, followed by **To**, followed by the higher value. Here is an example:

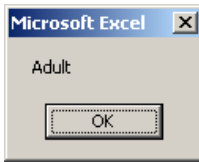
```

Sub Exercise
    Dim Age As Integer
    Age = 24

    Select Case Age
        Case 0 To 17
            MsgBox("Teen")
        Case 18 To 55
            MsgBox("Adult")
        Case Else
            MsgBox("Senior")
    End Select
End Sub

```

This would produce:



Checking Whether a Value IS

Consider the following procedure:

```
Sub Exercise
  Dim Number As Short

  Number = 448

  Select Case Number
    Case -602
      MsgBox("-602")
    Case 24
      MsgBox("24")
    Case 0
      MsgBox("0")
  End Select
End Sub
```

Obviously this **Select Case** statement will work in rare cases only when the expression of a case exactly match the value sought for. In reality, for this type of scenario, you could validate a range of values. The Visual Basic language provides an alternative. You can check whether the value of the *Expression* responds to a criterion instead of an exact value. To create it, you use the **Is** operator with the following formula:

Is Operator Value

You start with the **Is** keyword. It is followed by one of the Boolean operators we saw in the previous lessons: =, <>, <, <=, >, or >=. On the right side of the Boolean operator, type the desired value. Here are examples:

```
Sub Exercise
  Dim Number As Integer

  Number = -448

  Select Case Number
    Case Is < 0
      MsgBox("The number is negative")
    Case Is > 0
      MsgBox("The number is positive")
    Case Else
      MsgBox("0")
  End Select
End Sub
```

Although we used a natural number here, you can use any appropriate logical comparison that can produce a **True** or a **False** result. You can also combine it with the other alternatives we saw previously, such as separating the expressions of a case with commas.

Select...Case and the Conditional Built-In Functions

With the **Select...Case** statement, we saw how to check different values against a central one and take action when one of those matches the tag. Here is an example:

```
Sub Exercise
  Dim Number As Integer, MembershipType As String

  Number = 2

  Select Case Number
    Case 1
      MembershipType = "Teen"
    Case 2
      MembershipType = "Adult"
    Case Else
      MembershipType = "Senior"
  End Select

  MsgBox("Membership Type: " & MembershipType)
End Sub
```

This would produce:



We also saw that the Visual Basic language provides the **Choose()** function that can check a condition and take an action. The **Choose()** function is another alternative to a **Select...Case**

```
Function Choose( _
    ByVal Index As Double, _
    ByVal ParamArray Choice() As Variant _
) As Object
```

This function takes two required arguments. The first argument is equivalent to the *Expression* of our **Select Case** formula. As mentioned already, the first argument must be a number. This is the central value against which the other values will be compared. Instead of using **Case** sections, provide the equivalent *ExpressionX* values as a list of values in place of the second argument. The values are separated by commas. Here is an example:

```
Choose(Number, "Teen", "Adult", "Senior")
```

As mentioned already, the values of the second argument are provided as a list. Each member of the list uses an index. The first member of the list, which is the second argument of this function, has an index of 1. The second value of the argument, which is the third argument of the function, has an index of 2. You can continue adding the values of the second argument as you see fit.

When the **Choose()** function has been called, it returns a value of type **Variant**. You can retrieve that value, store it in a variable and use it as you see fit. Here is an example:

```
Sub Exercise
    Dim Number As Integer, MembershipType As String

    Number = 1

    MembershipType = Choose(Number, "Teen", "Adult", "Senior")

    MsgBox("Membership Type: " & MembershipType)
End Sub
```

This would produce:



Managing Conditional Statements

Conditional Nesting

So far, we have learned to create normal conditional statements and loops. Here is an example:

```
Sub Exercise
    Dim Number%

    Rem Request a number from the user
    Number% = InputBox("Enter a number that is lower than 5")

    Rem Find if the number is positive or 0
    If Number% >= 0 Then
        Rem If the number is positive, display it
        MsgBox (Number%)
    End If
End Sub
```

When this procedure executes, the user is asked to provide a number. If that number is positive, a message box displays it. If the user enters a negative number, nothing happens. In a typical program, after validating a condition, you may want to take action. To do that, you can create a section of program inside the validating conditional statement. In fact, you can create a conditional statement inside of another conditional statement. This is referred to as nesting a condition. Any condition can be nested inside of another and multiple conditions can be included inside of another.

Here is an example where an **If...Then** condition is nested inside of another **If...Then** statement:

```
Sub Exercise
    Dim Number%

    Rem Request a number from the user
    Number% = InputBox("Enter a number that is lower than 5")

    Rem Find if the number is positive or 0
    If Number% >= 0 Then
        Rem If the number is positive, accept it
        If Number% < 12 Then
            MsgBox (Number%)
        End If
    End If
End Sub
```

1. Change the code of the Exit event of the txtEmployeeNumber as follows:

```
Private Sub txtEmployeeNumber_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    Dim EmployeeNumber As Long
    Dim EmployeeName As String

    EmployeeNumber = 0

    If IsNumeric(txtEmployeeNumber) Then
        EmployeeNumber = CLng(txtEmployeeNumber)

        Select Case EmployeeNumber
            Case 22804
                EmployeeName = "Helene Mukoko"
            Case 92746
                EmployeeName = "Raymond Kouma"
            Case 54080
                EmployeeName = "Henry Larson"
            Case 86285
                EmployeeName = "Gertrude Monay"
            Case 20860
                EmployeeName = "Paul Bertrand Yamaguchi"
            Case Else
                EmployeeName = ""
        End Select

        txtEmployeeName = EmployeeName
    Else
        txtEmployeeNumber = ""
        txtEmployeeName = ""
        MsgBox "You must enter the employee number of " & _
            "the staff member who is processing this cleaning order", _
            vbOKOnly Or vbInformation, _
            "Georgetown Dry Cleaning Services"
    End If
End Sub
```

2. Press F5 to test the form
3. Enter the employee number as **92746**, and press Tab
4. Close the form and return to Microsoft Visual Basic

The Goto Statement

The **Goto** statement allows a program execution to jump to another section of a procedure in which it is being used. In order to use the **Goto** statement, insert a name on a particular section of your procedure so you can refer to that name. The name, also called a label, is made of one word and follows the rules we have applied to names (the name can be anything), then followed by a colon ":". Here is an example:

```
Sub Exercise()
    ' Do some thing(s) here

SomeLabelHere:
    ' Do some other thing(s) here
End Sub
```

After creating the label, you can process it. In the code before the label, you can do something. In that section, if a condition happens that calls for jumping to the label, then use a **GoTo** statement to send the flow to the corresponding label by typing the name of the label on the right side of **GoTo**. Here is an example:

```
Sub Exercise
    Dim Number%

    Rem Request a number from the user
    Number% = InputBox("Enter a number that is lower than 5")

    Rem Find if the number is positive or 0
    If Number% < 0 Then
        GoTo NegativeNumber
    Else
        Rem If the number is positive, display it
        MsgBox (Number%)
    End If

NegativeNumber:
    MsgBox "You entered a negative number"
End Sub
```

In the same way, you can create as many labels as you judge them necessary in your code and refer to them when you want. The name must be unique in its scope. This means that each label must have a unique name in the same procedure. Here is an example with various labels:

```

Sub Exercise
Dim Answer As Byte

Answer = InputBox(" ==- Multiple Choice Question ==- " & vbCrLf & _
    "To create a constant in your code, " & _
    "you can use the Constant keyword" & vbCrLf & _
    "Your choice (1=True/2=False)? ")

If Answer = 1 Then GoTo Wrong
If Answer = 2 Then GoTo Right

Wrong:
MsgBox("Wrong: The keyword used to create a constant is Const")
GoTo Leaving

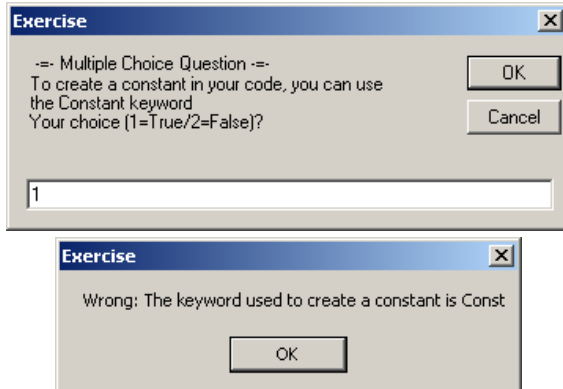
Right: MsgBox("Right: Constant is not a keyword")

Leaving:

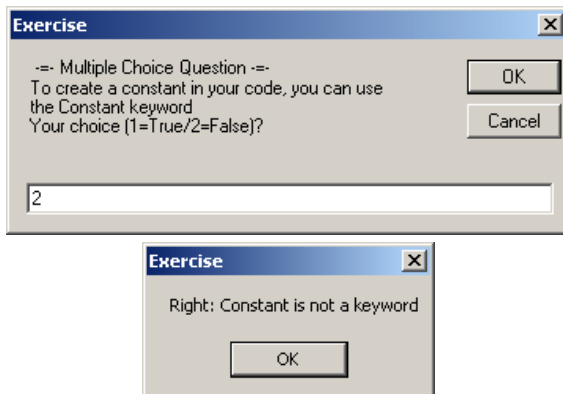
End Sub

```

Here is an example of executing the program with Answer = 1:



Here is another example of executing the same program with Answer = 2:



Negating a Conditional Statement

So far, we have learned to write a conditional statement that is true or false. You can reverse the true (or false) value of a condition by making it false (or true). To support this operation, the Visual Basic language provides an operator called **Not**. Its formula is:

Not *Expression*

When writing the statement, type **Not** followed by a logical expression. The expression can be a simple Boolean expression. Here is an example:

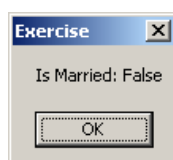
```

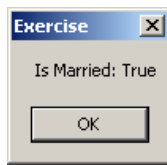
Sub Exercise
Dim IsMarried As Boolean

MsgBox("Is Married: " & IsMarried)
MsgBox("Is Married: " & Not IsMarried)
End Sub

```

This would produce:





In this case, the **Not** operator is used to change the logical value of the variable. When a Boolean variable has been "notted", its logical value has changed. If the logical value was **True**, it would be changed to **False** and vice versa. Therefore, you can inverse the logical value of a Boolean variable by "notting" or not "notting" it.

Now consider the following program we saw in Lesson 11:

```
Sub Exercise
    Dim IsMarried As Boolean
    Dim TaxRate As Double

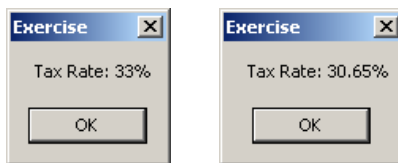
    TaxRate = 33.0

    MsgBox("Tax Rate: " & TaxRate & "%")

    IsMarried = True
    If IsMarried = True Then
        TaxRate = 30.65

        MsgBox("Tax Rate: " & TaxRate & "%")
    End If
End Sub
```

This would produce:



Probably the most classic way of using the **Not** operator consists of reversing a logical expression. To do this, you precede the logical expression with the **Not** operator. Here is an example:

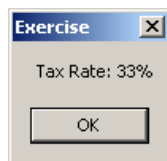
```
Sub Exercise
    Dim IsMarried As Boolean
    Dim TaxRate As Double

    TaxRate = 33.0
    MsgBox("Tax Rate: " & TaxRate & "%")

    IsMarried = True

    If Not IsMarried Then
        TaxRate = 30.65
        MsgBox("Tax Rate: " & TaxRate & "%")
    End If
End Sub
```

This would produce:



In the same way, you can negate any logical expression.

❖ Practical Learning: Negating a Condition

1. In the Object combo box, select txtDateLeft
2. In the Procedure combo box, select Exit
3. Implement the event as follows:

```
Private Sub txtDateLeft_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    If Not IsDate(txtDateLeft) Then
        MsgBox "The value you entered is not a valid date"
        txtDateLeft = Date
    End If
End Sub
```

4. Press F5 to test the form
5. Enter the employee number as **92746**, and press Tab

Loop Repeaters

Introduction

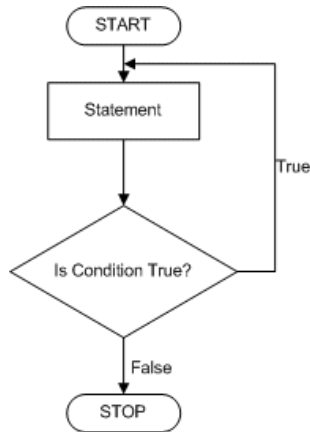
A loop is a technique used to repeat an action. The Visual Basic language presents many variations of loops. They combine the **Do** and the **Loop** keywords.

The Do...Loop While Loop

A typical loop can be used to perform an action while a condition is maintained true. To support this type of loop, the Visual Basic language provides the **Do...Loop While** statement.

The formula of the **Do... Loop While** loop is:

```
Do
    Statement(s)
Loop While Condition
```



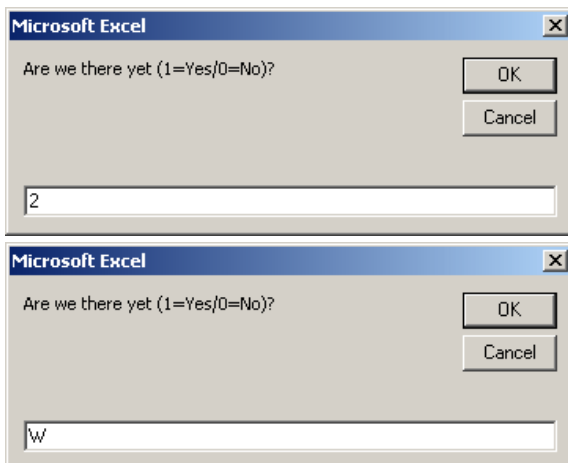
This interpreter first executes the *Statement* or *Statements*. After executing the *Statement(s)* section, the interpreter checks the *Condition*. If the *Condition* is true, then the interpreter returns to the *Statement(s)* and execute(s) it(them). The interpreter keeps doing this check-execution gymnastic. As long as the *Condition* is true, the *Statement(s)* section will be executed and the *Condition* will be tested again. If the *Condition* is false or once the condition becomes false, the statement will not be executed and the program will move on. Here is an example:

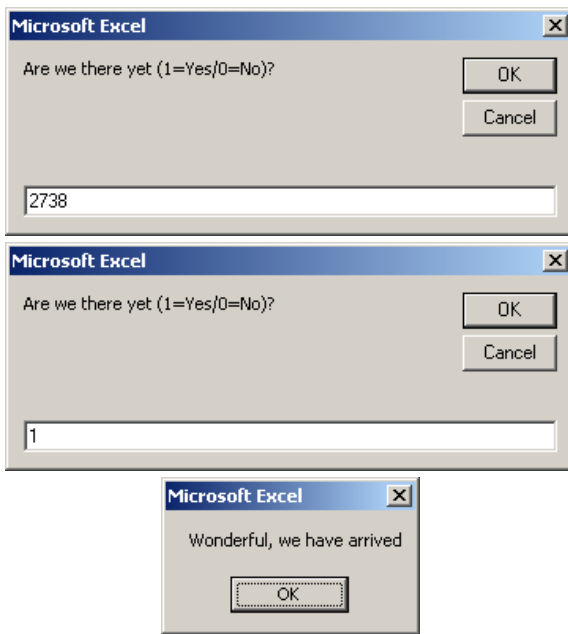
```
Sub Exercise
    Dim Answer As String

    Do
        Answer = InputBox("Are we there yet (1=Yes/0=No)? ")
    Loop While Answer <> "1"

    MsgBox("Wonderful, we have arrived")
End Sub
```

Here is an example of running the program:





As you may guess already, the *Condition* must provide a way for it to be true or to be false. Otherwise, the looping would be executed continually.

❖ Practical Learning: Using a Do...Loop While

1. In the Objects combo box, select txtCustomerPhone
2. In the Procedure combo box, select Exit
3. Implement the event as follows:

```
Private Sub txtCustomerPhone_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    ' This variable will be used to check the cells based on a row
    DimRowIndex As Integer
    ' This variable holds the customer phone number from the form
    Dim CustomerPhoneFromForm As String
    ' This variable holds the customer phone number from the worksheet
    Dim CustomerPhoneFromWorksheet As String

    Dim CustomerName As String

    ' Get the customer phone from the form
    CustomerPhoneFromForm = txtCustomerPhone

    ' Trim the left
    CustomerPhoneFromForm = LTrim(txtCustomerPhone)
    ' Trim the right side
    CustomerPhoneFromForm = RTrim(CustomerPhoneFromForm)
    ' Replace all spaces (in the middle of the number
    CustomerPhoneFromForm = Replace(CustomerPhoneFromForm, " ", "")
    ' Replace the left parentheses, if any
    CustomerPhoneFromForm = Replace(CustomerPhoneFromForm, "(", "")
    ' Replace the right parentheses, if any
    CustomerPhoneFromForm = Replace(CustomerPhoneFromForm, ")", "")
    ' Replace the dash -, if any
    CustomerPhoneFromForm = Replace(CustomerPhoneFromForm, "-", "")

    ' The phone number records start on row 6
    RowIndex = 6

    Do
        ' Get the customer phone from the second column
        CustomerPhoneFromWorksheet = _
            Worksheets("Customers").Cells(RowIndex, 2).Value

        CustomerName = (Worksheets("Customers").Cells(RowIndex, 3).Value)

        ' Trim the left
        CustomerPhoneFromWorksheet = LTrim(CustomerPhoneFromWorksheet)
        ' Trim the right side
        CustomerPhoneFromWorksheet = RTrim(CustomerPhoneFromWorksheet)
        ' Replace all spaces (in the middle of the number
        CustomerPhoneFromWorksheet = _
            Replace(CustomerPhoneFromWorksheet, " ", "")
        ' Replace the left parentheses, if any
        CustomerPhoneFromWorksheet = _
            Replace(CustomerPhoneFromWorksheet, "(", "")

        ' Replace the right parentheses, if any
        CustomerPhoneFromWorksheet = _
            Replace(CustomerPhoneFromWorksheet, ")", "")
        ' Replace the dash -, if any
        CustomerPhoneFromWorksheet = _
```


```

If CustomerPhoneFromWorksheet = CustomerPhoneFromForm Then
    txtCustomerName = CustomerName
End If

' Move to (continue with) the next row
RowIndex = RowIndex + 1

Loop While RowIndex <= 100
End Sub

```

4. On the Standard toolbar, click the Run Sub/UserForm button 
5. In the Employee # of the form, enter one of the numbers such as **54080** and press Tab
6. In the Customer Phone text box, enter one of the phone numbers and press Tab
7. Close the form and return to Microsoft Visual Basic

The Do...Loop Until Statement

While still supporting the ability to perform an action while a condition is true, the Visual Basic language provides an alternative to the **Do... Loop While** we saw earlier. The other solution uses the following formula:

```

Do
    Statement(s)
Loop Until Condition

```

Once again, the *Statement(s)* section executes first. After executing the *Statement(s)*, the interpreter checks the *Condition*. If the *Condition* is true, the interpreter returns to the *Statement(s)* section to execute it. This will continue until the *Condition* becomes false. Once the *Condition* becomes false, the interpreter gets out of this loop and continues with the section under the **Loop Until** line.

Here is an example:

```

Sub Exercise
    Dim Answer As String

    Do
        Answer = InputBox("Are we there yet (1=Yes/0=No)? ")
    Loop Until Answer = "1"

    MsgBox("Wonderful, we have arrived")
End Sub

```

The Do While... Loop Statement

As mentioned above, the **Do While... Loop** expression executes a statement first before checking a condition that would allow it to repeat. If you want to check a condition first before executing a statement, you can use another version as **Do While... Loop**. Its formula is:

```

Do While Condition
    Statement(s)
Loop

```

In this case, the interpreter checks the *Condition* first. If the *Condition* is true, the interpreter then executes the *Statement(s)* and checks the *Condition* again. If the *Condition* is false, or when the *Condition* becomes false, the interpreter skips the *Statement(s)* section and continues with the code below the **Loop** keyword.

Here is an example:

```

Sub Exercise
    Dim Number As Integer

    Do While Number < 46
        Number = CInt(InputBox("Enter a number"))
        Number = Number + 1
    Loop

    MsgBox ("Counting Stopped at: " & Number)
End Sub

```

The Do Until... Loop Statement

Instead of performing an action while a condition is true, you may want to do something until a condition becomes false. To support this, the Visual Basic language provides a loop that involves the **Until** keyword. The formula to use is:

```

Do Until Condition
    Statement(s)
Loop

```

This loop works like the **Do While... Loop** expression. The interpreter examines the *Condition* first. If the condition is true, then it executes the *Statement(s)* section.

Here is an example:

```
Sub Exercise
  Dim Answer As String
  Answer = "0"

  Do Until Answer = "1"
    Answer = InputBox("Are we there yet (1=Yes/0=No)? ")
  Loop

  MsgBox("Wonderful, we have arrived")
End Sub
```

Loop Counters

Introduction

The looping statements we reviewed above are used when you do not know or cannot anticipate the number of times a condition needs to be checked in order to execute a statement. If you know with certainty how many times you want to execute a statement, you can use another form of loops that use the **For...Next** expression.

