

Excel 2013/2016 Programming with VBA

WEXP16-ONLINE-06/29/16



Excel 2013/2016 Programming using Visual Basic for Applications

© 2016, Online Consulting, Inc. All Rights Reserved

No part of this manual may be reproduced in any form or by any means, or stored in a data base or retrieval system, without prior written permission of Online Consulting (except in the case of brief quotations embodied in critical articles and reviews). Making copies of any part of this manual for any purpose other than your own personal use is a violation of United States copyright laws. Products and brand names mentioned in this document are trademarks of their respective companies.

Online Consulting, 505 Carr Road, Suite 100, Wilmington, DE 19809
800-288-8221 ♦ <http://www.onlc.com>

Course Code: WEXP16-ONLINE-06/29/16

WEXP16: Excel 2013/2016 Programming – Setup Requirements

This course was developed and tested using Microsoft Office Professional Plus 2016 running on Windows 7. The course is appropriate for student using Excel 2013 or Excel 2016.

System Requirements

Each student and instructor computer should meet the system requirements for Office 2016 provided by Microsoft:

<https://products.office.com/en-us/office-system-requirements#Office2016-suites-section>

VBA for Excel Installation

Excel VBA is part of a complete Excel 2016 installation.

About the Course Data

The course data is compressed into a self-extracting executable file named **WEXP16.EXE**. The course is designed to access the data files from a folder on the C: drive named Data. If a folder named Data already exists, you should rename it or remove it before extracting the data.

In the \Data\Finished folder, you will find completed copies of the workbooks that are used for the exercises in the class. In the first section of the course, the projects are briefly reviewed to provide a preview of the work to be completed in the class.

Also in the Finished folder are additional folders named *Section1*, *Section2*, and so on, which correspond to the sections in the manual. Each folder contains the workbooks used in that section with the exercises completed through that point. If necessary, students can copy a completed workbook from one of these section folders to the Data folder before starting the next section.

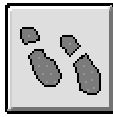
Section 8 does not have finished files because no permanent changes are made in the exercises in that section.

Downloading the Course Data

Follow the steps below to download and extract the course data to the Data folder using Internet Explorer. Alternatively, use a browser of your choice.

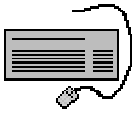
1. Open the Internet Explorer and navigate to www.onlc.com/data.
2. Scroll down and click the [Excel 2013/2016 Programming with VBA](#) link.
3. At the bottom of the Internet Explorer window, click **Run**.
4. If necessary, do the following:
 - At the bottom of the Internet Explorer window, click **Actions**;
 - Click **Run anyway**.
5. Note the destination folder for the data and then click **Extract**.

How To Use This Manual



STEPS

Step-by-step directions for carrying out a particular command or task. Use the Procedure for later reference.



Try It

Examples of how a procedure works. This is your guide during the hands-on exercises.



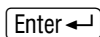
A note indicates optional information and provides a more complete understanding of a particular procedure, command, or feature.



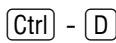
A shortcut gives you a tip or a hint that you can use to make a command or procedure more efficient.



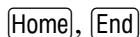
A trap warns you about a menu choice, command, or keystroke that may result in an error or unexpected results.



A word inside a box represents a key on the keyboard.



This PC key combination indicates two keys which must be held down simultaneously. Hold down the first key, tap the second key, and then release both keys.



This indicates a key sequence. Press the first key, release it, and then press the second key.

Program Manager

Any text in this typeface indicates the name of a folder, windows group, or icon.

Save

Any text in this typeface indicates a menu choice.

First Qtr.

Any text in this typeface indicates something that you will type as part of an exercise or part of a procedure.

If x>1 then

Any text in this typeface indicates program code that you will need to type in an exercise.

If x>1 then

Any text in this typeface and shading indicates program code which already exists. Use this as a reference for typing new code in programming exercises.

[argument]

A word in square brackets indicates it is optional. This is typically used in command syntax examples.



This symbol indicates that a command, line, or program code is the continuation of the previous line.

Contents

1. Getting Started.....	1-1
Introducing Visual Basic for Applications	1-2
Displaying the Developer Tab on the Ribbon	1-3
Recording a Macro	1-4
Saving a Macro-Enabled Workbook	1-6
Running a Macro	1-8
Editing a Macro in the Visual Basic Editor.....	1-10
Understanding the Development Environment	1-13
Using Visual Basic Help	1-19
Closing the Visual Basic Editor	1-21
Understanding Macro Security.....	1-22
2. Working with Procedures and Functions	2-1
Understanding Modules	2-2
Creating a Standard Module.....	2-3
Understanding Procedures.....	2-4
Creating a Sub Procedure	2-5
Calling Procedures	2-8
Using the Immediate Window to Call Procedures	2-9
Creating a Function Procedure.....	2-11
Naming Procedures	2-15
Working with the Code Editor	2-16
3. Understanding Objects	3-1
Understanding Objects	3-2
Navigating the Excel Object Hierarchy.....	3-3
Understanding Collections	3-4
Using the Object Browser	3-7
Working with Properties	3-11
Using the With Statement.....	3-13
Working with Methods.....	3-14
Creating an Event Procedure	3-17