The For...To...Next Loop

One of the loop counters you can use is **For...To...Next**. Its formula is:

```
For Counter = Start To End
  Statement(s)
Next
```

Used for counting, the expression begins counting at the *Start* point. Then it examines whether the current value (after starting to count) is lower than *End*. If that's the case, it then executes the *Statement(s)*. Next, it increments the value of *Counter* by 1 and examines the condition again. This process goes on until the value of *Counter* becomes equal to the *End* value. Once this condition is reached, the looping stops.

Here is an example:

```
Sub Exercise
  Dim Number As Integer

  For Number = 5 To 16
    MsgBox(Number)
  Next

  MsgBox("Counting Stopped at: " & Number)
End Sub
```

❖ Practical Learning: Using a For Loop


1. Locate the Exit event of the txtEmployeeNumber control and change it as follows:

```
Private Sub txtEmployeeNumber_Exit(ByVal Cancel As MSForms.ReturnBoolean)
  Dim RowCounter As Integer
  Dim EmployeeNumberFromForm As Long
  Dim EmployeeNumberFromWorksheet As Long
  Dim EmployeeName As String

  ' If the user had entered an employee number on the form,
  ' retrieve it
  If IsNumeric(txtEmployeeNumber) Then
    EmployeeNumberFromForm = CLng(txtEmployeeNumber)
  Else
    EmployeeNumberFromForm = 0
  End If

  ' We assume the employee numbers are stored in the second column,
  ' from row 6 to row 106. That is, about 100 employees
  For RowCounter = 6 To 106
    ' Get the employee number on the current cell
    EmployeeNumberFromWorksheet = _
      Worksheets("Employees").Cells(RowCounter, 2).Value
    EmployeeName = Worksheets("Employees").Cells(RowCounter, 3).Value

    ' If you find an employee number that is the same as
    ' the user entered into the form, get its corresponding name
    If EmployeeNumberFromWorksheet = EmployeeNumberFromForm Then
      ' and display it on the form
      txtEmployeeName = EmployeeName
    End If
  Next
End Sub
```

2. On the Standard toolbar, click the Run Sub/UserForm button 
3. In the Employee # of the form, enter one of the numbers such as 54080 and press Tab

Stepping the Counting Loop

The formula above will increment the counting by 1 at the end of each statement. If you want to control how the incrementing processes, you can set your own, using the **Step** option. Here is the formula:

```
For Counter = Start To End Step Increment
    Statement(s)
Next
```

You can set the incrementing value to your choice. If the value of *Increment* is positive, the *Counter* will be added its value. Here is an example:

```
Sub Exercise
    Dim Number As Integer

    For Number = 5 To 42 Step 4
        MsgBox(Number)
    Next

    MsgBox("Counting Stopped at: " & Number)
End Sub
```

You can also set a negative value to the *Increment* factor, in which case the *Counter* will be subtracted the set value.

For Each Item In the Loop

Since the **For...Next** loop is used to execute a group of statements based on the current result of the loop counting from *Start* to *End*, an alternative is to state various steps in the loop and execute a group of statements for each one of the elements in the group. This is mostly used when dealing with a collection of items.

The formula is:

```
For Each Element In Group
    Statement(s)
Next Element
```

The loop will execute the *Statement(s)* for each *Element* in the *Group*.

Exiting a Procedure or a Loop

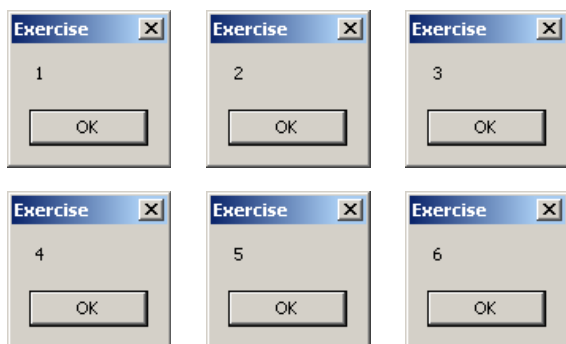
Exiting a Procedure

In the conditional statements and loops we have created so far, we assumed that the whole condition would be processed. Here is an example:

```
Sub Exercise
    Dim Number As Integer

    For Number = 1 To 6
        MsgBox(Number)
    Next
End Sub
```

This would produce:



In some cases, you may want to exit a conditional statement or a loop before its end. To assist with this, the Visual Basic language provides the **Exit** keyword. This keyword works like an operator. It can be applied to a procedure or a **For** loop. Consider the following procedure:

```
Sub Exercise()
    MsgBox("Patricia Katts")
    MsgBox("Gertrude Monay")
    MsgBox("Hermine Nkolo")
    MsgBox("Paul Bertrand Yamaguchi")
End Sub
```

When the procedure is called, it displays four message boxes that each shows a name. Imagine that at some point you want to ask the interpreter to stop in the middle of a procedure. To do this, in the section where you want to stop the flow of a procedure, type **Exit Sub**. Here is an example:

```
Sub Exercise()
    MsgBox("Patricia Katts")
    MsgBox("Gertrude Monay")
    Exit Sub
    MsgBox("Hermine Nkolo")
    MsgBox("Paul Bertrand Yamaguchi")
End Sub
```

This time, when the program runs, the procedure would be accessed and would start displaying the message boxes. After displaying two, the **Exit Sub** would ask the interpreter to stop and get out of the procedure.

Because a function is just a type of procedure that is meant to return a value, you can use the **Exit** keyword to get out of a function before the **End Function** line. To do this, in the section where you want to stop the flow of the function, type **Exit Function**.

- Change the code of the FindEmployee macro as follows:

```
Sub FindEmployeee()
'
' FindEmployeee Macro
'
' Keyboard Shortcut: Ctrl+Shift+E
'
    Dim EmployeeNumber As Long, EmployeeName As String

    If IsEmpty(Range("C4")) Then
        MsgBox "You must enter the employee number in cell C4"
        Range("D4").FormulaR1C1 = ""
        Exit Sub
    Else
        EmployeeNumber = CLng(Range("C4"))
    End If

    If EmployeeNumber = 22804 Then
        Range("D4").FormulaR1C1 = "Helene Mukoko"
    ElseIf EmployeeNumber = 92746 Then
        Range("D4").FormulaR1C1 = "Raymond Kouma"
    ElseIf EmployeeNumber = 54080 Then
        Range("D4").FormulaR1C1 = "Henry Larson"
    ElseIf EmployeeNumber = 86285 Then
        Range("D4").FormulaR1C1 = "Gertrude Monay"
    Else
        Range("D4").FormulaR1C1 = "Unknown"
    End
End
```

Exiting a For Loop Counter

You can also exit a **For** loop. To do this, in the section where you want to stop, type **Exit For**. Here is an example to stop a continuing **For** loop:

```
Sub Exercise()
    Dim Number As Integer

    For Number = 1 To 12
        MsgBox(Number)

        If Number = 4 Then
            Exit For
        End If
    Next
End Sub
```

When this program executes, it is supposed to display numbers from 1 to 12, but an **If...Then** condition states that if it gets to the point where the number is 4, it should stop. If you use an **Exit For** statement, the interpreter would stop the flow of **For** and continue with code after the **Next** keyword.

Exiting a Do Loop

You can also use the **Exit** operator to get out of a **Do** loop. To do this, inside of a **Do** loop where you want to stop, type **Exit Do**.

❖ Practical Learning: Exiting Code

1. Locate the Exit event of the txtCustomerPhone control and change it as follows:

```
Private Sub txtCustomerPhone_Exit(ByVal Cancel As MSForms.ReturnBoolean)
' This variable will be used to check the cells based on a row
Dim RowIndex As Integer
' This variable holds the customer phone number from the form
```

```

Dim CustomerPhoneFromForm As String
' This variable holds the customer phone number from the worksheet
Dim CustomerPhoneFromWorksheet As String

Dim CustomerName As String

' Get the customer phone from the form
CustomerPhoneFromForm = txtCustomerPhone

' Trim the left side
CustomerPhoneFromForm = LTrim(txtCustomerPhone)
' Trim the right side
CustomerPhoneFromForm = RTrim(CustomerPhoneFromForm)
' Replace all spaces (in the middle of the phone number)
CustomerPhoneFromForm = Replace(CustomerPhoneFromForm, " ", "")
' Replace the left parenthesis, if any
CustomerPhoneFromForm = Replace(CustomerPhoneFromForm, "(", "")
' Replace the right parenthesis, if any
CustomerPhoneFromForm = Replace(CustomerPhoneFromForm, ")", "")
' Replace the dash -, if any
CustomerPhoneFromForm = Replace(CustomerPhoneFromForm, "-", "")

' The phone number records start on row 6
RowIndex = 6

Do
    If IsEmpty(Worksheets("Customers").Cells(CellIndex, 2).Value) Then
        Exit Sub
    End If

    CustomerPhoneFromWorksheet = _
        Worksheets("Customers").Cells(RowIndex, 2).Value

    CustomerName = (Worksheets("Customers").Cells(RowIndex, 3).Value)


    ' Trim the left
    CustomerPhoneFromWorksheet = LTrim(CustomerPhoneFromWorksheet)
    ' Trim the right side
    CustomerPhoneFromWorksheet = RTrim(CustomerPhoneFromWorksheet)
    ' Replace all spaces (in the middle of the number)
    CustomerPhoneFromWorksheet = _
        Replace(CustomerPhoneFromWorksheet, " ", "")
    ' Replace the left parentheses, if any
    CustomerPhoneFromWorksheet = _
        Replace(CustomerPhoneFromWorksheet, "(", "")

    ' Replace the right parentheses, if any
    CustomerPhoneFromWorksheet = _
        Replace(CustomerPhoneFromWorksheet, ")", "")
    ' Replace the dash -, if any
    CustomerPhoneFromWorksheet = _
        Replace(CustomerPhoneFromWorksheet, "-", "")

    If CustomerPhoneFromWorksheet = CustomerPhoneFromForm Then
        txtCustomerName = CustomerName
        Exit Do
    End If

    RowIndex = RowIndex + 1
Loop While RowIndex <= 100
End Sub

```

2. On the Standard toolbar, click the Run Sub/UserForm button 
3. Process an order
4. Close the form and return to Microsoft Visual Basic



Logical Conjunction and Disjunction

Logical Conjunction

Introduction

As mentioned already, you can nest one conditional statement inside of another. To illustrate, imagine you create a workbook that would be used by a real estate company that sells houses. You may face a customer who wants to purchase a single family house but the house should not cost more than \$550,001. To implement this scenario, you can first write a procedure that asks the user to specify a type of house and then a conditional statement would check it. Here is an example:

```
Sub Exercise
    Dim TypeOfHouse As String
    Dim Choice As Integer
    Dim Value As Double

    TypeOfHouse = "Unknown"

    Choice = CInt(InputBox("Enter the type of house you want to purchase" _
        & vbCrLf & _
        "1. Single Family" & vbCrLf & _
        "2. Townhouse" & vbCrLf & _
        "3. Condominium" & vbCrLf & vbCrLf & _
        "You Choice? "))
    Value = CDbI(InputBox("Up to how much can you afford?"))

    TypeOfHouse = Choose(Choice, "Single Family", _
        "Townhouse", _
        "Condominium")
End Sub
```

If the user selects a single family, you can then write code inside the conditional statement of the single family. Here is an example:

```
Sub Exercise
    Dim TypeOfHouse As String
    Dim Choice As Integer
    Dim Value As Double

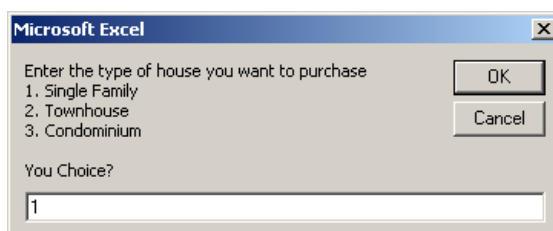
    TypeOfHouse = "Unknown"

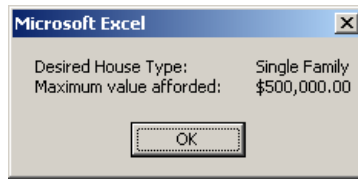
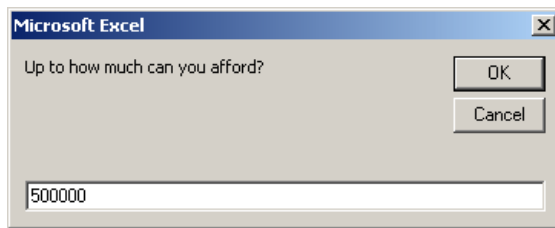
    Choice = CInt(InputBox("Enter the type of house you want to purchase" _
        & vbCrLf & _
        "1. Single Family" & vbCrLf & _
        "2. Townhouse" & vbCrLf & _
        "3. Condominium" & vbCrLf & vbCrLf & _
        "You Choice? "))
    Value = CDbI(InputBox("Up to how much can you afford?"))

    TypeOfHouse = Choose(Choice, "Single Family", _
        "Townhouse", _
        "Condominium")

    If Choice = 1 Then
        MsgBox("Desired House Type:      " & vbTab & TypeOfHouse & vbCrLf & _
            "Maximum value afforded:  " & vbTab & FormatCurrency(Value))
    End If
End Sub
```

Here is an example of running the program:





In that section, you can then write code that would request and check the value the user entered. If that value is valid, you can take necessary action. Here is an example:

```
Sub Exercise
    Dim TypeOfHouse As String
    Dim Choice As Integer
    Dim Value As Double

    TypeOfHouse = "Unknown"

    Choice = CInt(InputBox("Enter the type of house you want to purchase" _
        & vbCrLf & _
        "1. Single Family" & vbCrLf & _
        "2. Townhouse" & vbCrLf & _
        "3. Condominium" & vbCrLf & vbCrLf & _
        "You Choice? "))
    Value = CDbI(InputBox("Up to how much can you afford?"))

    TypeOfHouse = Choose(Choice, "Single Family", _
        "Townhouse", _
        "Condominium")

    If Choice = 1 Then
        MsgBox ("Desired House Type:    " & vbTab & TypeOfHouse & vbCrLf & _
            "Maximum value afforded:    " & vbTab & FormatCurrency(Value))

        If Value <= 550000 Then
            MsgBox ("Desired House Matched")
        Else
            MsgBox ("The House Doesn't Match the Desired Criteria")
        End If
    End If
End Sub
```

A Conditional Conjunction

Using conditional nesting, we have seen how you can write one conditional statement that depends on another. But you must write one first condition, check it, then nest the other condition. This works fine and there is nothing against it.

To provide you with an alternative, you can use what is referred to as a logical conjunction. It consists of writing one **If...Then** expression that checks two conditions at the same time. To illustrate, once again consider a customer who wants to purchase a single family home that is less than \$550,000. You can consider two statements as follows:

- a. The house is single family
- b. The house costs less than \$550,000

To implement it, you would need to write an If...Then condition as:

```
If The house is single family AND The house costs less than $550,000 Then
    Validate
End If
```

In the Visual Basic language, the operator used to perform a logical conjunction is **And**. Here is an example of using it:

```
Sub Exercise
    Dim TypeOfHouse As String
    Dim Choice As Integer
    Dim Value As Double

    TypeOfHouse = "Unknown"

    Choice = _
        CInt(InputBox("Enter the type of house you want to purchase" & vbCrLf & _
            "1. Single Family" & vbCrLf & _
            "2. Townhouse" & vbCrLf & _
            "3. Condominium" & vbCrLf & vbCrLf & _
            "You Choice? "))
    Value = CDbI(InputBox("Up to how much can you afford?"))

    TypeOfHouse = Choose(Choice, "Single Family", _
```

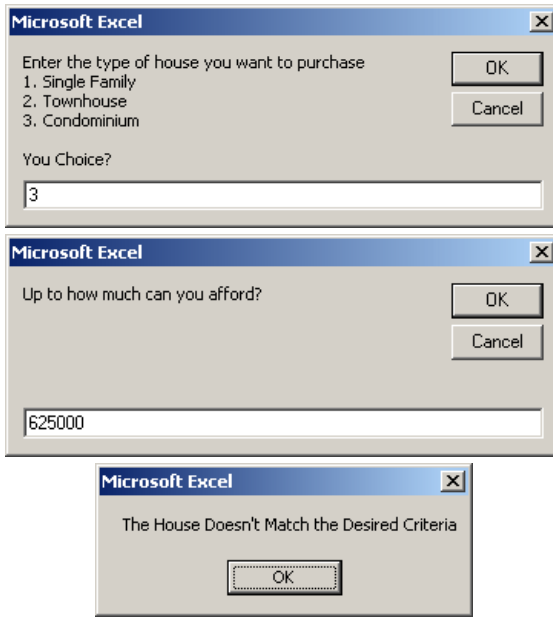
```

        "Townhouse", _
        "Condominium")

If TypeOfHouse = "Single Family" And Value <= 550000 Then
    MsgBox("Desired House Type:      " & vbTab & TypeOfHouse & vbCrLf & _
           "Maximum value afforded:  " & vbTab & FormatCurrency(Value))
    MsgBox("Desired House Matched")
Else
    MsgBox("The House Doesn't Match the Desired Criteria")
End If
End Sub

```

Here is an example of running the program:



By definition, a logical conjunction combines two conditions. To make the program easier to read, each side of the conditions can be included in parentheses. Here is an example:

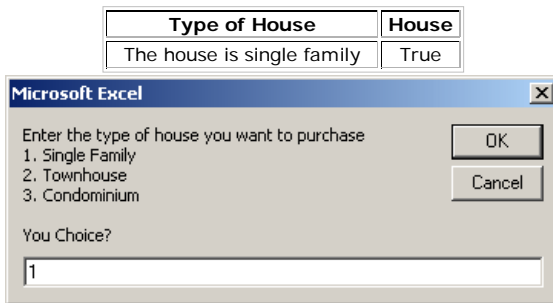
```

Sub Exercise
    . . . No Change

    If (TypeOfHouse = "Single Family") And (Value <= 550000) Then
        MsgBox("Desired House Type:      " & vbTab & TypeOfHouse & vbCrLf & _
               "Maximum value afforded:  " & vbTab & FormatCurrency(Value))
        MsgBox("Desired House Matched")
    Else
        MsgBox("The House Doesn't Match the Desired Criteria")
    End If
End Sub

```

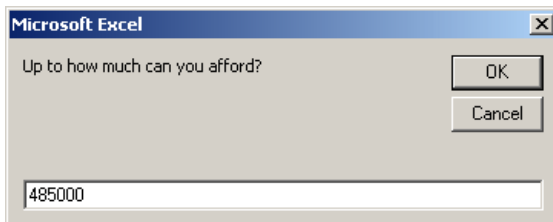
To understand how logical conjunction works, from a list of real estate properties, after selecting the house type, if you find a house that is a single family home, you put it in the list of considered properties:



If you find a house that is less than or equal to \$550,000, you retain it:

Price Range	Value
\$550,000	True

For the current customer, you want a house to meet BOTH criteria. If the house is a town house, based on the request of our customer, its conditional value is false. If the house is less than \$550,000, such as \$485,000, the value of the Boolean Value is true:





If the house is a town house, based on the request of our customer, its conditional value is false. If the house is more than \$550,000, the value of the Boolean Value is true. In logical conjunction, if one of the conditions is false, the result is false also. This can be illustrated as follows:

Type of House	House Value	Result
Town House	\$625,000	Town House AND \$625,000
False	False	False

Suppose we find a single family home. The first condition is true for our customer. With the AND Boolean operator, if the first condition is true, then we consider the second criterion. Suppose that the house we are considering costs \$750,500: the price is out of the customer's range. Therefore, the second condition is false. In the AND Boolean algebra, if the second condition is false, even if the first is true, the whole condition is false. This would produce the following table:

Type of House	House Value	Result
Single Family	\$750,500	Single Family AND \$750,500
True	False	False

Suppose we find a townhouse that costs \$420,000. Although the second condition is true, the first is false. In Boolean algebra, an AND operation is false if either condition is false:

Type of House	House Value	Result
Town House	\$420,000	Town House AND \$420,000
False	True	False

If we find a single family home that costs \$345,000, both conditions are true. In Boolean algebra, an AND operation is true if BOTH conditions are true. This can be illustrated as follows:

Type of House	House Value	Result
Single Family	\$345,000	Single Family AND \$345,000
True	True	True

These four tables can be resumed as follows:

If Condition1 is	If Condition2 is	Condition1 AND Condition2
False	False	False
False	True	False
True	False	False
True	True	True

As you can see, a logical conjunction is true only if BOTH conditions are true.

Combining Conjunctions

As seen above, the logical conjunction operator is used to combine two conditions. In some cases, you will need to combine more than two conditions. Imagine a customer wants to purchase a single family house that costs up to \$450,000 with an indoor garage. This means that the house must fulfill these three requirements:

- A. The house is a single family home
- B. The house costs less than \$450,001
- C. The house has an indoor garage

Here is the program that could be used to check these conditions:

```
Sub Exercise
Dim TypeOfHouse As String
Dim Choice As Integer
Dim Value As Double
Dim IndoorGarageAnswer As Integer
Dim Answer As String

TypeOfHouse = "Unknown"

Choice = _
    CInt(InputBox("Enter the type of house you want to purchase" _
        & vbCrLf & _
        "1. Single Family" & vbCrLf & _
```

```

    "2. Townhouse" & vbCrLf & _
    "3. Condominium" & vbCrLf & vbCrLf & _
    "You Choice? ")
Value = CDBl(InputBox("Up to how much can you afford?"))

TypeOfHouse = Choose(Choice, "Single Family", _
    "Townhouse", _
    "Condominium")

IndoorGarageAnswer = _
    MsgBox("Does the house have an indoor garage (1=Yes/0=No)?", _
        vbQuestion Or vbYesNo, _
        "Real Estate")
Answer = IIf(IndoorGarageAnswer = vbYes, "Yes", "No")

If (TypeOfHouse = "Single Family") And _
(Value <= 550000) And _
(IndoorGarageAnswer = vbYes) Then
    MsgBox "Desired House Type:      " & vbCrLf & TypeOfHouse & vbCrLf & _
        "Maximum value afforded:    " & vbCrLf & _
        FormatCurrency(Value) & vbCrLf & _
        "House has indoor garage:    " & vbCrLf & Answer
    MsgBox "Desired House Matched"
Else
    MsgBox ("The House Doesn't Match the Desired Criteria")
End If
End Sub

```

We saw that when two conditions are combined, the interpreter first checks the first condition, followed by the second. In the same way, if three conditions need to be considered, the interpreter evaluates the truthfulness of the first condition:

Type of House
A
Town House
False

If the first condition (or any condition) is false, the whole condition is false, regardless of the outcome of the other(s). If the first condition is true, then the second condition is evaluated for its truthfulness:

Type of House	Property Value
A	B
Single Family	\$655,000
True	False

If the second condition is false, the whole combination is considered false:

A	B	A AND B
True	False	False

When evaluating three conditions, if either the first or the second is false, since the whole condition would become false, there is no reason to evaluate the third. If both the first and the second conditions are false, there is also no reason to evaluate the third condition. Only if the first two conditions are true will the third condition be evaluated whether it is true:

Type of House	Property Value	Indoor Garage
A	B	C
Single Family	\$425,650	None
True	True	False

The combination of these conditions in a logical conjunction can be written as **A AND B AND C**. If the third condition is false, the whole combination is considered false:

A	B	A AND B	C	A AND B AND C
True	True	True	False	False

From our discussion so far, the truth table of the combinations can be illustrated as follows:

A	B	C	A AND B AND C
False	Don't Care	Don't Care	False
True	False	Don't Care	False
True	True	False	False

The whole combination is true only if all three conditions are true. This can be illustrated as follows:

A	B	C	A AND B AND C
False	False	False	False
False	False	True	False
True	False	False	False
True	False	True	False

False	True	False	False
False	True	True	False
True	True	False	False
True	True	True	True

Logical Disjunction: OR

Introduction

Our real estate company has single family homes, townhouses, and condominiums. All of the condos have only one level, also referred to as a story. Some of the single family homes have one story, some have two and some others have three levels. All townhouses have three levels.

Another customer wants to buy a home. The customer says that he primarily wants a condo, but if our real estate company doesn't have a condominium, that is, if the company has only houses, whatever it is, whether a house or a condo, it must have only one level (story) (due to an illness, the customer would not climb the stairs). When considering the properties of our company, we would proceed with these statements:

- The property is a condominium
- The property has one story

If we find a condo, since all of our condos have only one level, the criterion set by the customer is true. Even if we were considering another (type of) property, it wouldn't matter. This can be resumed in the following table:

Type of House	House
Condominium	True

The other properties would not be considered, especially if they have more than one story:

Number of Stories	Value
3	False

We can show this operation as follows:

Condominium	One Story	Condominium or 1 Story
True	False	True

Creating a Logical Disjunction

To support "either or" conditions in the Visual Basic language, you use the **Or** operator. Here is an example:

```
Sub Exercise
    Dim TypeOfHouse As String
    Dim Choice As Integer
    Dim Stories As Integer

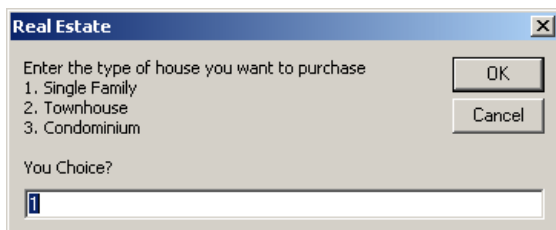
    TypeOfHouse = "Unknown"

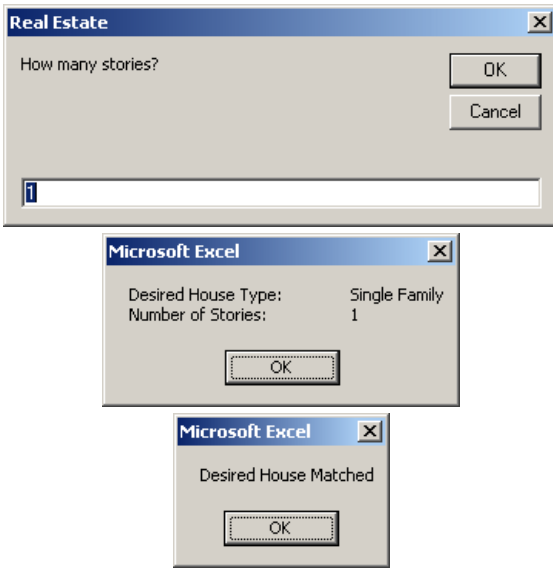
    Choice = _
        CInt(InputBox("Enter the type of house you want to purchase" & vbCrLf & _
            "1. Single Family" & vbCrLf & _
            "2. Townhouse" & vbCrLf & _
            "3. Condominium" & vbCrLf & vbCrLf & _
            "You Choice? ", "Real Estate", 1))

    TypeOfHouse = Choose(Choice, "Single Family", _
        "Townhouse", _
        "Condominium")
    Stories = CInt(InputBox("How many stories?", "Real Estate", 1))

    If Choice = 1 Or Stories = 1 Then
        MsgBox("Desired House Type:" & vbTab & TypeOfHouse & vbCrLf & _
            "Number of Stories:" & vbTab & vbTab & Stories)
        MsgBox("Desired House Matched")
    Else
        MsgBox("The House Doesn't Match the Desired Criteria")
    End If
End Sub
```

Here is an example of running the program:





As done for the And operator, to make a logical disjunction easy to read, you can include each statement in parentheses:

```

Sub Exercise
    . . . No Change

    If (Choice = 1) Or (Stories = 1) Then
        MsgBox ("Desired House Type:" & vbTab & TypeOfHouse & vbCrLf & _
            "Number of Stories:" & vbTab & vbTab & Stories)
        MsgBox ("Desired House Matched")
    Else
        MsgBox ("The House Doesn't Match the Desired Criteria")
    End If
End Sub
    
```

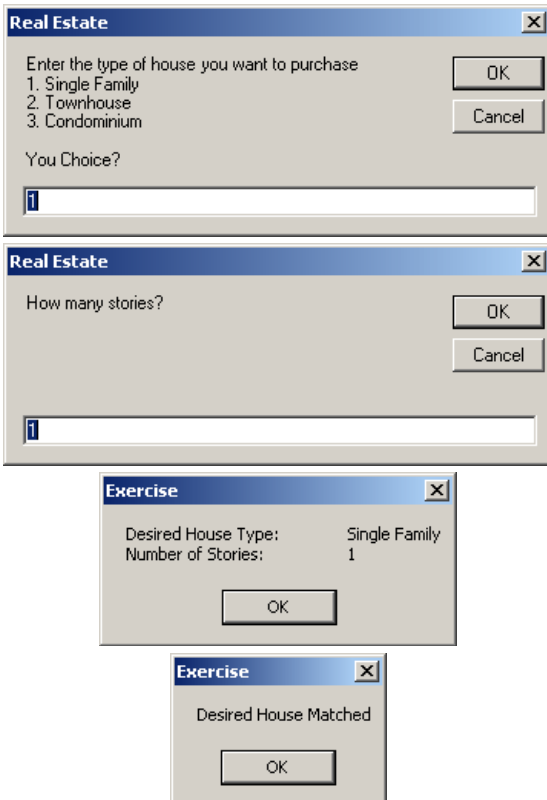
Suppose that, among the properties our real estate company has available, there is no condominium. In this case, we would then consider the other properties:

Type of House	House
Single Family	False

If we have a few single family homes, we would look for one that has only one story. Once we find one, our second criterion becomes true:

Type of House	One Story	Condominium OR 1 Story
False	True	True

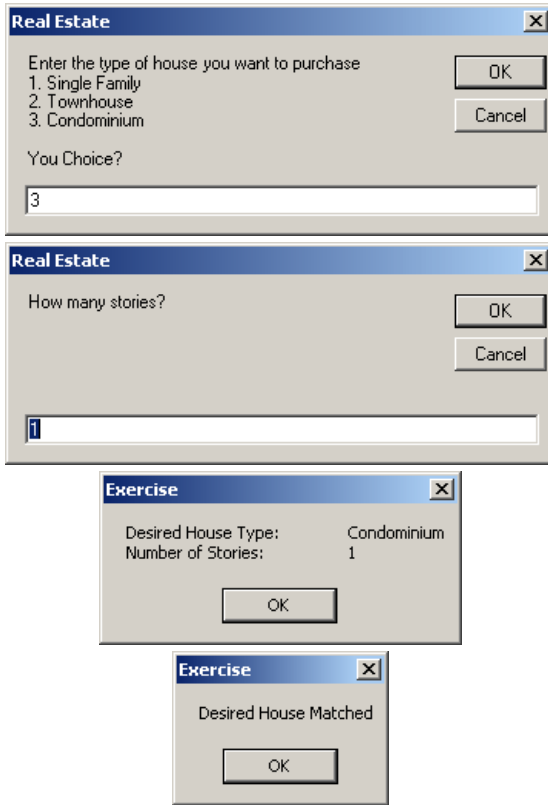
This can be illustrated in the following run of the above program:



If we find a condo and it is one story, both criteria are true. This can be illustrated in the

Type of House	One Story	Condominium OR 1 Story
False	True	True
True	True	True

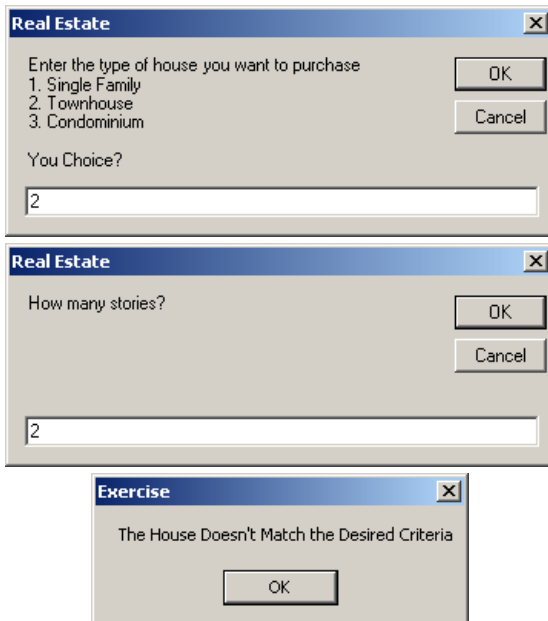
The following run of the program demonstrates this:



A Boolean OR operation produces a false result only if BOTH conditions ARE FALSE:

If Condition1 is	If Condition2 is	Condition1 OR Condition2
False	True	True
True	False	True
True	True	True
False	False	False

Here is another example of running the program:



Combinations of Disjunctions

As opposed to evaluating only two conditions, you may face a situation that presents three of them and must consider a combination of more than two conditions. You would apply the same logical approach we reviewed for the logical conjunction, except that, in a group of logical disjunctions, if one of them is true, the whole statement becomes true.



Error Handling

Handling Errors

Introduction to Errors

A computer application is supposed to run as smooth as possible. Unfortunately, this is not always the case. A form may close unexpectedly. A control on a form may hide itself at the wrong time. The application may crash. A calculation may produce unexpected results, etc.

You can predict some of these effects and take appropriate actions. Some other problems are not under your control. Fortunately, both Microsoft Excel and the VBA language provide various tools or means of dealing with errors.

❖ Practical Learning: Introducing Error Handling

1. Open the **Georgetown Dry Cleaning Services1** spreadsheet and click the Employees tab

Employee #	First Name	Last Name	Hourly Salary
95947	Lydia	Becker	20.50
80850	Lynda	Backers	22.82
60485	Maria	Stonner	18.25
47490	George	Machs	32.05
30608	David	Leaner	14.50
26846	Jerry	Leggs	16.05

2. Click the Payroll tab
3. Click the TimeSheet tab
4. To save the workbook and prepare it for code, press F12
5. Specify the folder as (My) Documents
6. In the Save As Type combo box, select Excel Macro-Enabled Workbook
7. Click Save

Introduction to Handling Errors

To deal with errors in your code, the Visual Basic language provides various techniques. One way you can do this is to prepare your code for errors. When an error occurs, you would present a message to the user to make him/her aware of the issue (the error).

To prepare a message, you create a section of code in the procedure where the error would occur. To start that section, you create a **label**. Here is an example:

```
Private Sub cmdCalculate_Click()
ThereWasBadCalculation:
End Sub
```

After (under) the label, you can specify your message. Most of the time, you formulate the message using a message box. Here is an example:

```
Private Sub cmdCalculate_Click()
ThereWasBadCalculation:
MsgBox "There was a problem when performing the calculation"
End Sub
```

If you simply create a label and its message like this, its section would always execute:


```
Private Sub cmdCalculate_Click()
    Dim HourlySalary As Double, WeeklyTime As Double
    Dim WeeklySalary As Double

    ' One of these two lines could produce an error, such as
    ' if the user types an invalid number
    HourlySalary = Cdbl(txtHourlySalary)
    WeeklyTime = Cdbl(txtWeeklyTime)

    ' If there was an error, the flow would jump to the label
    WeeklySalary = HourlySalary * WeeklyTime

    txtWeeklySalary = FormatNumber(WeeklySalary)

ThereWasBadCalculation:
    MsgBox "There was a problem when performing the calculation"
End Sub
```

To avoid this, you should find a way to interrupt the flow of the program before the label section. One way you can do this is to add a line marked **Exit Sub** before the label. This would be done as follows:

```
Private Sub cmdCalculate_Click()
    Dim HourlySalary As Double, WeeklyTime As Double
    Dim WeeklySalary As Double

    ' One of these two lines could produce an error, such as
    ' if the user types an invalid number
    HourlySalary = Cdbl(txtHourlySalary)
    WeeklyTime = Cdbl(txtWeeklyTime)

    ' If there was an error, the flow would jump to the label
    WeeklySalary = HourlySalary * WeeklyTime

    txtWeeklySalary = FormatNumber(WeeklySalary)

    Exit Sub

ThereWasBadCalculation:
    MsgBox "There was a problem when performing the calculation"
End Sub
```

In Case of Error

Jump to a Label

We saw that you can create a label that would present a message to the user when an error occurs. Before an error occurs, you would indicate to the compiler where to go if an error occurs. To provide this information, under the line that starts the procedure, type an **On Error GoTo** expression followed by the name of the label where you created the message. Here is an example:

```
Private Sub cmdCalculate_Click()
    On Error GoTo ThereWasBadCalculation

    Dim HourlySalary As Double, WeeklyTime As Double
    Dim WeeklySalary As Double

    ' One of these two lines could produce an error, such as
    ' if the user types an invalid number
    HourlySalary = Cdbl(txtHourlySalary)
    WeeklyTime = Cdbl(txtWeeklyTime)

    ' If there was an error, the flow would jump to the label
    WeeklySalary = HourlySalary * WeeklyTime

    txtWeeklySalary = FormatNumber(WeeklySalary)

    Exit Sub

ThereWasBadCalculation:
    MsgBox "There was a problem when performing the calculation"
End Sub
```

The **On Error GoTo** indicates to the compiler where to transfer code if an error occurs.

Go to a Numbered Label

Instead of defining a lettered label where to jump in case of error, you can create a numeric label:

```
Private Sub cmdCalculate_Click()
    Dim HourlySalary As Double, WeeklyTime As Double
    Dim WeeklySalary As Double

    HourlySalary = Cdbl(txtHourlySalary)
    WeeklyTime = Cdbl(txtWeeklyTime)
    WeeklySalary = HourlySalary * WeeklyTime

    txtWeeklySalary = FormatNumber(WeeklySalary)

    Exit Sub

28:
    MsgBox "There was a problem when performing the calculation"
End Sub
```

After creating the numeric label, you can ask the compiler to jump to it if a problem occurs. To do

this, type **On Error GoTo** followed by the numeric label. The compiler would still jump to it when appropriate. Here is an example:

```
Private Sub cmdCalculate_Click()
    On Error GoTo 28

    Dim HourlySalary As Double, WeeklyTime As Double
    Dim WeeklySalary As Double

    HourlySalary = Cdbl(txtHourlySalary)
    WeeklyTime = Cdbl(txtWeeklyTime)
    WeeklySalary = HourlySalary * WeeklyTime

    txtWeeklySalary = FormatNumber(WeeklySalary)

Exit Sub
28:
    MsgBox "There was a problem when performing the calculation"
End Sub
```

Notice that the numeric label works like the lettered label. In other words, before writing the **On Error GoTo** expression, you must have created the label. In reality, this is not a rule. You can ask the compiler to let you deal with the error one way or another. To do this, use the **On Error GoTo 0** (or **On Error GoTo -1**) expression. Here is an example:

```
Private Sub cmdCalculate_Click()
    On Error GoTo 0

    Dim HourlySalary As Double, WeeklyTime As Double
    Dim WeeklySalary As Double

    HourlySalary = Cdbl(txtHourlySalary)
    WeeklyTime = Cdbl(txtWeeklyTime)
    WeeklySalary = HourlySalary * WeeklyTime

    txtWeeklySalary = FormatNumber(WeeklySalary)
End Sub
```

In this case, if/when the error occurs, you must have a way to deal with it.

Resume the Code Flow

In every code we have explored so far, we anticipated that there could be a problem and we dealt with it. In most cases, after dealing with the error, you must find a way to continue with a normal flow of your program. In some other cases, you may even want to ignore the error and proceed as if everything were normal, or you don't want to bother the user with some details of the error.

After you have programmatically [deal](#) with an error, to resume with the normal flow of the program, you use the **Resume** operator. It presents many options.

After an error has occurred, to ask the compiler to proceed with the regular flow of the program, type the **Resume** keyword. Here is an example:

```
Private Sub cmdCalculate_Click()
    On Error GoTo ThereWasBadCalculation

    Dim HourlySalary As Double, WeeklyTime As Double
    Dim WeeklySalary As Double

    ' One of these two lines could produce an error, such as
    ' if the user types an invalid number
    HourlySalary = Cdbl(txtHourlySalary)
    WeeklyTime = Cdbl(txtWeeklyTime)

    ' If there was an error, the flow would jump to the label
    WeeklySalary = HourlySalary * WeeklyTime

Resume

    txtWeeklySalary = FormatNumber(WeeklySalary)

Exit Sub

ThereWasBadCalculation:
    MsgBox "There was a problem when performing the calculation"
End Sub
```

Notice that you can write the **Resume** operator almost anywhere. In reality, you should identify where the program would need to resume. Where else than after presenting the error message to the user? If you want the program to continue with an alternate value than the one that caused the problem, in the label section, type **Resume Next**. Here is an example:

```
Private Sub cmdCalculate_Click()
    On Error GoTo ThereWasBadCalculation

    Dim HourlySalary As Double, WeeklyTime As Double
    Dim WeeklySalary As Double

    ' One of these two lines could produce an error, such as
    ' if the user types an invalid number
    HourlySalary = Cdbl(txtHourlySalary)
    WeeklyTime = Cdbl(txtWeeklyTime)

    ' If there was an error, the flow would jump to the label
    WeeklySalary = HourlySalary * WeeklyTime
```

```
txtWeeklySalary = FormatNumber(WeeklySalary)
```

```
Exit Sub
```

```
ThereWasBadCalculation:
    MsgBox "There was a problem when performing the calculation"
    Resume Next
End Sub
```

We know that in our code, there was probably a problem, which is the reason we presented a message to the user. Then, when code resumes, where should the compiler go? After all, the problem was not solved. One way you can deal with the problem is to provide an alternative to what caused the problem, since you are supposed to know what type of problem occurred (in the next sections, we will analyze the types of problems that can occur). In the case of an arithmetic calculation, imagine we know that the problem was caused by the user typing an invalid number (such as typing a name where a number was expected). Instead of letting the program crash, we can provide a number as an alternative. The easiest number is 0.

Before asking the compiler to resume, to provide an alternative solution (a number in this case), you can re-initialize the variable that caused the error. Here is an example:

```
Private Sub cmdCalculate_Click()
    On Error GoTo ThereWasBadCalculation

    Dim HourlySalary As Double, WeeklyTime As Double
    Dim WeeklySalary As Double

    ' One of these two lines could produce an error, such as
    ' if the user types an invalid number
    HourlySalary = Cdbl(txtHourlySalary)
    WeeklyTime = Cdbl(txtWeeklyTime)

    ' If there was an error, the flow would jump to the label
    WeeklySalary = HourlySalary * WeeklyTime

    txtWeeklySalary = FormatNumber(WeeklySalary)

Exit Sub

ThereWasBadCalculation:
    MsgBox "There was a problem when performing the calculation"
    HourlySalary = 0
    Resume Next
End Sub
```

If there are many variables involved, as is the case for us, you can initialize each. Here an example:

```
Private Sub cmdCalculate_Click()
    On Error GoTo ThereWasBadCalculation

    Dim HourlySalary As Double, WeeklyTime As Double
    Dim WeeklySalary As Double

    ' One of these two lines could produce an error, such as
    ' if the user types an invalid number
    HourlySalary = Cdbl(txtHourlySalary)
    WeeklyTime = Cdbl(txtWeeklyTime)

    ' If there was an error, the flow would jump to the label
    WeeklySalary = HourlySalary * WeeklyTime

    txtWeeklySalary = FormatNumber(WeeklySalary)

Exit Sub

ThereWasBadCalculation:
    MsgBox "There was a problem when performing the calculation"
    HourlySalary = 0
    WeeklyTime = 0
    Resume Next
End Sub
```

Types of Error

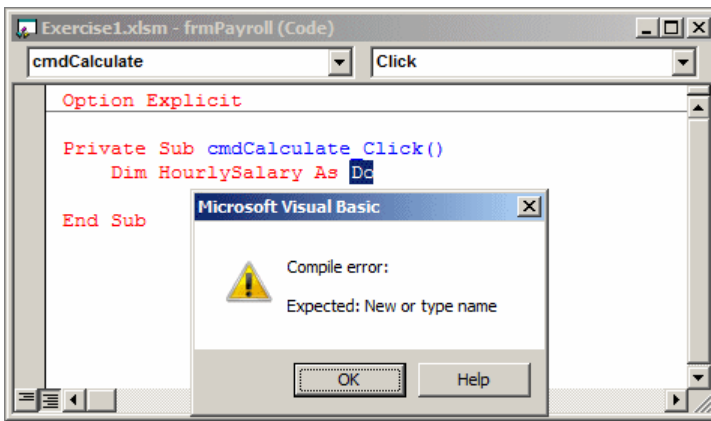
Introduction

In our introductions to errors, we mostly anticipated only problems related to arithmetic calculations. In reality, a program can face various categories of bad occurrences. The more problems you prepare for, the least phone calls and headaches you will have. Problems are divided in two broad categories.

Syntax Errors

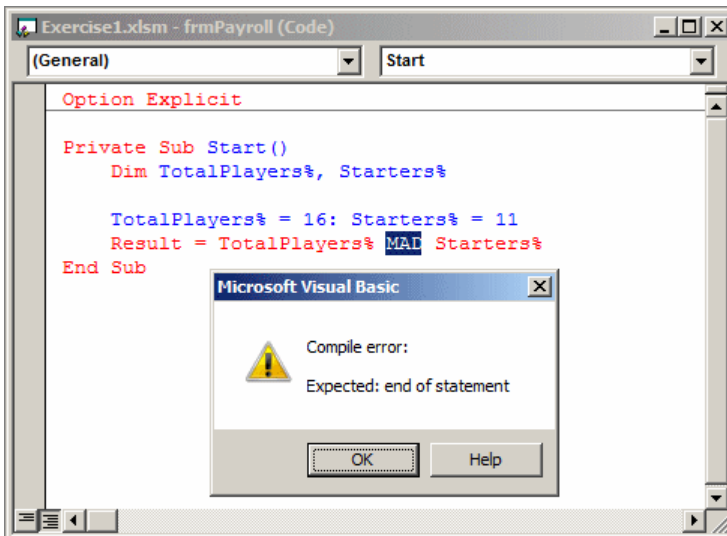
A syntax error occurs if your code tries to perform an operation that the VBA language does not allow. These errors are probably the easiest to locate because the Code Editor is configured to point them out at the time you are writing your code.

If you try typing or try inserting an operator or keyword in the wrong place on your code, the Code Editor would point it out. Here is an example:



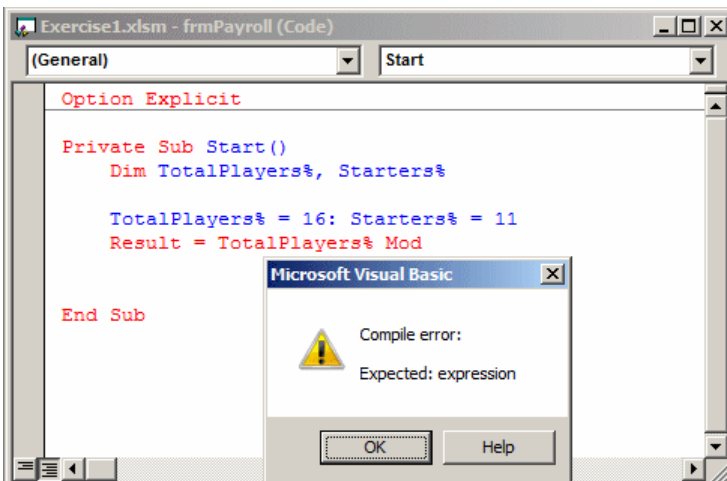
In this case, if you were trying to use the **Do** keyword instead of a data type (probably **Double** in this case), the Code Editor would show it right away. This type of error is pointed out for every keyword and operator you try to use.

Notice that, in the above example, we used a valid keyword but at the wrong time. If you mistype a keyword or an operator, you would receive an error. Fortunately, the Code Editor is equipped to know all keywords of the Visual Basic language. Consider the following example:



The programmer mistyped the **Mod** operator and wrote MAD instead.

If you forget to include a necessary factor in your code, you would get a syntax error. For example, if you are creating a binary arithmetic expression that expects a second operand after the operator, you would receive an error. Here is an example:



In this case, the programmer pressed Enter after the **Mod** operator, as if the expression was complete. This resulted in an error.

These are just a few types of syntax errors you may encounter. As mentioned already, if you work in Microsoft Visual Basic to write your code, most of these errors are easy to detect and fix.

Run-Time Errors

A run-time error occurs when your application tries to do something that the operating system does not allow. In some cases, only your application would crash (Microsoft Excel may stop working). In some other cases, the user may receive a more serious error. As its name indicates, a run-time error occurs when the program runs; that is, after you have created your application.

Fortunately, during the testing phase, you may encounter some of the errors so you can fix them

before distributing your application. Some other errors may not occur even if you test your application. They may occur to the users after you have distributed your application. For example, you can create a car rental application that is able to display pictures 100% of the time on your computer while locating them from the E: drive. Without paying attention, after distributing your application, the user's computer may not have an E: drive and, when trying to display the pictures, the application may crash.

Examples of run-time errors are:

- Trying to use computer memory that is not available
- Performing a calculation that the computer hardware (for example the processor) does not allow. An example is division by 0
- Trying to use or load a library that is not available or is not accessible, for any reason
- Performing an arithmetic operation on two incompatible types (such as trying to assign to an **Integer** variable the result of adding a string to a **Double** value)
- Using a loop that was not properly initialized
- Trying to access a picture not accessible. Maybe the path specified for the picture is wrong. Maybe your code gives the wrong extension to the file, even though the file exists
- Accessing a value beyond the allowable range. For example, using a **Byte** variable to assign a performed operation that produces a value the variable cannot hold

As you may imagine, because run-time errors occur after the application has been described as ready, some of these errors can be difficult to identify. Some other errors depend on the platform that is running the application (the operating system, the processor, the version of the application, the (available) memory, etc).

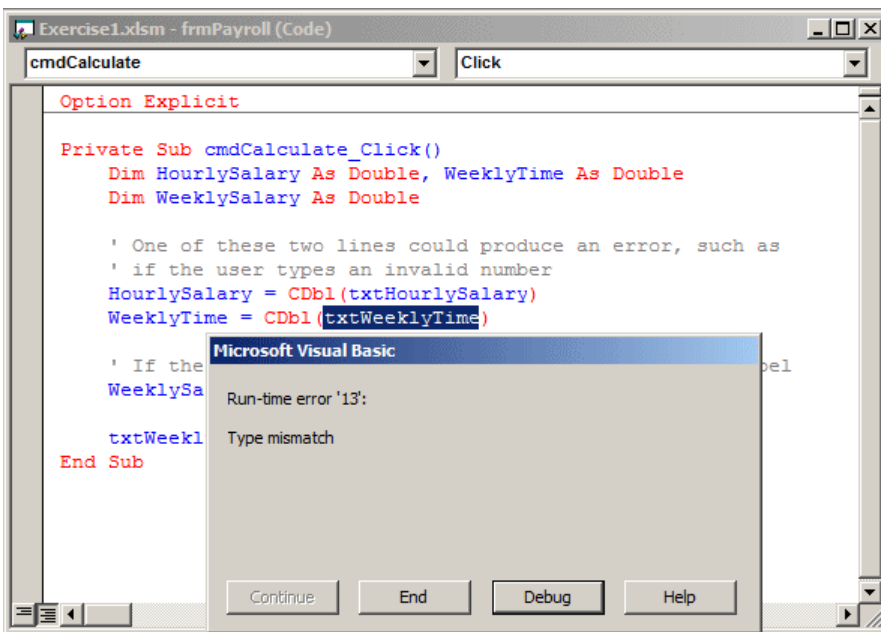
The Err Object

Introduction

To assist you with handling errors, the Visual Basic language provides a class named **Err**. You don't have to declare a variable for this class. An **Err** object is readily available as soon as you start working on VBA code and you can directly access its members.

The Error Number

As mentioned already, there are various types of errors that can occur to your program. To assist you with identifying them, the **Err** object is equipped with a property named **Number**. This property holds a specific number to most errors that can occur to your program. When your program runs and encounters a problem, it may stop and display the number of the error. Here is an example:



As you can see, this is error number 13. Because there are many types of errors, there are also many numbers, so much that we cannot review all of them. We can only mention some of them when we encounter them.

When a program runs, to find out what type of error occurred, you can question the **Number** property of the **Err** object to find out whether the error that has just occurred holds this or that number. To do this, you can use an **If...Then** conditional statement to check the number. You can then display the necessary message to the user. Here is an example:

```
Private Sub cmdCalculate_Click()
    On Error GoTo WrongValue

    Dim HourlySalary As Double, WeeklyTime As Double
    Dim WeeklySalary As Double
```

```

' One of these two lines could produce an error, such as
' if the user types an invalid number
HourlySalary = CDb1(txtHourlySalary)
WeeklyTime = CDb1(txtWeeklyTime)

' If there was an error, the flow would jump to the label
WeeklySalary = HourlySalary * WeeklyTime

txtWeeklySalary = FormatNumber(WeeklySalary)

Exit Sub

WrongValue:
If Err.Number = 13 Then
    MsgBox "You typed an invalid value"
    HourlySalary = 0
    WeeklyTime = 0
    Resume Next
End If
End Sub

```

The Error Message

As mentioned already, there are many errors and therefore many numbers held by the **Number** property of the **Err** object. As a result, just knowing an error number can be vague. To further assist you with decrypting an error, the **Err** object provides a property named **Description**. This property holds a (usually short) message about the error number. This property works along with the **Number** property holding the message corresponding to the **Number** property.

To get the error description, after inquiring about the error number, you can get the equivalent **Description** value. Here is an example:

```

Private Sub cmdCalculate_Click()
    On Error GoTo WrongValue

    Dim HourlySalary As Double, WeeklyTime As Double
    Dim WeeklySalary As Double

    ' One of these two lines could produce an error, such as
    ' if the user types an invalid number
    HourlySalary = CDb1(txtHourlySalary)
    WeeklyTime = CDb1(txtWeeklyTime)

    ' If there was an error, the flow would jump to the label
    WeeklySalary = HourlySalary * WeeklyTime

    txtWeeklySalary = FormatNumber(WeeklySalary)

    Exit Sub

WrongValue:
If Err.Number = 13 Then
    MsgBox Err.Description
    HourlySalary = 0
    WeeklyTime = 0
    Resume Next
End If
End Sub

```

In some cases, the error message will not be explicit enough, especially if a user simply reads it to you over the phone. The alternative is to create your own message in the language you easily understand, as we did earlier. If you want, you can also display a message that combines both the error description and your own message. Here is an example:

```

Private Sub cmdCalculate_Click()
    On Error GoTo WrongValue

    Dim HourlySalary As Double, WeeklyTime As Double
    Dim WeeklySalary As Double

    ' One of these two lines could produce an error, such as
    ' if the user types an invalid number
    HourlySalary = CDb1(txtHourlySalary)
    WeeklyTime = CDb1(txtWeeklyTime)

    ' If there was an error, the flow would jump to the label
    WeeklySalary = HourlySalary * WeeklyTime

    txtWeeklySalary = FormatNumber(WeeklySalary)


    Exit Sub

WrongValue:
If Err.Number = 13 Then
    MsgBox Err.Description & " : The value you typed cannot be accepted."
    HourlySalary = 0
    WeeklyTime = 0
    Resume Next
End If
End Sub

```

❖ Practical Learning: Handling an Error

1. Make sure the TimeSheet worksheet is displaying.
On the Ribbon, click Developer

2. In the Controls section, click Insert and, in the Form Controls section, click Button (Form Control) 
3. Click an empty on the TimeSheet worksheet
4. On the Assign Macro dialog box, set the Macro Name to btnSubmitTimeSheet_Click
5. Click New
6. Implement the event as follows:

```

Sub btnSubmitTimeSheet_Click()
    On Error GoTo btnSubmitTimeSheet_Error

    ' This variable will help us check the rows
    Dim CurrentRow As Integer
    ' This variable will get the employee # from the payroll
    Dim PayrollEmployeeNumber As String
    ' This variable will get the employee # from the time sheet
    Dim TimeSheetEmployeeNumber As String
    ' These 2 variables will get the date values from the time sheet
    Dim StartDate As Date, EndDate As Date

    ' These variables represent the time worked from the time sheet
    Dim Week1Monday As Double, Week1Tuesday As Double
    Dim Week1Wednesday As Double, Week1Thursday As Double
    Dim Week1Friday As Double, Week1Saturday As Double
    Dim Week1Sunday As Double, Week2Monday As Double
    Dim Week2Tuesday As Double, Week2Wednesday As Double
    Dim Week2Thursday As Double, Week2Friday As Double
    Dim Week2Saturday As Double, Week2Sunday As Double

    ' We will check the records starting at Row 8
    CurrentRow = 8

    ' Get the employee number from the time sheet
    TimeSheetEmployeeNumber = Worksheets("TimeSheet").Range("C6")

    ' Get the starting date from the time sheet
    StartDate = CDate(Worksheets("TimeSheet").Range("C8"))
    ' Add 2 weeks to the starting date
    EndDate = DateAdd("d", 13, StartDate)

    ' Get the time worked for each day
    Week1Monday = CDb(Worksheets("TimeSheet").Range("C11"))
    Week1Tuesday = CDb(Worksheets("TimeSheet").Range("D11"))
    Week1Wednesday = CDb(Worksheets("TimeSheet").Range("E11"))
    Week1Thursday = CDb(Worksheets("TimeSheet").Range("F11").Value)
    Week1Friday = CDb(Worksheets("TimeSheet").Range("G11").Value)
    Week1Saturday = CDb(Worksheets("TimeSheet").Range("H11").Value)
    Week1Sunday = CDb(Worksheets("TimeSheet").Range("I11").Value)
    Week2Monday = CDb(Worksheets("TimeSheet").Range("C12").Value)
    Week2Tuesday = CDb(Worksheets("TimeSheet").Range("D12").Value)
    Week2Wednesday = CDb(Worksheets("TimeSheet").Range("E12").Value)
    Week2Thursday = CDb(Worksheets("TimeSheet").Range("F12").Value)
    Week2Friday = CDb(Worksheets("TimeSheet").Range("G12").Value)
    Week2Saturday = CDb(Worksheets("TimeSheet").Range("H12").Value)
    Week2Sunday = CDb(Worksheets("TimeSheet").Range("I12").Value)

    ' Get ready to check each employee number from the payroll
    Do
        ' To process a payroll, an employee from the Accounting department
        ' enters an employee's employee number
        ' Get that employee number from the payroll
        PayrollEmployeeNumber = Worksheets("Payroll").Cells(CurrentRow, 8).Value

        ' Check all records from the Payroll
        ' If you find an empty cell in the columns for the employee number,
        ' this means that there is no record in that row.
        ' If there is no record, ...
        If PayrollEmployeeNumber = "" Then
            ' ... fill out that record with values from the time sheet
            Worksheets("Payroll").Cells(CurrentRow, 2) = TimeSheetEmployeeNumber
            Worksheets("Payroll").Cells(CurrentRow, 3) = StartDate
            Worksheets("Payroll").Cells(CurrentRow, 4) = EndDate
            Worksheets("Payroll").Cells(CurrentRow, 5) = Week1Monday
            Worksheets("Payroll").Cells(CurrentRow, 6) = Week1Tuesday
            Worksheets("Payroll").Cells(CurrentRow, 7) = Week1Wednesday
            Worksheets("Payroll").Cells(CurrentRow, 8) = Week1Thursday
            Worksheets("Payroll").Cells(CurrentRow, 9) = Week1Friday
            Worksheets("Payroll").Cells(CurrentRow, 10) = Week1Saturday
            Worksheets("Payroll").Cells(CurrentRow, 11) = Week1Sunday
            Worksheets("Payroll").Cells(CurrentRow, 12) = Week2Monday
            Worksheets("Payroll").Cells(CurrentRow, 13) = Week2Tuesday
            Worksheets("Payroll").Cells(CurrentRow, 14) = Week2Wednesday
            Worksheets("Payroll").Cells(CurrentRow, 15) = Week2Thursday
            Worksheets("Payroll").Cells(CurrentRow, 16) = Week2Friday
            Worksheets("Payroll").Cells(CurrentRow, 17) = Week2Saturday
            Worksheets("Payroll").Cells(CurrentRow, 18) = Week2Sunday
            Exit Do
        End If

        ' If you found a record, increase the row count by 1 ...
        CurrentRow = CurrentRow + 1
        ' ... and check the next record
        ' Continue until the next 93 records
    Loop While CurrentRow <= 93

```

```
' If there was a problem, get out of this procedure
Exit Sub
```

```
btnSubmitTimeSheet_Error:
' If there was an error, check what type of error this was.
' If the error is 13, it means the user entered a bad value.
' Let the user know
If Err.Number = 13 Then
    MsgBox "You entered an invalid value." & vbCrLf & _
        "Check all the values on your time sheet."
End If

Resume Next
End Sub
```

- Close Microsoft Visual Basic
- Adjust the button to your liking

	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
Week 1							
Week 2							

Submit Time Sheet

- Process a timesheet and click the button

	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
Week 1	8.00	8.00	8.50	8.50	8.50	0.00	0.00
Week 2	6.00	6.50	8.00	6.00	8.00	0.00	0.00

Submit Time Sheet

- Click the Payroll tab to see the result

The Source of the Error

Most of the time, you will know what caused an error, since you will have created the application. The project that causes an error is known as the source of error. In some cases, you may not be able to easily identify the source of error. To assist you with this, the **Err** object is equipped with a property named **Source**.

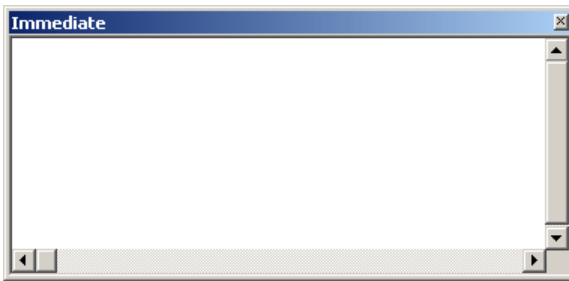
To identify the application that caused an error, you can inquire about the value of this property.

Debugging and the Immediate Window

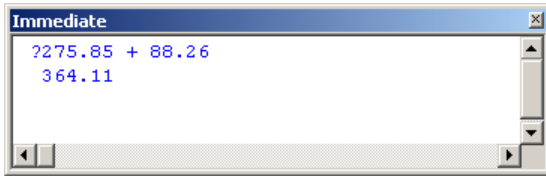
The Immediate Window

Debugging consists of examining and testing portions of your code or parts of your application to identify problems that may occur when somebody is using your database. Microsoft Visual Basic provides as many tools as possible to assist you with this task.

The Immediate window is an object you can use to test functions and expressions. To display the Immediate window, on the main menu of Microsoft Visual Basic, you can click View -> Immediate Window. It's a habit to keep the Immediate window in the bottom section of the Code Editor but you can move it from there by dragging its title bar:



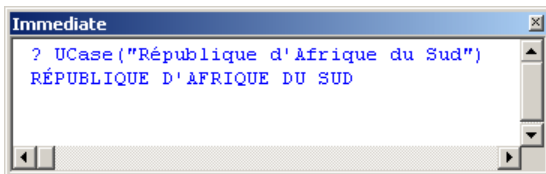
Probably the simplest action you can perform in the Immediate window consists of testing an expression. For example, you can write an arithmetic operation and examine its result. To do this, in the Immediate window, type the question mark "?" followed by the expression and press Enter. Here is an example that tests the result of $275.85 + 88.26$:



One of the most basic actions you can perform in the Immediate window consists of testing a built-in function. To do this, type ? followed by the name of the function and its arguments, if any. For example, to test the **UCase** function, in the Immediate window, you could type:

```
? UCase("République d'Afrique du Sud")
```

After typing the function and pressing Enter, the result would display in the next line:



The Debug Object

The Immediate window is recognized in code as the **Debug** object. To programmatically display something, such as a string, in the Immediate window, the Debug object provides the **Print** method. The simplest way to use it consist of passing it a string. For example, imagine you create a button on a form, you name it cmdTestFullName and initialize it with a string. Here is an example of how you can display that string in the Immediate window:

```
Private Sub cmdTestFullName_Click()
    Dim strFullName$

    strFullName$ = "Daniel Ambassa"
    Debug.Print strFullName$
End Sub
```

When you click the button, the Immediate window would display the passed string:



In the same way, you can create a more elaborate expression and test its value in the Immediate window. You can also pass a value, such as a date, that can easily be converted to a string.



File Processing

Creating a File

Introduction

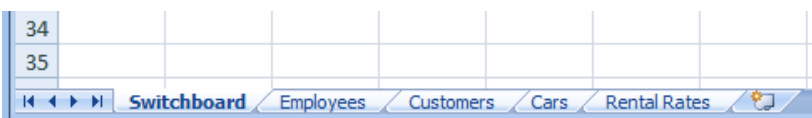
In Microsoft Excel, instead of a worksheet, you could create a form-based application that your users would use. If (since) you have already used Windows-based applications, you are surely familiar with data entry on a form, in which case you use Windows controls.

File processing is the ability to store the values of a document in the computer so you can retrieve such values another time.

File processing is the ability to save values from an application and be able to get those values back when needed. The VBA language supports file processing.

❖ Practical Learning: Introducing File Processing

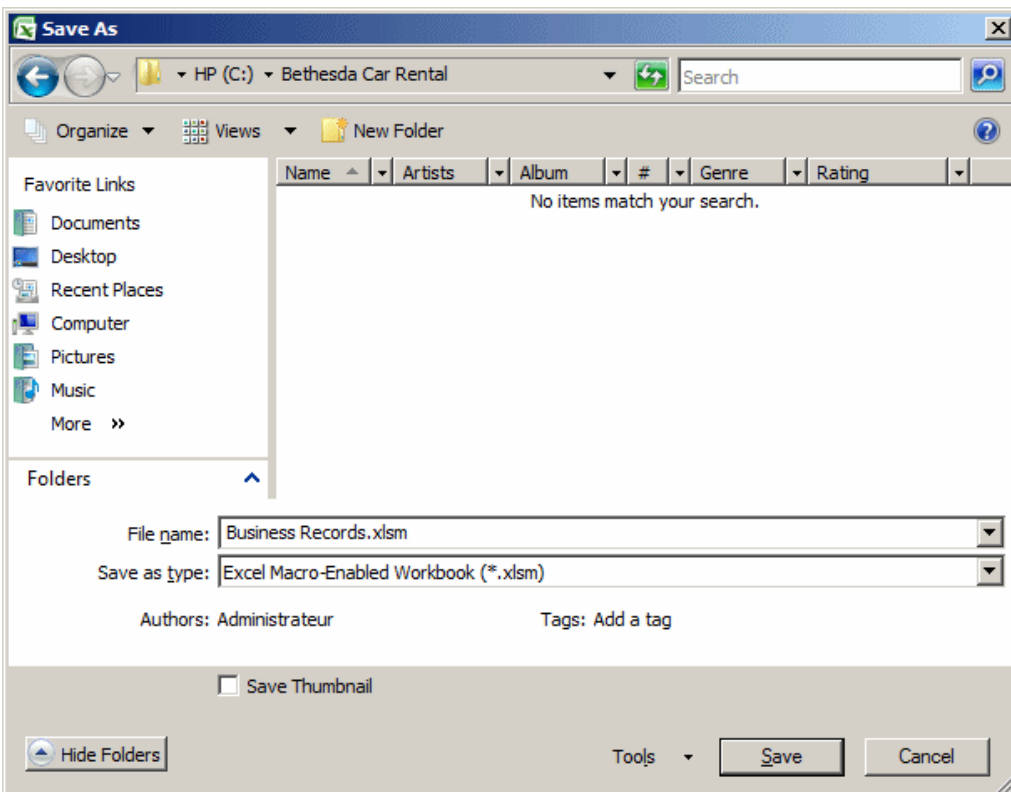
1. Start Microsoft Excel
2. Double-click Sheet1, type **Switchboard**
3. Double-click Sheet2 and type **Employees**
4. Double-click Sheet3 and type **Customers**
5. Click the next sheet tab (the Insert Worksheet)
6. Double-click the new sheet tab and type **Cars**
7. Click the next sheet tab (the Insert Worksheet)
8. Double-click the new sheet tab and type **Rental Rates**
9. Click the Switchboard tab
10. Press and hold Shift
11. Click the Rental Rates tab
12. Release Shift



13. Click Cell B2 and type **Bethesda Car Rental**
14. Click the Enter button
15. Format it as you see fit:



16. Click the Employees sheet tab
17. To save the workbook, press Ctrl + S
18. In the top combo box, select the C drive (or any drive you want)
19. Click the Create New Folder (Windows XP) or New Folder (Windows Vista) button
20. Type **Bethesda Car Rental** as the name of the new folder and press Enter
21. Make sure the new folder is selected.
Change the file name to **Business Records**
22. In the Save As Type combo box, select Excel Macro-Enabled Workbook



23. Click Save
24. In the Employees sheet tab, click Cell B6 and type **Employee #**
25. **Create a list of employees**
26. Click Cell E7, type **=D7 & ", " & C7** and click the Enter button
27. Drag its AutoFill down to Cell E13

Employee	First	Last	Full Name	Title	Hourly
62-845	Patricia	Katts	Katts, Patricia	General Manager	42.25
92-303	Henry	Larson	Larson, Henry	Sales Representative	12.5
25-947	Gertrude	Monay	Monay, Gertrude	Sales Representative	14.05
73-947	Helene	Sandt	Sandt, Helene	Intern	8.85
40-508	Melanie	Karron	Karron, Melanie	Sales Representative	12.75
22-580	Ernest	Chisen	Chisen, Ernest	Sales Manager	22.95
20-308	Melissa	Roberts	Roberts, Melissa	Administrative Assistant	15.45

28. Click the Customers sheet tab
29. Click Cell B6 and type **Driver's Lic. #**
30. **Create a list of customers**

Driver's Lic. #	State	Full Name	Address	City	ZIP Code
M-505-862-575	MD	Lynda Melman	4277 Jamison Avenue	Silver Spring	20904
379-82-7397	DC	John Villard	108 Hacken Rd NE	Washington	20012
J-938-928-274	MD	Chris Young	8522 Aulage Street	Rockville	20852
497-22-0614	PA	Pamela	12075 Famina Rd	Blain	17006
922-71-8395	VA	Helene	806 Hyena Drive	Alexandria	22231
C-374-830-422	MD	Hermine	6255 Old Georgia Ave	Silver Spring	20910
836-55-2279	NY	Alan Pastore	4228 Talion Street	Amherst	14228
397-59-7487	TN	Phillis Buster	724 Cranston Circle	Knoxville	37919
115-80-2957	FL	Elmus	808 Rasters Ave	Orlando	32810
294-90-7744	VA	Helena	10448 Great Pollard	Arlington	22232

31. Click the Cars sheet tab
32. Click Cell B6 and type **Tag Number**
33. **Create a list of cars**
34. Click the Rental Rates sheet tab
35. Click Cell B6 and type **Category**
36. Complete the table with the following values:

Category	Daily	Weekly	Monthly	Weekend
Economy	35.95	32.75	28.95	24.95
Compact	39.95	35.75	32.95	28.95
Standard	45.95	39.75	35.95	32.95
Full Size	49.95	42.75	38.95	35.95
Mini Van	55.95	50.75	45.95	42.95
SUV	55.95	50.75	45.95	42.95
Truck	42.75	38.75	35.95	32.95
Van	69.95	62.75	55.95	52.95

38. On the [Ribbon](#), click Developer



39. In the Code section of the Ribbon, click Visual Basic

40. On the main menu of Microsoft Visual Basic, click Insert -> UserForm

41. If the Properties window is not available, right-click the form and click Properties. In the Properties window, click (Name) and type **frmNewRentalOrder**

42. Click Caption and type **Bethesda Car Rental - Order Processing - New Rental Order**

43. Design the form as follows:

Control	(Name)	Caption/Text	Other Properties
Label		Processed By	BackColor: &H00808080& BorderColor: &H00000000& ForeColor: &H00FFFFFF&
Label		Car Selected	BackColor: &H00808080& BorderColor: &H00000000& ForeColor: &H00FFFFFF&
Label		Employee #:	
Text Box	txtEmployeeNumber		
Text Box	txtEmployeeName		
Label		Tag Number:	
Text Box	txtTagNumber		
Label		Condition:	
Combo Box	cbxCarConditions		
Label		Processed For	BackColor: &H00808080& BorderColor: &H00000000& ForeColor: &H00FFFFFF&
Label		Make:	
Text Box	txtMake		
Label		Driver's Lic. #:	
Text Box	txtDrvLicenseNbr		
Label		Model:	
Text Box	txtModel		
Label		Name:	
Text Box	txtCustomerName		
Label		Year:	
Text Box	txtCarYear		TextAlign: 3 - fmTextAlignRight

Label		Tank Level:	
Combo Box	cbxTankLevels		
Label		Address:	
Text Box	txtAddress		
Label		Mileage Start:	
Text Box	txtMileageStart		TextAlign: 3 - fmTextAlignRight
Label		Mileage End:	
Text Box	txtMileageEnd		TextAlign: 3 - fmTextAlignRight
Label		City:	
Text Box	txtCity		
Label		Order Evaluation	BackColor: &H00808080& BorderColor: &H00000000& ForeColor: &H00FFFFFF&
Label		State:	
Text Box	txtState		
Label		ZIP Code:	
Text Box	txtZIPCode		
Label		Rate Applied:	
Text Box	txtRateApplied	24.95	TextAlign: 3 - fmTextAlignRight
Label		Tax Rate:	
Text Box	txtTaxRate	5.75	TextAlign: 3 - fmTextAlignRight
Label			BackColor: &H00808080& BorderColor: &H00000000& ForeColor: &H00FFFFFF&
Label		Days:	
Text Box	txtDays	0	TextAlign: 3 - fmTextAlignRight
Label		Tax Amount:	
Text Box	txtTaxAmount	0.00	TextAlign: 3 - fmTextAlignRight
Label		Start Date:	
Text Box	txtStartDate		
Label		End Date:	
Text Box	txtEndDate		
Label		Sub-Total:	
Text Box	txtSubTotal	0.00	TextAlign: 3 - fmTextAlignRight
Label		Order Total:	
Text Box	txtOrderTotal	0.00	TextAlign: 3 - fmTextAlignRight
Label		Receipt #:	
Text Box	txtReceiptNumber		
Command Button	cmdSave	Save	
Command Button	cmdReset	Reset / New Rental Order	

44. Right-click the Employee Number text box and click View Code

45. In the Procedure combo box, select Enter

46. Implement the event as follows:

```
Private Sub txtEmployeeNumber_Enter()
    REM When the Employee # has focus, activate the Employees worksheet
    Worksheets(2).Activate
End Sub
```

47. In the Procedure combo box, select Exit

48. Implement the event as follows:

```
Private Sub txtEmployeeNumber_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    On Error GoTo txtEmployeeNumber_Error

    ' Check if the user left the Employee Number empty
    If txtEmployeeNumber.Text = "" Then
        ' If so, put leave the Employee Name empty
        txtEmployeeName.Text = ""
    Else
        ' If the user entered a valid employee #, use the Microsoft Excel's
        ' VLOOKUP() function to get the corresponding employee name
        ' We are using the range of cells from B7 to E13 but you can use a
        ' range of your choice as long as it contains the employees records
        txtEmployeeName.Text = _
            Application.WorksheetFunction.VLookup(txtEmployeeNumber.Text, _
                Worksheets(2).Range("B7:E13"), 4, False)
    End If

Exit Sub
```

```

txtEmployeeNumber_Error:
' If the user entered an invalid employee #, put Unknown in the name
If Err.Number = 1004 Then
    txtEmployeeNumber.Text = ""
    txtEmployeeName.Text = "Unknown clerk"
End If
End Sub

```

49. In the Object combo box, select txtTagNumber

50. In the Procedure combo box, select Enter

51. Implement the event as follows:

```

Private Sub txtTagNumber_Enter()
    Worksheets(4).Activate
End Sub

```

52. In the Procedure combo box, select Exit

53. Implement the event as follows:

```

Private Sub txtTagNumber_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    On Error GoTo txtTagNumber_Error

    ' Check if the user left the Tag Number text box empty
    If txtTagNumber.Text = "" Then
        ' If so, leave the car information empty
        txtTagNumber.Text = ""
        txtMake.Text = ""
        txtModel.Text = ""
        txtCarYear.Text = ""
    Else
        ' If the user entered a valid Tag Number, use the Microsoft Excel's
        ' VLOOKUP() function to get the corresponding car information
        txtMake.Text = _
            Application.WorksheetFunction.VLookup(txtTagNumber.Text, _
                Worksheets(4).Range("B6:I26"), 2, False)
        txtModel.Text = _
            Application.WorksheetFunction.VLookup(txtTagNumber.Text, _
                Worksheets(4).Range("B6:I26"), 3, False)
        txtCarYear.Text = _
            Application.WorksheetFunction.VLookup(txtTagNumber.Text, _
                Worksheets(4).Range("B6:I26"), 4, False)
    End If

    Exit Sub

txtTagNumber_Error:
' If the user entered an invalid tag #, leave the Tag Number empty
If Err.Number = 1004 Then
    txtTagNumber.Text = ""
    txtMake.Text = ""
    txtModel.Text = ""
    txtCarYear.Text = ""
End If
End Sub

```

54. In the Object combo box, select txtDrvLicenseNbr

55. In the Procedure combo box, select Enter

56. Implement the event as follows:

```

Private Sub txtDrvLicenseNbr_Enter()
    Worksheets(3).Activate
End Sub

```

57. In the Procedure combo box, select Exit

58. Implement the event as follows:

```

Private Sub txtDrvLicenseNbr_Exit(ByVal Cancel As MSForms.ReturnBoolean)
    On Error GoTo txtDrvLicenseNbr_Error

    If txtDrvLicenseNbr.Text = "" Then
        txtCustomerName.Text = ""
        txtAddress.Text = ""
        txtCity.Text = ""
        txtState.Text = ""
        txtZIPCode.Text = ""
    Else
        txtCustomerName.Text = _
            Application.WorksheetFunction.VLookup(txtDrvLicenseNbr.Text, _

```

```

Worksheets(3).Range("B6:I26"), 2, False)
txtAddress.Text = _
    Application.WorksheetFunction.VLookup(txtDrvLicenseNbr.Text, _
        Worksheets(3).Range("B6:I26"), 3, False)
txtCity.Text = _
    Application.WorksheetFunction.VLookup(txtDrvLicenseNbr.Text, _
        Worksheets(3).Range("B6:I26"), 4, False)
txtState.Text = _
    Application.WorksheetFunction.VLookup(txtDrvLicenseNbr.Text, _
        Worksheets(3).Range("B6:I26"), 5, False)
txtZIPCode.Text = _
    Application.WorksheetFunction.VLookup(txtDrvLicenseNbr.Text, _
        Worksheets(3).Range("B6:I26"), 6, False)
End If

Exit Sub

```

```

txtDrvLicenseNbr_Error:
If Err.Number = 1004 Then
    txtDrvLicenseNbr.Text = ""
    txtCustomerName.Text = ""
    txtAddress.Text = ""
    txtCity.Text = ""
    txtState.Text = ""
    txtZIPCode.Text = ""
End If
End Sub

```

59. In the Object combo box, select txtRateApplied
60. In the Procedure combo box, select Enter
61. Implement the event as follows:

```

Private Sub txtRateApplied_Enter()
    Worksheets(5).Activate
End Sub

```

62. In the Object combo box, select UserForm
63. In the Procedure combo box, select Activate
64. Implement the event as follows:

```

Private Sub ResetRentalOrder()
    Dim strRandomNumber As String

    ' Fill the Conditions combo box
    cbxCarConditions.AddItem "Needs Repair"
    cbxCarConditions.AddItem "Drivable"
    cbxCarConditions.AddItem "Excellent"

    ' Fill the Tank Level combo box
    cbxTankLevels.AddItem "Empty"
    cbxTankLevels.AddItem "1/4 Empty"
    cbxTankLevels.AddItem "1/2 Full"
    cbxTankLevels.AddItem "3/4 Full"
    cbxTankLevels.AddItem "Full"

    ' For a receipt number, we will create a random number
    strRandomNumber = CStr(CInt(Rnd * 9))
    strRandomNumber = strRandomNumber & CStr(CInt(Rnd * 9))
    strRandomNumber = strRandomNumber & CStr(CInt(Rnd * 9))
    strRandomNumber = strRandomNumber & CStr(CInt(Rnd * 9))
    strRandomNumber = strRandomNumber & CStr(CInt(Rnd * 9))
    strRandomNumber = strRandomNumber & CStr(CInt(Rnd * 9))
    txtReceiptNumber = strRandomNumber
    ' In the real world, you would check the list of files
    ' in the Bethesda Car Rental folder. You would then get the
    ' name of the last file, or the highest receipt number. You
    ' would then increase this number by 1, and use that as the
    ' new receipt number

    txtEmployeeNumber.Text = ""
    txtEmployeeName.Text = ""
    txtDrvLicenseNbr.Text = ""
    txtCustomerName.Text = ""
    txtAddress.Text = ""
    txtCity.Text = ""
    txtState.Text = ""
    txtZIPCode.Text = ""
    txtStartDate.Text = ""
    txtEndDate.Text = ""
    txtTagNumber.Text = ""
    cbxCarConditions.Text = "Excellent"
    txtMake.Text = ""

```



```

txtModel.Text = ""
txtCarYear.Text = ""
cbxTankLevels.Text = ""
txtMileageStart.Text = "0"
txtMileageEnd.Text = "0"
txtRateApplied.Text = "24.95"
txtTaxRate.Text = "5.75"
txtDays.Text = "0"
txtTaxAmount.Text = "0.00"
txtSubTotal.Text = "0.00"
txtOrderTotal.Text = "0.00"
txtNotes.Text = ""

' Display today's date in the date text boxes
txtStartDate = Date
txtEndDate = Date
End Sub

Private Sub UserForm_Activate()
    Call ResetRentalOrder
End Sub

```

65. In the Object combo box, select cmdReset

66. Implement the Click event as follows:

```

Private Sub cmdReset_Click()
    Call ResetRentalOrder
End Sub

```

67. On the Standard toolbar, click the Save button

68. Return to Microsoft Excel and click the Switchboard tab sheet

69. In the Developer tab of the Ribbon, in the Controls section, click Insert

70. In the ActiveX Controls section, click Command Button

71. Click the worksheet

72. Right-click the new button and click Properties

73. In the properties window, change the following characteristics

(Name): **cmdCreateRentalOrder**

Caption: **Create New Rental Order**

74. Right-click the button and click View Code

75. Implement the event as follows:

```

Private Sub cmdCreateRentalOrder_Click()
    frmNewRentalOrder.Show
End Sub

```

76. Press Ctrl + S to save

File Creation

Before performing file processing, the first action you must perform consists of creating a file. To support file creation, the VBA provides a procedure named **Open**. Its syntax is:

```
Open pathname For Output [Access access] [lock] As [#]filename [Len=reclength]
```

The **Open** statement takes many factors, some are required and others are not. The **Open** (the name of the procedure) word, the **For Output** expression, and the **As #** expression are required.

The first argument, *pathname*, is required. This is a string that can be the name of the file. The file can have an extension or not. Here is an example:

```
Open "example.dat"
```

If you specify only the name of the file, it would be considered in the same folder where the current workbook is (the workbook that was opened when you called this statement). If you want, you can provide a complete path for the file. This would include the drive, the (optional) folder(s), up to the name of the file, with or without extension.

Besides the name of the file or its path, the *mode* factor is required. This factor specifies the actual action you want to perform, such as creating a new file or only opening an existing one. This factor can be one of the following keywords:

- **Output**: The file will be created and ready to receive (normal) values
- **Binary**: The file will be created and ready to receive values in binary format (as combinations of 1s and 0s)
- **Append**: If the file exists already, it will be opened and new values can be added to the end

Here is an example of creating a file:

```
Private Sub cmdSave_Click()
    Open "example.dat" For Output As #1
End Sub
```

The *access* factor is optional. It specifies what types of actions will be performed in the file, such as writing values to it or only reading existing values. This factor can have one of the following values:

- **Write:** After a new file has been created, new values will be written to it
- **Read Write:** When a new file has been created or an existing file has been opened, values can be read from it or written to it

If you decide to specify the *access* factor, precede its value with the **Access** keyword.

The *lock* factor is optional. It indicates how the processor should behave while the file is being used. Its possible values are:

- **Shared:** Other applications (actually called processes) can access this file while the current application is accessing it
- **Lock Write:** Do not allow other applications (processes) to access this file while the current application (process) is writing to it
- **Lock Read Write:** Do not allow other applications (processes) to access this file while the current application (process) is using it

On the right side of #, type a number, for the *filenumber* factor, between 1 and 511. If you are working on one file, use the number 1. If you are working on many files, you should use an incremental number. If you have not been keeping track of the number or you get confused at one time, to know the next number you can use, call the **FreeFile()** function, which returns the next available number in the sequence.

The *reclength* factor is optional. If the file was opened, this factor specifies the length of the record that was read.

Closing a File

When you create a file and start using it, or after opening a file and while you are using it, it uses memory and consumes (or can be consuming) memory (which could be significant). When you have finished using the file, you should free the memory it was using and release the resources it was consuming. To assist you with this, the VBA provides a procedure named **Close**. Its syntax is:

```
Close [filenumberlist]
```

The *filenumberlist* factor is the *filenumber* you would have previously used to create or open the file.

Here is an example of closing a file:

```
Private Sub cmdSave_Click()
    Open "example.dat" For Output As #1

    Close #1
End Sub
```

Printing to a File

After creating a file, you may want to write values to it. To support this, the VBA provides two procedures. One of them is called **Print** and its syntax is:

```
Print #filenumber, [outputlist]
```

The **Print** statement takes two factors but only the first is required.

The *filenumber* factor is the *filenumber* you would have used to create the file. The *filenumber* is followed by a comma.

The *outputlist* factor can be made of 0, 1 or more parts. Because it is optional, if you do not want to write a value to the file, leave this part empty. If you want to write a value, type a comma after the *filenumber* factor and follow these rules:

- If you want to start the value with empty spaces, use the **Spc()** function and pass an integer (in the parentheses) that represents the number of empty spaces. For example **Spc(4)** would include 4 empty spaces. This factor is optional, which means you can omit it
- Instead of a specific number of empty spaces, you can let the operating system specify a built-in number of empty spaces. To do this, call the **Tab()** function as part of your *outputlist* factor. The **Tab()** function specifies the number of columns to include before the value. The **Tab()** function can be more useful if you are concerned with the alignment of the value(s)

you will write in the file.

This factor is optional, which means you can omit it

- To write a string, include it in double-quotes
- To write a number, whether an integer, a float, or a double, simply include the number normally
- To write a Boolean value, type it as True or False
- To write a date or time value, type it between # and # and follow the rules of dates or times of your language such as US English
- To write a null value, type **Null**

Here is an example of writing some values:

```
Private Sub cmdSave_Click()
    Open "Employee.txt" For Output As #1

    Print #1, "James"
    Print #1, "Larenz"
    Print #1, True
    Print #1, #12/08/2008#

    Close #1
End Sub
```

Instead of writing one value per line, you can write more than one value with one statement. To do this, separate them with either a semi-colon or an empty space. Here is an example:

```
Private Sub cmdSave_Click()
    Open "Employee.txt" For Output As #1

    REM The values are separated by a semi-colon
    Print #1, "James"; "Larenz"
    REM The values are separated by an empty space
    Print #1, True #12/08/2008#

    Close #1
End Sub
```

Writing to a File

Besides the **Print** procedure, the VBA also provides a procedure named **Write** that can be used to write one or more values to a file. The syntax of the **Write** statement is the same as that of **Print**:

```
Write #filenumber, [outputlist]
```

The *filenumber* factor is required. It must be the *filenumber* specified when creating the file.

The *outputlist* factor is optional. If you want to skip it, type a comma after the *filenumber* and end the **Write** statement. In this case, an empty line would be written to the file. To write the values to the file, follow these rules:

- To start the value with empty spaces, call the **Spc()** function and pass a number that represents the number of empty spaces.
This factor is optional, which means you can omit it
- To start the value with a specific number of columns, call the **Tab()** function and pass the number of columns as argument.
This factor is optional, which means you can omit it
- To write a string, include it in double-quotes
- To write a number, include it normally
- To write a Boolean value, type it as **#TRUE#** or **#FALSE#**
- To write a null value, type **#NULL#**
- To write a date or time value, type it between # and #

Here is an example of writing some values:

```
Private Sub cmdSave_Click()
    Open "Employee.txt" For Output As #1

    Write #1, "James"
    Write #1, "M"
    Write #1, "Larenz"
    Write #1, #12/08/2008#
    Write #1, 24.50
    Write #1, True

    Close #1
End Sub
```

```
Private Sub cmdSave_Click()  
    Open "Employee.txt" For Output As #1  
  
    REM The values are separated by a semi-colon  
    Write #1, "James"; "M"; "Larenz"  
    REM The values are separated by a comma  
    Write #1, #12/08/2008#, 24.50  
    Write #1, True  
  
    Close #1  
End Sub
```

❖ Practical Learning: Saving a File

1. Display the form
2. Double-click the Save button
3. Implement its Click event as follows:

```
Private Sub cmdSave_Click()  
    On Error GoTo cmdSave_Error  
  
    Rem Make sure the user enters a valid employee number  
    If txtEmployeeNumber.Text = "" Then  
        MsgBox "You must enter a valid employee number."  
        Exit Sub  
    End If  
  
    Rem Make sure the user enters a valid car tag number  
    If txtTagNumber.Text = "" Then  
        MsgBox "You must enter a valid tag number."  
        Exit Sub  
    End If  
  
    Rem Make sure the user enters a valid customer  
    If txtDrvLicenseNbr.Text = "" Then  
        MsgBox "You must specify a valid car."  
        Exit Sub  
    End If  
  
    Open "C:\Bethesda Car Rental\" & txtReceiptNumber.Text & _  
        ".bcr" For Output As #1  
  
    Write #1, txtEmployeeNumber.Text  
    Rem Some people would not include the Employee Name in  
    Rem the file because it is already stored in the workbook.  
    Rem But we will include it in our file  
    Write #1, txtEmployeeName.Text  
    Write #1, txtDrvLicenseNbr.Text  
    Rem Some people would not include the customer name, address,  
    Rem city, state, and ZIP code in the file because they are  
    Rem already part of a workbook.  
    Rem But we will include them in our file  
    Write #1, txtCustomerName.Text  
    Write #1, txtAddress.Text  
    Write #1, txtCity.Text  
    Write #1, txtState.Text  
    Write #1, txtZIPCode.Text  
    Write #1, txtStartDate.Text  
    Write #1, txtEndDate.Text  
    Write #1, txtTagNumber.Text  
    Write #1, cbxCarConditions.Text  
    Rem Some people would not include the car make, model,  
    Rem and year in the file because they are  
    Rem already stored in a workbook.  
    Rem But we will include them here  
    Write #1, txtMake.Text  
    Write #1, txtModel.Text  
    Write #1, txtCarYear.Text  
    Write #1, cbxTankLevels.Text  
    Write #1, txtMileageStart.Text  
    Write #1, txtMileageEnd.Text  
    Write #1, txtRateApplied.Text  
    Write #1, txtTaxRate.Text  
    Write #1, txtDays.Text  
    Write #1, txtTaxAmount.Text  
    Write #1, txtSubTotal.Text  
    Write #1, txtOrderTotal.Text  
    Write #1, "Car Rented"  
    Write #1, txtNotes.Text  
  
    Close #1
```

```
cmdSave_Error:
    MsgBox "There is a problem with the form. It cannot be saved."
    Resume Next
End Sub
```

4. On the Standard toolbar, click the Save button
5. Return to Microsoft Excel and click the Switchboard tab sheet if necessary
6. In the Developer tab of the Ribbon, in the Controls section, click Insert
7. In the ActiveX Controls section, click Command Button
8. Click the worksheet
9. Right-click the new button and click Properties
10. In the properties window, change the following characteristics
(Name): **cmdCreateRentalOrder**
Caption: **Create New Rental Order**
11. Right-click the button and click View Code
12. Implement the event as follows:


```
Private Sub cmdCreateRentalOrder_Click()
    frmNewRentalOrder.Show
End Sub
```
13. Press Ctrl + S to save
14. Return to Microsoft Excel
15. In the Controls section of the Ribbon, click the Design Mode button to uncheck it
16. Click the button to display the form
17. Enter some values for a rental order

Bethesda Car Rental - Order Processing - New Rental Order			
Processed By		Car Selected	
Employee #:	25-947	Monay, Gertrude	
Processed For		Tag Number:	297419
Driver's Lic. #:	J-938-928-274		Condition:
Name:	Chris Young		Make:
Address:	8522 Aulage Street		Model:
City:	Rockville		Year:
State:	MD	ZIP Code:	20852
		Mileage Start:	8442
		Order Evaluation	
Start Date:	8/26/2009	End Date:	8/26/2009
Notes:			
Rate Applied:	24.95	Tax Rate:	
Days:	0	Tax Amount:	
Sub-Total:	0.00	Order Total:	
Receipt #:	655337	Save	Reset / New Rental Order

18. Write down the receipt number on a piece of paper
19. Click the Save button
20. Click the Reset button
21. Enter some values for another rental order

Bethesda Car Rental - Order Processing - New Rental Order			
Processed By		Car Selected	
Employee #:	62-845	Katts, Patricia	Tag Number: M931429
Processed For		Condition:	E
Driver's Lic. #:	M-505-862-575	Make:	Ford
Name:	Lynda Melman	Model:	E150XL
Address:	4277 Jamison Avenue	Year:	2010
City:	Silver Spring	Tank Level:	1
State:	MD	Mileage Start:	26905
ZIP Code:	20904	Mileage End:	
		Order Evaluation	
Start Date:	8/26/2009	End Date:	8/26/2009
Rate Applied:	24.95	Tax Rate:	
Days:	0	Tax Amount:	
Sub-Total:	0.00	Order Total:	
Notes:			
Receipt #:	873981	Save	Reset / New Rental Order

22. Click the Save button

23. Close the form and return to Microsoft Visual Basic

Opening a File

Opening a File

Instead of creating a new file, you may want to open an existing file. To support this operation, the VBA provides a procedure named **Open**. Its syntax is:

```
Open pathname For Input [Access access] [lock] As [#]filenumber [Len=reclength]
```

The **Open** procedure takes many arguments, some are required and others are not. The **Open** word, **For Input** expression, and the **As #** expression are required.

The first argument, *pathname*, is required. This is a string that can be the name of the file. The file can have an extension or not. Here is an example:

```
Open "example.dat"
```

If you specify only the name of the file, the interpreter would look for the file in the same folder where the current workbook is. If you want, you can provide a complete path for the file. This would include the drive, the (optional) folder(s), up to the name of the file, with or without extension.

Besides the name of the file or its path, the *mode* factor is required. To open a file, the mode factor can be:

- **Binary**: The file will be opened and its value(s) would be read as (a) binary value(s)
- **Append**: The file will be opened and new values can be added to the end of the existing values
- **Input**: The file will be opened normally
- **Random**: The will be opened for random access

Here is an example of opening a file:

```
Private Sub cmdSave_Click()  
    Open "example.dat" For Input As #1
```

The *access* factor is optional. This factor can have one of the following values:

- **Read:** After the file has been opened, values will be read from it
- **Read Write:** Whether the file was created or opened, values can be read from it and/or written to it

If you decide to specify the *access* factor, precede its value with the **Access** keyword.

The *lock* factor is optional and its possible values can be:

- **Shared:** Other applications can access this file while the current application is accessing it
- **Lock Read:** Other applications are not allowed to access this file while the current application is reading from it
- **Lock Read Write:** Other applications are not allowed to access this file while the current application is using it

On the right side of #, type a number, for the *filenumber* factor, between 1 and 511. Use the same rules/description we saw for creating a file.

The *reclength* factor is optional. If the file was opened, this factor specifies the length of the record that was read.

❖ Practical Learning: Introducing File Opening

1. Click the body of the form.
From the properties window, write down the values of the Height and the Width properties
2. Click the body of the form
3. Press Ctrl + A to select all controls on the form
4. To add a new form, on the main menu, click Insert -> UserForm
5. In the Properties window, change the following characteristics:
(Name): **frmRentalOrderReview**
Caption: **Car Rental - Order Processing - Rental Order Review**
6. Enlarge the form using the height and width of the first form
7. Complete the design of the form as follows:

Control	(Name)	Caption/Text	Other Properties
Label		Receipt #:	
Text Box	txtReceiptNumber		
Command Button	cmdOpen	Open	
Label		Order Status:	
Text Box	cbxOrderStatus		

Label		Processed By	BackColor: &H00808080& BorderColor: &H00000000& ForeColor: &H00FFFFFF&
Label		Car Selected	BackColor: &H00808080& BorderColor: &H00000000& ForeColor: &H00FFFFFF&
Label		Employee #:	
Text Box	txtEmployeeNumber		
Text Box	txtEmployeeName		
Label		Tag Number:	
Text Box	txtTagNumber		
Label		Condition:	
Combo Box	cbxCarConditions		
Label		Processed For	BackColor: &H00808080& BorderColor: &H00000000& ForeColor: &H00FFFFFF&
Label		Make:	
Text Box	txtMake		
Label		Driver's Lic. #:	
Text Box	txtDrvLicenseNbr		
Label		Model:	
Text Box	txtModel		
Label		Name:	
Text Box	txtCustomerName		
Label		Year:	
Text Box	txtCarYear		TextAlign: 3 - fmTextAlignRight
Label		Tank Level:	
Combo Box	cbxTankLevels		
Label		Address:	
Text Box	txtAddress		
Label		Mileage Start:	
Text Box	txtMileageStart		TextAlign: 3 - fmTextAlignRight
Label		Mileage End:	
Text Box	txtMileageEnd		TextAlign: 3 - fmTextAlignRight
Label		City:	
Text Box	txtCity		
Label		Order Evaluation	BackColor: &H00808080& BorderColor: &H00000000& ForeColor: &H00FFFFFF&
Label		State:	
Text Box	txtState		
Label		ZIP Code:	
Text Box	txtZIPCode		
Label		Rate Applied:	
Text Box	txtRateApplied	24.95	TextAlign: 3 - fmTextAlignRight
Label		Tax Rate:	
Text Box	txtTaxRate	5.75	TextAlign: 3 - fmTextAlignRight
Label			BackColor: &H00808080& BorderColor: &H00000000& ForeColor: &H00FFFFFF&
Label		Days:	
Text Box	txtDays	0	TextAlign: 3 - fmTextAlignRight
Label		Tax Amount:	
Text Box	txtTaxAmount	0.00	TextAlign: 3 - fmTextAlignRight
Label		Start Date:	
Text Box	txtStartDate		
Label		End Date:	
Text Box	txtEndDate		
Label		Sub-Total:	
Text Box	txtSubTotal	0.00	TextAlign: 3 - fmTextAlignRight
Label		Order Total:	
Text Box	txtOrderTotal	0.00	TextAlign: 3 - fmTextAlignRight
Command Button	cmdUpdateFile	Update and Save the File	

8. Right-click the form and click View Code
9. In the Procedure combo box, select Activate
10. Implement the event as follows:


```

Private Sub UserForm_Activate()
    Dim strRandomNumber As String

    cbxOrderStatus.AddItem "Car Rented"
    cbxOrderStatus.AddItem "Order Finalized"
    cbxOrderStatus.AddItem "Order Reserved"

    cbxCarConditions.AddItem "Needs Repair"
    cbxCarConditions.AddItem "Drivable"
    cbxCarConditions.AddItem "Excellent"

    cbxTankLevels.AddItem "Empty"
    cbxTankLevels.AddItem "1/4 Empty"
    cbxTankLevels.AddItem "1/2 Full"
    cbxTankLevels.AddItem "3/4 Full"
    cbxTankLevels.AddItem "Full"
End Sub

```

11. In the Object combo box, select txtRateApplied
12. In the Procedure combo box, select Exit
13. Implement the event as follows:

```

Private Sub CalculateRentalOrder()
    Dim RateApplied As Double
    Dim Days As Integer
    Dim SubTotal As Double
    Dim TaxRate As Double
    Dim TaxAmount As Double
    Dim OrderTotal As Double

    ' Check the value in the Rate Applied text box
    ' If there is no valid value, set the Rate Applied to 0
    If txtRateApplied.Text = "" Then
        RateApplied = 0
    ElseIf Not IsNumeric(txtRateApplied.Text) Then
        RateApplied = 0
    Else
        ' Otherwise, get the rate applied
        RateApplied = CDBl(txtRateApplied.Text)
    End If

    ' We will let the employee enter the number of days the car was rented
    ' Check whether the employee entered a valid number
    ' If the number is not good, set the number of days to 0
    If txtDays.Text = "" Then
        Days = 0
    ElseIf Not IsNumeric(txtDays.Text) Then
        Days = 0
    Else
        ' Otherwise, get the number of days
        Days = CInt(txtDays.Text)
    End If

    If txtTaxRate.Text = "" Then
        TaxRate = 0
    ElseIf Not IsNumeric(txtTaxRate.Text) Then
        TaxRate = 0
    Else
        TaxRate = CDBl(txtTaxRate.Text)
    End If

    ' Calculate the things
    SubTotal = RateApplied * Days
    TaxAmount = SubTotal * TaxRate / 100
    OrderTotal = SubTotal + TaxAmount

    txtSubTotal.Text = FormatNumber(SubTotal)
    txtTaxAmount.Text = FormatNumber(TaxAmount)
    txtOrderTotal.Text = FormatNumber(OrderTotal)
End Sub

Private Sub txtRateApplied_Exit(ByVal Cancel As MSForms.ReturnBoolean)
On Error GoTo txtRateApplied_Error
    Call CalculateRentalOrder

    Exit Sub
txtRateApplied_Error:
    MsgBox "There is something wrong with the " & _
        "value you entered for the rate applied"
End Sub

```

14. In the Object combo box, select txtDays

15. In the Procedure combo box, select Exit

16. Implement the event as follows:

```
Private Sub txtDays_Exit(ByVal Cancel As MSForms.ReturnBoolean)
On Error GoTo txtDays_Error
    Call CalculateRentalOrder

Exit Sub
txtDays_Error:
    MsgBox "There is something wrong with the number " & _
        "of days you entered."
End Sub
```

17. In the Object combo box, select txtTaxRate

18. In the Procedure combo box, select Exit

19. Implement the event as follows:

```
Private Sub txtTaxRate_Exit(ByVal Cancel As MSForms.ReturnBoolean)
On Error GoTo txtTaxRate_Error
    Call CalculateRentalOrder

Exit Sub
txtTaxRate_Error:
    MsgBox "There is something wrong with the tax rate you specified."
End Sub
```

20. On the Standard toolbar, click the Save button

Reading From a File

After opening a file, you can read values from it. Before reading the value(s), you should declare one or more variables that would receive the values to be read. Remember that the idea of using a variable is to reserve a memory space where you can store a value. In the same way, when reading a value from a file, you would get the value from the file and then store that value in the computer memory. A variable would make it easy for you to refer to that value when necessary.

To support the ability to open a file, the VBA provides two procedures. If you wrote the values using the **Print** statement, to read the values, use the **Input** or the **Line Input** statement (using **Input** or **Line Input** is only a suggestion, not a rule). The syntax of the **Input** procedure is:

```
Input #filenumber, varlist
```

The **Input** statement takes two required factors but the second can be made of various parts.

The *filenumber* factor is the *filenumber* you would have used to open the file. The *filenumber* is followed by a comma.

The *varlist* factor can be made of 1 or more parts. To read only one value, after the comma of the *filenumber* factor, type the name of the variable that will receive the value. Here is an example:

```
Private Sub cmdOpen_Click()
    Dim FirstName As String

    Open "Employee.txt" For Input As #1

    Input #1, FirstName

    Close #1
End Sub
```

In the same way, you can read each value on its own line. One of the better uses of the **Input** statement is the ability to read many values using a single statement. To do this, type the variables on the same **Input** line but separate them with commas. Here is an example:

```
Private Sub cmdOpen_Click()
    Dim FirstName As String
    Dim LastName As String
    Dim IsFullTimeEmployee As Boolean

    Open "Employee.txt" For Input As #1

    Input #1, FirstName, LastName, IsFullTimeEmployee

    Close #1
End Sub
```

If you have a file that contains many lines, to read one line at a time, you can use the **Line Input** statement. Its syntax is:

```
Line Input #filenumber, varname
```

This statement takes two factors and both are required. The *filenumber* is the number you would have used to open the file. When the **Line Input** statement is called, it reads a line of text until it gets to the end of the file. One of the limitations of the **Line Input** statement is that it has a hard time reading anything other than text because it may not be able to determine where the line ends.

When reviewing the ability to write values to a file, we saw that the **Print** statement writes a Boolean value as **True** or **False**. If you use the **Input** statement to read such a value, the interpreter may not be able to read the value. We saw that an alternative to the **Print** statement was **Write**. We saw that, among the differences between **Print** and **Write**, the latter writes Boolean values using the # symbol. This makes it possible for the interpreter to easily read such a value. For these reasons, in most cases, it may be a better idea to prefer using the **Write** statement when writing values other than strings to a file.

❖ Practical Learning: Reading From a File

1. On the form, double-click the Open button
2. Implement the event as follows:

```
Private Sub cmdOpen_Click()
    On Error GoTo cmdOpen_Error

    Dim EmployeeNumber As String
    Dim EmployeeName As String, DrvLicenseNbr As String
    Dim CustomerName As String, Address As String
    Dim City As String, State As String
    Dim ZIPCode As String, StartDate As String
    Dim EndDate As String, TagNumber As String
    Dim CarConditions As String, Make As String
    Dim Model As String, CarYear As String
    Dim TankLevels As String, MileageStart As String
    Dim MileageEnd As String, RateApplied As String
    Dim TaxRate As String, Days As String
    Dim TaxAmount As String, SubTotal As String
    Dim OrderTotal As String, OrderStatus As String
    Dim Notes As String

    Rem We are not doing any validation here because there are
    ' issues we haven't explored yet. For example, we haven't yet
    ' learned how to check the list of files in a directory.
    ' We also haven't yet learned how to check whether a file
    ' exists in a directory.
    Open "C:\Bethesda Car Rental\" & _
        txtReceiptNumber.Text & ".bcr" For Input As #1

    Input #1, EmployeeNumber
    Input #1, EmployeeName
    Input #1, DrvLicenseNbr
    Input #1, CustomerName
    Input #1, Address
    Input #1, City
    Input #1, State
    Input #1, ZIPCode
    Input #1, StartDate
    Input #1, EndDate
    Input #1, TagNumber
    Input #1, CarConditions
    Input #1, Make
    Input #1, Model
    Input #1, CarYear
    Input #1, TankLevels
    Input #1, MileageStart
    Input #1, MileageEnd
    Input #1, RateApplied
    Input #1, TaxRate
    Input #1, Days
    Input #1, TaxAmount
    Input #1, SubTotal
    Input #1, OrderTotal
    Input #1, OrderStatus
    Input #1, Notes

    txtEmployeeNumber.Text = EmployeeNumber
    txtEmployeeName.Text = EmployeeName
    txtDrvLicenseNbr.Text = DrvLicenseNbr
    txtCustomerName.Text = CustomerName
    txtAddress.Text = Address
    txtCity.Text = City
    txtState.Text = State
    txtZIPCode.Text = ZIPCode
    txtStartDate.Text = StartDate
    txtEndDate.Text = EndDate
    txtTagNumber.Text = TagNumber
    cbxCarConditions.Text = CarConditions
```

```

txtMake.Text = Make
txtModel.Text = Model
txtCarYear.Text = CarYear
cbxTankLevels.Text = TankLevels
txtMileageStart.Text = MileageStart
txtMileageEnd.Text = MileageEnd
txtRateApplied.Text = RateApplied
txtTaxRate.Text = TaxRate
txtDays.Text = Days
txtTaxAmount.Text = TaxAmount
txtSubTotal.Text = SubTotal
txtOrderTotal.Text = OrderTotal
cbxOrderStatus.Text = OrderStatus
txtNotes.Text = Notes

```

```
Close #1
```

```
Exit Sub
```

```
cmdOpen_Error:
```

```
MsgBox "There was a problem when trying to open the file."
```

```
Resume Next
```

```
End Sub
```

3. In the Object combo box, select cmdUpdateRentalOrder
4. Implement the Click event as follows:

```
Private Sub cmdUpdateRentalOrder_Click()
On Error GoTo cmdSave_Error
```

```
Open "C:\Bethesda Car Rental\" & txtReceiptNumber.Text & _
 ".bcr" For Output As #1
```

```
Write #1, txtEmployeeNumber.Text
Write #1, txtEmployeeName.Text
Write #1, txtDrvLicenseNbr.Text
Write #1, txtCustomerName.Text
Write #1, txtAddress.Text
Write #1, txtCity.Text
Write #1, txtState.Text
Write #1, txtZIPCode.Text
Write #1, txtStartDate.Text
Write #1, txtEndDate.Text
Write #1, txtTagNumber.Text
Write #1, cbxCarConditions.Text
Write #1, txtMake.Text
Write #1, txtModel.Text
Write #1, txtCarYear.Text
Write #1, cbxTankLevels.Text
Write #1, txtMileageStart.Text
Write #1, txtMileageEnd.Text
Write #1, txtRateApplied.Text
Write #1, txtTaxRate.Text
Write #1, txtDays.Text
Write #1, txtTaxAmount.Text
Write #1, txtSubTotal.Text
Write #1, txtOrderTotal.Text
Write #1, cbxCarConditions.Text
Write #1, txtNotes.Text
```

```
Close #1
```

```
Exit Sub
```

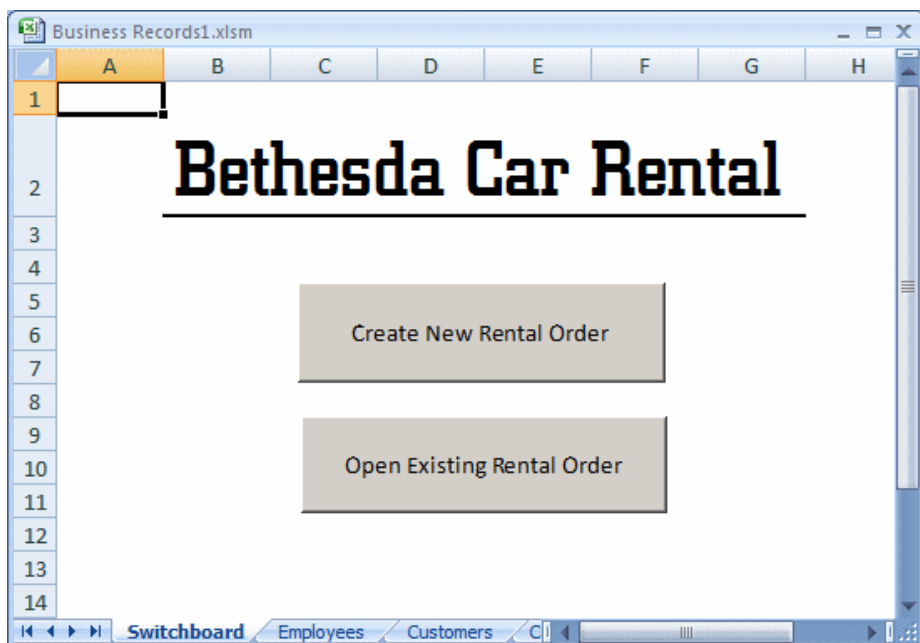
```
cmdSave_Error:
```

```
MsgBox "There is a problem with the form. " & _
 "The rental order cannot be updated."
```

```
Resume Next
```

```
End Sub
```

5. On the Standard toolbar, click the Save button
6. Return to Microsoft Excel
7. Click the Switchboard tab sheet
8. In the Developer tab of the Ribbon, in the Controls section, click Insert
9. In the ActiveX Controls section, click Command Button
10. Click the worksheet
11. Right-click the new button and click Properties
12. In the properties window, change the following characteristics
(Name): **cmdOpenRentalOrder**
Caption: **Open Existing Rental Order**



13. Right-click the open Existing Rental Order button and click View Code
14. Implement the event as follows:

```
Private Sub cmdOpenRentalOrder_Click()
    frmRentalOrderReview.Show
End Sub
```

15. Press Ctrl + S to save
16. Close Microsoft Visual Basic
17. In the Controls section of the Ribbon, click the Design Mode button to uncheck it
18. Click the button to display the form
19. Click the Receipt # text box
20. Type a receipt number of one of the rental orders you created earlier
21. Click the Open button
22. Select a different option in the order status combo box
23. Change the value of mileage end, the end date and the days

Car Rental - Order Processing - Rental Order Review			
Receipt #:	<input type="text" value="655337"/>	<input type="button" value="Open"/>	Order Status: <input type="text" value="Order Finalized"/>
Processed By		Car Selected	
Employee #:	<input type="text" value="25-947"/> <input type="text" value="Monay, Gertrude"/>	Tag Number:	<input type="text" value="297419"/> Condition: <input type="text" value="E"/>
Processed For		Make:	<input type="text" value="BMW"/>
Driver's Lic. #:	<input type="text" value="J-938-928-274"/>	Model:	<input type="text" value="335i"/>
Name:	<input type="text" value="Chris Young"/>	Year:	<input type="text" value="2010"/> Tank Level: <input type="text" value="3"/>
Address:	<input type="text" value="8522 Aulage Street"/>	Mileage Start:	<input type="text" value="8442"/> Mileage End: <input type="text"/>
City:	<input type="text" value="Rockville"/>	Order Evaluation	
State:	<input type="text" value="MD"/> ZIP Code: <input type="text" value="20852"/>	Rate Applied:	<input type="text" value="49.95"/> Tax Rate: <input type="text"/>
		Days:	<input type="text" value="2"/> Tax Amount: <input type="text"/>
Start Date:	<input type="text" value="8/26/2009"/> End Date: <input type="text" value="8/28/2009"/>	Sub-Total:	<input type="text" value="99.90"/> Order Total: <input type="text"/>
Notes:	<input type="text"/>		
<input type="button" value="Update and Save"/>			

- 24. Click the Update and Save rental Order button
- 25. Select the number in the Receipt # text box
- 26. Type another receipt number you saved previously
- 27. Click the Open button
- 28. Select different values on the rental order:

- 29. Click the Update and Save rental Order button
- 30. Close the form and return to Microsoft Visual Basic

Other Techniques of Opening a File

Besides calling the **Show()** method of the **FileDialog** class, the Application class provides its own means of opening a file. To support it, the Application class provides the **FindFile()** method. Its syntax is:

```
Public Function Application.FindFile() As Boolean
```

If you call this method, the Open File dialog with its default settings would come up. The user can then select a file and click open. If the file is a workbook, it would be opened and its content displayed in Microsoft Excel. If the file is text-based, or XML, etc, Microsoft Excel would proceed to open or convert it.

Project



Dates and Times in VBA Excel

Fundamentals of Dates

Introduction

The Visual Basic language has a strong support for date values. It is equipped with a data type named **Date**. To create and manipulate dates, you have various options. To declare a date variable, you use the **Date** data type. To support date and time-based operations, the Visual Basic language provides various functions. Besides the Visual Basic language, the Microsoft Excel library provides its own support for dates and times.

If you already know the components of the date value you want to use, you can include them between two # signs but following the rules of a date format from the Regional Settings of Control Panel. Here is an example:

```
Public Sub Exercise
    Dim DateHired As Date

    DateHired = # 02/08/2003 #

    MsgBox("Date Hired: " & DateHired)
End Sub
```

This would produce:



The Current Date

To get the current date of the computer, you can call the Visual Basic's Date function. Here is an example:

```
Sub Exercise()
    MsgBox Date
End Sub
```

In Microsoft Excel, to get the current date, you can call the TODAY() function. Here is an example:

```
Sub Exercise()
    Range("B2").FormulaR1C1 = "=TODAY()"
End Sub
```

The Parts of a Date Value

When you compose a date value, you must follow some rules. The rules depend on the language you are using. We will review those of the US English.

In a year, a month is recognized by an index in a range from 1 to 12. A month also has a name. The name of a month is given in two formats: complete or short. These are:

Month Index	Full Name	Short Name
1	January	Jan
2	February	Feb
3	March	Mar
4	April	Apr
5	May	May
6	June	Jun
7	July	Jul
8	August	Aug

9	September	Sep
10	October	Oct
11	November	Nov
12	December	Dec

A week is a combination of 7 consecutive days of a month. Each day can be recognized by an index from 1 to 7 (1, 2, 3, 4, 5, 6, 7). The day of each index is recognized by a name. In US English, the first day with an index of 1 is named Sunday while the last day with an index of 7 is named Monday. Like the months of a year, the days of a week have long and short names. These are:

US English Day Index	Full Name	Short Name
1	Sunday	Sun
2	Monday	Mon
3	Tuesday	Tue
4	Wednesday	Wed
5	Thursday	Thu
6	Friday	Fri
7	Saturday	Sat

These are the default in US English. In most calculations, the Visual Basic language allows you to specify what day should be the first in a week.

The year is expressed as a numeric value.

Dates Formats

In US English, to express a date value, you can use one of the following formats:

- mm-dd-yy
- mm-dd-yyyy

You must start the date with a number that represents the month (a number from 1 to 12). After the month value, enter -. Then type the day value as a number between 1 and 28, 29, 30, or 31 depending on the month and the (leap) year. Follow it with -. End the value with a year in 2 or 4 digits. Here are examples 06-12-08 or 10-08-2006.

You can also use one of the following formats:

- dd-mmm-yy
- dd mmm yy
- dd-mmmm-yy
- dd mmmm yy
- dd-mmm-yyyy
- dd mmm yyyy
- dd-mmmm-yyyy
- dd mmmm yyyy

This time, enter the day value followed either by an empty space or -. Follow with the short name of the month in the mmm placeholder or the complete name of the month for the mmmm placeholder, followed by either an empty space or -. End the value with the year, using 2 or 4 digits.

As you may know already, in US English, you can start a date with the month. In this case, you can use one of the following formats:

- mmm dd, yy
- mmm dd, yyyy
- mmmm dd, yy
- mmmm dd, yyyy

As seen with the previous formats, mmm represents the short name of a month and mmmm represents the full name of a month. As mentioned already, the dd day can be expressed with 1 or 2 digits and the single digit can have a leading 0. After the day value, (you must) enter a comma followed by the year either with 2 or 4 digits.

A Date Value

We have seen that, when creating a date, you can include its value between # signs. An alternative is to provide a date as a string. To support this, the Visual Basic language provides a function named **DateValue**. Its syntax is:

When calling this function, provide a valid date as argument. The validity depends on the language of the operating system. If working in US English, you can use one of the formats we saw above. Here is an example:

```
Public Sub Exercise
    Dim DateHired As Date

    DateHired = DateValue("22-Aug-2006")

    MsgBox("Date Hired: " & DateHired)
End Sub
```

This would produce:



A Date as Serial

An alternative to initializing a date variable is to use a function named **DateSerial**. Its syntax is:

```
Function DateSerial(ByVal [Year] As Integer, _
    ByVal [Month] As Integer, _
    ByVal [Day] As Integer) As Variant
```

As you can see, this function allows you to specify the year, the month, and the day of a date value, of course without the # signs. When it has been called, this function returns a **Variant** value, which can be converted into a **Date**. Here is an example:

```
Public Sub Exercise
    Dim DateHired As Date

    DateHired = DateSerial(2003, 02, 08)
    MsgBox("Date Hired: " & DateHired)
End Sub
```

This would produce:



When passing the values to this function, you must restrict each component to the allowable range of values. You can pass the year with two digits from 0 to 99. Here is an example:

```
Public Sub Exercise
    Dim DateHired As Date

    DateHired = DateSerial(03, 2, 8)
    MsgBox("Date Hired: " & DateHired)
End Sub
```

If you pass the year as a value between 0 and 99, the interpreter would refer to the clock on the computer to get the century. At the time of this writing (in 2009), the century would be 20 and the specified year would be added, which would produce 2003. To be more precise and reduce any confusion, you should always pass the year with 4 digits.

The month should (must) be a value between 1 and 12. If you pass a value higher than 12, the interpreter would calculate the remainder of that number by 12 (that number MOD 12 = ?). The result of the integer division would be used as the number of years and added to the first argument. The remainder would be used as the month of the date value. For example, if you pass the month as 18, the integer division would produce 1, so 1 year would be added to the first argument. The remainder is 6 (18 MOD 12 = 6); so the month would be used as 6 (June). Here is an example:

```
Public Sub Exercise
    Dim DateHired As Date

    DateHired = DateSerial(2003, 18, 8)
    MsgBox("Date Hired: " & DateHired)
End Sub
```

This would produce:



As another example, if you pass the month as 226, the integer division ($226 \setminus 12$) produces 18 and that number would be added to the first argument ($2003 + 18 = 2021$). The remainder of 226 to 12 ($226 \text{ MOD } 12 = 10$) is 10 and that would be used as the month. Here is an example:

```
Public Sub Exercise
    Dim DateHired As Date

    DateHired = DateSerial(2003, 226, 8)
    MsgBox("Date Hired: " & DateHired)
End Sub
```

This would produce:



If the month is passed as 0, it is considered 12 (December) of the previous year. If the month is passed as -1, it is considered 11 (November) of the previous year and so on. If the month is passed as a number lower than -11, the interpreter would calculate its integer division to 12, add 1 to that result, use that number as the year, calculate the remainder to 12, and use that result as the month.

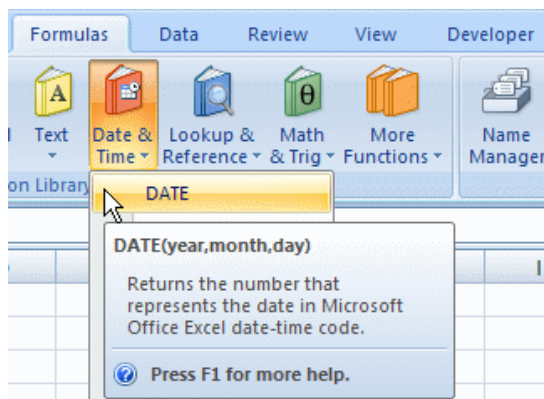
Depending on the month, the value of the day argument can be passed as a number between 1 and 28, between 1 and 29, between 1 and 30, or between 1 and 31. If the day argument is passed as a number lower than 1 or higher than 31, the interpreter uses the first day of the month passed as the second argument. This is 1.

If the day is passed as -1, the day is considered the last day of the previous month of the *Month* argument. For example, if the *Month* argument is passed as 4 (April) and the *Day* argument is passed as -1, the interpreter would use 31 as the day because the last day of March is 31.

If the *Month* argument is passed as 3 (March) and the *Day* argument is passed as -1, the interpreter would refer to the *Year* argument to determine whether the year is leap or not. This would allow the interpreter to use either 28 or 29 for the day value. The interpreter uses this algorithm for any day value passed as the third argument when the number is lower than 1.

If the *Day* argument is passed with a value higher than 28, 29, 30, or 31, the interpreter uses this same algorithm in reverse order to determine the month and the day.

Besides the Visual Basic's **DateSerial()** function, the Microsoft Excel library provides a function named DATE



When using this function, pass the values of the year, the month, and the day. You can use exactly the rules we reviewed for the **DateSerial()** function. Here is an example:

```
Sub Exercise()
    Range("B2").FormulaR1C1 = "=DATE(2003, 226, 8)"
End Sub
```

This would produce:

	A	B	C	D	E	F
1						
2		10/8/2021				
3						

Converting a Value to Date

If you have a value such as one provided as a string and you want to convert it to a date, you can call the **CDate()** function. Its syntax is:

```
Function CDate(Value As Object) As Date
```

This function can take any type of value but the value must be convertible to a valid date. If the function succeeds in the conversion, it produces a Date value. If the conversion fails, it produces an error.

The Components of a Date

Introduction

As seen so far, a date is a value made of at least three parts: the year, the month, and the day. The order of these components and how they are put together to constitute a recognizable date depend on the language and they are defined in the Language and Regional Settings in Control Panel.

The Year of a Date

The Visual Basic language supports the year of a date ranging from 1 to 9999. This means that this is the range you can consider when dealing with dates in your worksheets. In most operations, when creating a date, if you specify a value between 1 and 99, the interpreter would use the current century for the left two digits. This means that, at the time of this writing (2009), a year such as 4 or 04 would result in the year 2004. In most cases, to be more precise, you should usually or always specify the year with 4 digits.

If you have a date value whose year you want to find out, you can call the Visual Basic's **Year()** function. Its syntax is:

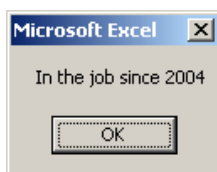
```
Public Function Year(ByVal DateValue As Variant) As Integer
```

As you can see, this function takes a date value as argument. The argument should hold a valid date. If it does, the function returns the numerical year of a date. Here is an example:

```
Public Sub Exercise
    Dim DateHired As Date

    DateHired = #2/8/2004#
    MsgBox ("In the job since " & Year(DateHired))
End Sub
```

This would produce:



Besides the Visual Language's Year() function, the Microsoft Excel library provides a function named YEAR that can be used to get the year value of a date. The date must be provided in the format the **DATE()** function.

The Month of a Year

The month part of a date is a numeric value that goes from 1 to 12. When creating a date, you can specify it with 1 or 2 digits. If the month is between 1 and 9 included, you can precede it with a leading 0.

If you have a date value and want to get its month, you can call the **Month()** function. Its syntax is:

```
Function Month(ByVal DateValue As Variant) As Integer
```

This function takes a **Date** object as argument. If the date is valid, the function returns a number between 1 and 12 for the month. Here is an example:

```
Public Sub Exercise
    Dim DateHired As Date
```

```
DateHired = #2/8/2004#
MsgBox ("Month hired " & Month(DateHired))
End Sub
```

This would produce:



Besides the Visual Basic's **Month()** function, you can use the **MONTH()** function of the Microsoft Excel library. This function takes one argument as the type of date produced by a call to the **DATE()** function.

As mentioned already, the **Month()** function produces a numeric value that represents the month of a date. Instead of getting the numeric index of the month of a date, if you want to get the name of the month, you can call the Visual Basic function named **MonthName()**. Its syntax is:

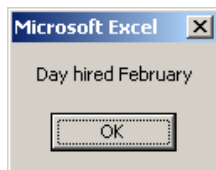
```
Function MonthName(ByVal Month As Integer, _
    Optional ByVal Abbreviate As Boolean = False) As String
```

This function takes one required and one optional arguments. The required argument must represent the value of a month. If it is valid, this function returns the corresponding name. Here is an example:

```
Public Sub Exercise
    Dim DateHired As Date

    DateHired = #2/8/2004#
    MsgBox("Day hired " & MonthName(Month(DateHired)))
End Sub
```

This would produce:



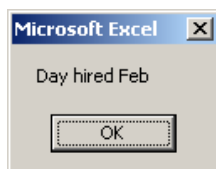
The second argument allows you to specify whether you want to get the complete or the short name. The default is the complete name, in which case the default value of the argument is **False**. If you want to get the short name, pass the second argument as **True**. Here is an example:

```
Public Sub Exercise
    Dim DateHired As Date

    DateHired = #2/8/2004#

    MsgBox("Month hired " & MonthName(Month(DateHired), True))
End Sub
```

This would produce:



The Day of a Month

The day is a numeric value in a month. Depending on the month (and the year), its value can range from 1 to 29 (February in a leap year), from 1 to 28 (February in a non-leap year), from 1 to 31 (January, March, May, July, August, October, and December), or from 1 to 30 (April, June, September, and November).

If you have a date value and you want to know its day in a year, you can call the **Day()** function. Its syntax is:

```
Function Day(ByVal DateValue As Variant) As Integer
```

This function takes a date as argument. If the date is valid, the function returns the numeric day in the month of the date argument. Here is an example:

```
Public Sub Exercise
    Dim DateHired As Date
```

```
DateHired = #2/8/2004#
MsgBox("Day hired " & Day(DateHired))
End Sub
```

This would produce:



The Day of a Week

To get the name of the day of a week, you can use a function named **WeekdayName**. Its syntax is:

```
Function WeekdayName( _
    ByVal Weekday As Integer, _
    Optional ByVal Abbreviate As Boolean = False, _
    Optional ByVal FirstDayOfWeekValue As Integer = 0 _
) As String
```

This function takes one required and two optional arguments. The required argument must be, or represent, a value between 0 and 7. If you pass it as 0, the interpreter will refer to the operating system's language to determine the first day of the week, which in US English is Sunday. Otherwise, if you pass one of the indexes we saw **above**, the function would return the corresponding name of the day. Here is an example:

```
Public Sub Exercise
    MsgBox("Day hired: " & WeekdayName(4))
End Sub
```

This would produce:



If you pass a negative value or a value higher than 7, you would receive an error.

The second argument allows you to specify whether you want to get the complete or the short name. The default value of this argument is **False**, which produces a complete name. If you want a short name, pass the second argument as **True**. Here is an example:

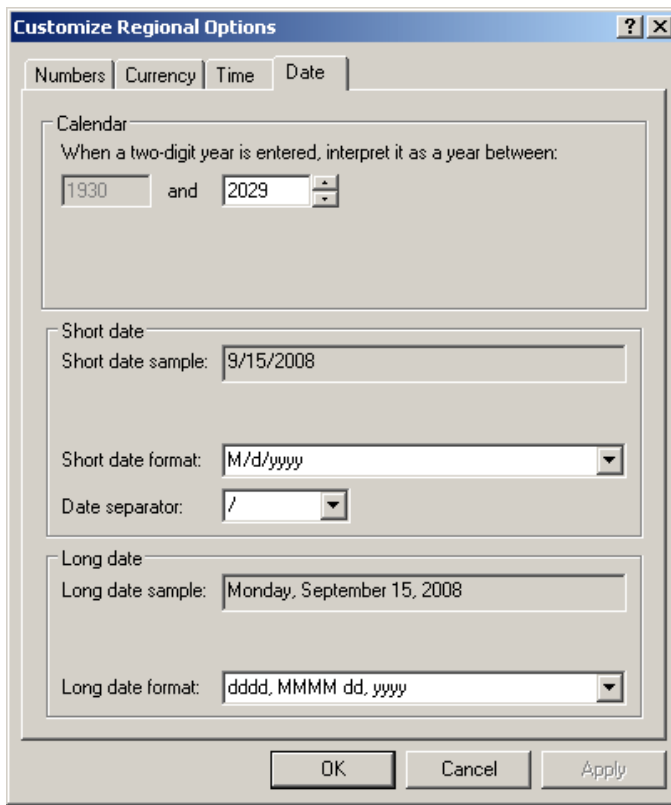
```
Public Sub Exercise
    MsgBox("Day hired: " & WeekdayName(4, True))
End Sub
```

As mentioned already, the Visual Basic language allows you to specify what days should be the first day of the week. This is the role of the third argument.

Formatting a Date Value

Introduction

Formatting a date consists of specifying how the value would be displayed to the user. The Visual Basic language provides various options. The US English language supports two primary date formats known as long date and short date. You can check them in the Date property page of the Customize Regional Options accessible from the Regional Settings in Control Panel:



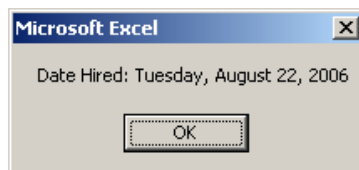
To support these primary formats, the Visual Basic language provides a function named **FormatDateTime**. Its syntax is:

```
Function FormatDateTime(  
    ByVal Expression As Variant,  
    Optional ByVal NamedFormat As Integer = 0  
) As String
```

The first argument of this function must be a valid **Date** value. The second argument is an integer. For a date, this argument can be 1 or 2. Here is an example:

```
Public Sub Exercise  
    Dim DateHired$  
  
    DateHired$ = FormatDateTime("22-Aug-2006", 1)  
    MsgBox("Date Hired: " & DateHired)  
End Sub
```

This would produce:



Using the Format Function

To support more options, the Visual Basic language provides the **Format()** function that we saw in the previous lesson. We saw that its syntax was:

```
Function Format( _  
    ByVal Expression As Object, _  
    Optional ByVal Style As String = "" _  
) As String
```

Remember that the first argument is the date that needs to be formatted. The second argument is a string that contains the formatting to apply. To create it, you use a combination of the month, day, and/or year characters we saw as **date formats**. Here is an example:

```
Public Sub Exercise  
    Dim DateHired As Date  
  
    DateHired = #12/28/2006#  
    MsgBox("Date Hired: " & Format(DateHired, "MMMM dd, yyyy"))  
End Sub
```

This would produce:



Built-In Time Functions

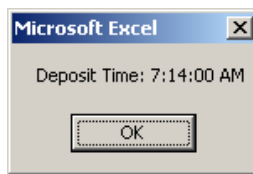
Introduction

The Visual Basic language supports time values. To create a time value, you can declare a variable of type **Date**. To initialize the variable, create a valid value using the rules specified in the Regional and language Settings of Control Panel, and include that value between two # signs. Here is an example:

```
Public Sub Exercise
    Dim DepositTime As Date

    DepositTime = #7:14#
    MsgBox("Deposit Time: " & DepositTime)
End Sub
```

This would produce:



The Current Time

To get the current time of the computer, you can call the Time function of the Visual Basic language. Here is an example:

```
Sub Exercise()
    Range("B2").FormulaR1C1 = Time
End Sub
```

To get a combination of the date and the time of the computer, you can call a function named Now. Here is an example:

```
Sub Exercise()
    Range("B2").FormulaR1C1 = Now
End Sub
```

In Microsoft Excel, to get a combination of the date and time of the computer, you can call a function named NOW. Here is an example:

```
Sub Exercise()
    Range("B2").FormulaR1C1 = "=NOW()"
End Sub
```

Creating a Time Value

Instead of including the time in # signs, you can also provide it as a string. To support this, the Visual Basic language provides a function named **TimeValue**. Its syntax is:

```
Function TimeValue(ByVal StringTime As String) As Variant
```

This function expects a valid time as argument. If that argument is valid, the function returns a time value. Here is an example:

```
Public Sub Exercise
    Dim DepositTime As Date

    DepositTime = TimeValue("7:14")
    MsgBox("Deposit Time: " & DepositTime)
End Sub
```

As an alternative to initializing a time variable, you can call a function named **TimeSerial**. Its syntax is:

```
Function TimeSerial(ByVal Hour As Integer, _
    ByVal Minute As Integer, _
    ByVal Second As Integer) As Variant
```

This function allows you to specify the hour, the minute, and the second values of a time. If you pass valid values, the function returns a time. Here is an example:

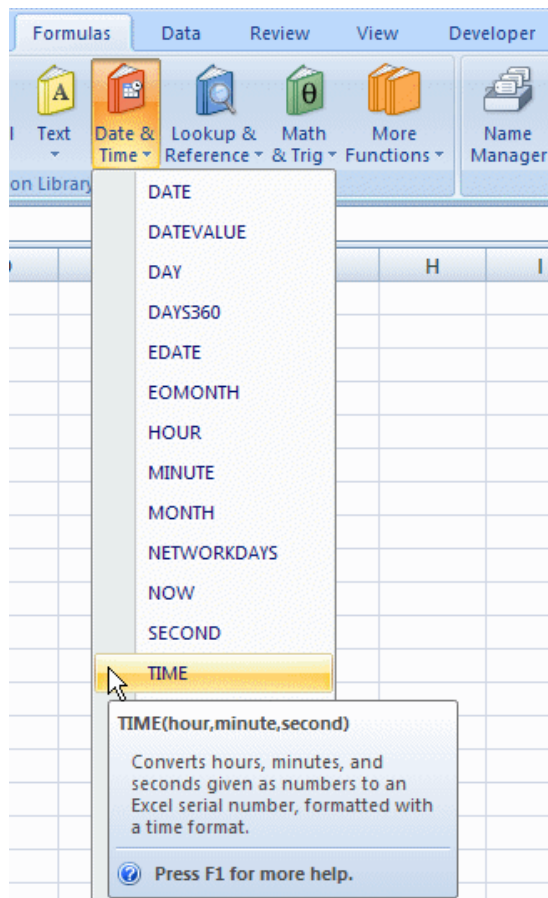
```
Public Sub Exercise
```

```

DepositTime = TimeSerial(7, 14, 0)
MsgBox("Deposit Time: " & DepositTime)
End Sub

```

To support the ability to create a time value, the Microsoft Excel library provides a function named **TIME**:



This function takes three arguments as the hour, the minute, and the second.

The Components of a Time Value

The Hours of a Day

In US English, a time is made of various parts. The first of them is the hour. The time is a 24th spatial division of a day. It is represented by a numeric value between 0 and 23. When creating a time value, you specify the hour on the left side. To get the hour of a valid time, you can call a function named **Hour**. Its syntax is:

```
Function Hour(ByVal TimeValue As Variant) As Integer
```

This function takes a time value as argument. If a valid time is passed, the function returns the hour part.

To support the hour part of a time value, the Microsoft Excel library provides a function named **Hour**.

The Minutes of an Hour

An hour is divided in 60 parts. Each part is called a minute and is represented by a numeric value between 0 and 59. If you have a time value and want to get its minute part, you can call a function named **Minute**. Its syntax is:

```
Function Minute(ByVal TimeValue As Variant) As Integer
```

When calling this function, pass it a time value. If the argument holds a valid value, the function returns a number between 0 and 59 and that represents the minutes.

To support the minute part of a time value, the Microsoft Excel library provides a function named **Minute**.

A minute is divided in 60 parts and each part is called a second. It is represented by a numeric value between 0 and 59. If you have a time value and want to extract a second part from it, you can call the **Second()** function named `.Second`. Its syntax is:

```
Public Function Second(ByVal TimeValue As Variant) As Integer
```

If you call this function, pass a valid time. If so, the function would return a number represents the seconds part.

To support the second part of a time value, the Microsoft Excel library provides a function named **SECOND**.

Operations on Date and Time Values

Introduction

Because dates and times are primarily considered as normal values, there are various operations you can perform on them. You can add or subtract a number of years or add or subtract a number of months, etc. The Visual Basic language provides its own mechanisms for performing such operations thanks to its vast library of functions.

Adding a Value to a Date or a Time

To support the addition of a value to a date or a time, the Visual Basic language provides a function named **DateAdd**. Its syntax is:

```
Function DateAdd( _
    ByVal Interval As String, _
    ByVal Number As Double, _
    ByVal DateValue As Object _
) As Variant
```

This function takes three arguments that all are required.

The *DateValue* argument is the date or time value on which you want to perform this operation. It must be a valid **Date** value.

The *Interval* argument is passed as a string. It specifies the kind of value you want to add. This argument will be enclosed between double quotes and can have one of the following values:

Interval	Used To Add
s	Second
n	Minute
h	Hour
w	Numeric Weekday
ww	Week of the Year
d	Day
y	Numeric Day of the Year
m	Month
q	Quarter
yyyy	Year

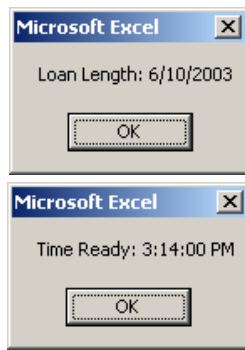
The *Number* argument specifies the number of *Interval* units you want to add to the *DateValue* value. If you set it as positive, its value will be added. Here are examples:

```
Public Sub Exercise
    Dim LoanStartDate As Date
    Dim DepositTime As Date

    LoanStartDate = #6/10/1998#
    DepositTime = TimeValue("7:14:00")

    MsgBox ("Loan Length: " & DateAdd("yyyy", 5, LoanStartDate))
    MsgBox ("Time Ready: " & DateAdd("h", 8, DepositTime))
End Sub
```

This would produce:



Subtracting a Value From a Date or a Time

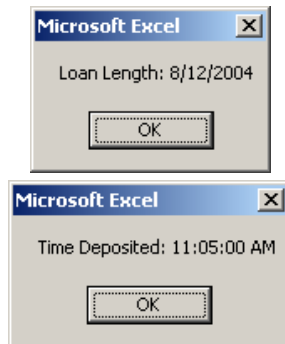
Instead of adding a value to a date or a time value, you may want to subtract. To perform this operation, pass the *Number* argument as a negative value. Here are examples:

```
Public Sub Exercise
    Dim LoanPayDate As Date
    Dim TimeReady As Date

    LoanPayDate = #8/12/2008#
    TimeReady = TimeValue("17:05")

    MsgBox ("Loan Length: " & DateAdd("m", -48, LoanPayDate))
    MsgBox ("Time Deposited: " & DateAdd("n", -360, TimeReady))
End Sub
```

This would produce:



The Difference Between Two Date or Time Values

Another valuable operation performed consists of finding the difference between two date or time values. To help you perform this operation, the Visual Basic language provides a function named **DateDiff**. This function allows you to find the number of seconds, minutes, hours, days, weeks, months, or years from two valid date or time values. The **DateDiff** function takes 5 arguments, 3 are required and 2 are optional.

The syntax of the function is

```
Function DateDiff( _
    ByVal Interval As [ DateInterval | String ], _
    ByVal Date1 As Variant, _
    ByVal Date2 As Variant, _
    Optional ByVal DayOfWeek As Integer = 1, _
    Optional ByVal WeekOfYear As Integer = 1 _
) As Long
```

This function takes five arguments, three of which are required and two are optional.

The *Date1* argument can be the start date or start time. The *Date2* argument can be the end date or end time. These two arguments can also be reversed, in which case the *Date2* argument can be the start date or start time and the *Date1* argument would be the end date or end time. These two values must be valid date or time values

The *Interval* argument specifies the type of value you want as a result. This argument will be enclosed between double quotes and can have one of the following values:

Interval	Used To Get
s	Second
n	Minute

h	Hour
w	Numeric Weekday
ww	Week of the Year
d	Day
y	Numeric Day of the Year
m	Month
q	Quarter
yyyy	Year

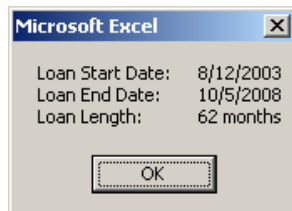
Here is an example:

```
Public Sub Exercise
    Dim LoanStartDate As Date
    Dim LoanEndDate As Date
    Dim Months As Long

    LoanStartDate = #8/12/2003#
    LoanEndDate = #10/5/2008#
    Months = DateDiff("m", LoanStartDate, LoanEndDate)

    MsgBox("Loan Start Date: " & vbTab & LoanStartDate & vbCrLf & _
        "Loan End Date: " & vbTab & LoanEndDate & vbCrLf & _
        "Loan Length: " & vbTab & Months & " months")
End Sub
```

This would produce:



By default, the days of a week are counted starting on Sunday. If you want to start counting those days on another day, supply the *Option1* argument using one of the following values: **vbSunday**, **vbMonday**, **vbTuesday**, **vbWednesday**, **vbThursday**, **vbFriday**, **vbSaturday**. There are other variances to that argument.

If your calculation involves weeks or finding the number of weeks, by default, the weeks are counted starting January 1st. If you want to count your weeks starting at a different date, use the *Option2* argument to specify where the function should start.

We saw that we could use the **DateDiff()** function to get the difference between two date or time values. The first argument can be specified as a string. A better idea is to use a member of the **DateInterval** enumeration. The members are:

Value	Constant Value	Description
vbUseSystemDayOfWeek	0	The interpreter will refer to the operating system to find out what day should be the first. In US English, this should be Sunday
vbSunday	1	Sunday (the default in US English)
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

By default, the first week of a year is the one that includes January 1st of that year. This is how it is considered in the regular date-based calculations. If you want to change this default setting, you can use the last argument of the **DateDiff()** function. The value of this argument can be:

Value	Constant Value	Description
vbUseSystem	0	The interpreter will refer to the operating system to find out what day should be the first. This should be the week that includes January 1st

vbFirstJan1	1	This will be the week that includes January 1st
vbFirstFourDays	2	This will be the first week that includes at least the first 4 days of the year
vbFirstFullWeek	3	This will be the first week that includes the first 7 4 days of the year

To calculate the number of days between two dates considering that a year has 360 days, you can use the **DAYS360()** function of the Microsoft Excel library.



Example Application: College Park Auto Repair

Description

This is an example of an application used by a fictitious car repair shop. The starting spreadsheet allows an employee to register a repair order. This includes a customer's name and the car.

To process an order, the employee must provide a list of auto parts that would have been used to repair the car. Then the clerk must enter the list of jobs that were performed for the order.

❖ Practical Learning: Introducing Workbooks

1. Start Microsoft Excel
2. Open the **CPAR1 workbook**
3. To save it, press F12
4. In the Save As Type combo box, select Excel Macro-Enabled Workbook (*.xlsm)
5. Change the name of the file to **College Park Auto Repair1**
6. Click Save
7. On the Ribbon, click Developer
8. In the Controls section of the Ribbon, click Insert
9. In the ActiveX Controls section, click Command Button
10. On the worksheet, click on the right side of Invoice #
11. Right-click the newly added button and click Properties
12. Using the Properties window, change the characteristics of the button as follows:
(Name): cmdOpenAutoRepair
Caption: open Auto Repair
13. In the Controls section of the [Ribbon](#), click Insert
14. In the ActiveX Controls section, click Command Button
15. Change its properties as follows:
(Name): cmdNewAutoRepair
Caption: New Auto Repair
16. In the Controls section of the Ribbon, click Insert
17. In the ActiveX Controls section, click Command Button (ActiveX Control)
18. On the worksheet, click under the previously added button
19. Using the Properties window, change the characteristics of the button as follows:
(Name): cmdSaveAutoRepair
Caption: Save and Close Auto Repair
20. Move and enlarge the button appropriately:

	A	B	C	D	E	F	G	H	I	J
1										
2	College Park Auto Repair									
3										
4	Invoice #:						Open Auto Repair			
5	Date:									
6										
7	Customer Information									
8	Name								
9	Address:								
10	City:	State:	ZIP Code:				
11	Car Information									
12	Make:	Model:						
13	Tag #:	Mileage:						
14										
15	Part #	Part Name	Unit Price	Qty	Sub Total					
16										
17										
18										
19										
20										
21										
22										
23	Job Performed								Rate	
24	
25	
26	
27	
28	
29	
30										
31	New Auto Repair									
32										
33	Save and Close Auto Repair									
34										
35										

Total Parts:	\$ -
Total Labor:	\$ -
Tax Rate:	
Tax Amount:	\$ -
Order Total:	\$ -

- 32. On the worksheet, right-click the New Auto Repair button and click View Code
- 33. Write the code as follows:

```

Option Explicit

Private AutoRepairExists As Boolean

Private Sub cmdNewAutoRepair_Click()
    AutoRepairExists = False

    Range("D4") = "": Range("D5") = Date: Range("D8") = ""
    Range("D9") = "": Range("D10") = "": Range("G10") = ""
    Range("J10") = "": Range("D12") = "": Range("G12") = ""
    Range("J12") = "": Range("D13") = "": Range("G13") = ""
    Range("B16") = "": Range("C16") = "": Range("H16") = ""
    Range("I16") = "": Range("J16") = "": Range("B17") = ""
    Range("C17") = "": Range("H17") = "": Range("I17") = ""
    Range("B18") = "": Range("C18") = "": Range("H18") = ""
    Range("I18") = "": Range("B19") = "": Range("C19") = ""
    Range("H19") = "": Range("I19") = "": Range("B20") = ""
    Range("C20") = "": Range("H20") = "": Range("I20") = ""
    Range("B21") = "": Range("C21") = "": Range("H21") = ""
    Range("I21") = "": Range("B24") = "": Range("J24") = ""
    Range("B25") = "": Range("J25") = "": Range("B26") = ""
    Range("J26") = "": Range("B27") = "": Range("J27") = ""
    Range("B28") = "": Range("J28") = "": Range("B29") = ""
    Range("J29") = "": Range("J33") = "5.75%"

    Range("D4").Select
End Sub
    
```

- 34. In the Object combo box, select cmdOpenAutoRepair
- 35. Implement its Click event as follows:

```

Private Sub cmdOpenAutoRepair_Click()
    Dim InvoiceNumber As String
    Dim Filename As String

    InvoiceNumber = Range("D4")
    AutoRepairExists = True

    If InvoiceNumber = "" Then
        MsgBox "You must enter an invoice number in Cell D4"
        Range("D4").Select
    Else
        Workbooks.Open InvoiceNumber & ".xlsx"
    End If
End Sub

```

36. In the Object combo box, select cmdSaveAutoRepair

37. Implement its Click event as follows:

```

Private Sub cmdSaveAutoRepair_Click()
    Dim InvoiceNumber As String
    Dim CurrentAutoRepair As Workbook

    InvoiceNumber = Range("D4")

    If InvoiceNumber = "" Then
        MsgBox "You must enter an invoice number in Cell D4"
        Range("D4").Select
    Else
        If AutoRepairExists = True Then
            Set CurrentAutoRepair = Workbooks(1)
            CurrentAutoRepair.Save
        Else
            Set CurrentAutoRepair = Workbooks(1)
            CurrentAutoRepair.SaveAs InvoiceNumber & ".xlsx"
        End If

        ActiveWorkbook.Close
    End If
End Sub

```

38. Return to Microsoft Excel

39. Create a repair order with an invoice number of 1001

40. Click Save Auto Repair

41. Click New Auto Repair

42. Return to Microsoft Excel