4. Using Expressions, Variables, and Intrinsic Functions	4-1
Understanding Expressions and Statements	4-2
Declaring Variables	4-5
Understanding Data Types	4-8
Working with Variable Scope	4-11
Using Intrinsic Functions	4-14
Understanding Constants.....	4-19
Using Intrinsic Constants	4-20
Using Message Boxes	4-21
Using Input Boxes	4-26
Declaring and Using Object Variables	4-28
5. Controlling Program Execution.....	5-1
Understanding Control-of-Flow Structures.....	5-2
Working with Boolean Expressions	5-3
Using the If...End If Decision Structures	5-5
Using the Select Case...End Select Structure	5-12
Using the Do...Loop Structure.....	5-15
Using the For...To...Next Structure	5-18
Using the For Each...Next Structure.....	5-20
Guidelines for use of Control-of-Flow Structures.....	5-21
6. Working with Forms and Controls	6-1
Understanding UserForms.....	6-2
Using the Toolbox	6-4
Working with UserForm Properties, Events, and Methods.....	6-5
Understanding Controls.....	6-7
Setting Control Properties in the Properties Window	6-9
Working with the Label Control	6-10
Working with the Text Box Control.....	6-12
Working with the Command Button Control	6-14
Working with the Combo Box Control	6-16
Working with the Frame Control	6-18
Working with Option Button Controls	6-19
Working with Control Appearance	6-20
Setting the Tab Order	6-23

Populating a Control.....	6-25
Adding Code to Controls.....	6-27
Launching a Form in Code.....	6-31
7. Working with the PivotTable Object.....	7-1
Understanding PivotTables	7-2
Creating a PivotTable Using Worksheet Data.....	7-2
Creating a PivotTable Using Worksheet Data.....	7-3
Working with PivotTable Objects	7-6
Working with the PivotFields Collection	7-8
Assigning a Macro to the Quick Access Toolbar	7-16
8. Debugging Code.....	8-1
Understanding Errors	8-2
Using Debugging Tools.....	8-4
Setting Breakpoints	8-6
Stepping through Code.....	8-8
Using Break Mode during Run mode.....	8-11
Determining the Value of Expressions.....	8-12
9. Handling Errors	9-1
Understanding Error Handling	9-2
Understanding VBA's Error Trapping Options	9-3
Trapping Errors with the On Error Statement	9-6
Understanding the Err Object.....	9-7
Writing an Error-Handling Routine	9-8
Working with Inline Error Handling	9-11



Course Objectives

Record and edit macros

Use the Visual Basic Editor

Create sub and function procedures

Understand objects, properties, methods, and events

Explore the Excel object hierarchy and use the Object Browser

Work with variables and understand data types

Use intrinsic functions

Work with control-of-flow structures

Design UserForms and work with controls

Control PivotTables programmatically

Use debugging tools

Add error handling to code

A vertical yellow bar with a gradient, transitioning from a darker yellow at the top to a lighter yellow at the bottom, runs along the left side of the page.

Getting Started

Introducing Visual Basic for Applications

Displaying the Developer tab in the ribbon

Recording and running macros

Editing a macro in the Visual Basic Editor

Understanding the Visual Basic Development Environment

Using Visual Basic Help

Saving a macro-enabled workbook

Understanding macro security

Introduction to the course exercises

Introducing Visual Basic for Applications

Visual Basic for Applications or VBA refers to the development environment that is built into the Microsoft Office suite of products. By using the Visual Basic programming language, VBA allows you to create custom functionality and automate tasks in Office applications like Word and Excel.

Visual Basic is considered an object-oriented language, meaning that it works by accessing and manipulating objects. An object represents an element of an application like a document, worksheet, chart, or dialog box. In this course we will learn how to use the Visual Basic programming language and various Excel objects to write code that can control Excel. We'll start by using the macro recorder, which is a tool that converts user interface actions into Visual Basic code.

Displaying the Developer Tab on the Ribbon

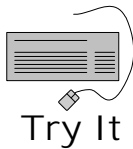
The Microsoft Office user interface consists of several tabs of commands that appear in an area called the Ribbon. The tools for working with macros and VBA are housed on the Developer tab which is not enabled by default in a typical installation of Office. The tab is easily enabled in the Excel Options window.



STEPS

Displaying the Developer Tab on the Ribbon

1. Click the File tab and then click Options.
2. Click the Customize Ribbon category.
3. In the drop-down list under Customize the Ribbon, make sure *Main Tabs* is selected.
4. In the list box under *Main Tabs*, select *Developer*.
5. Click OK.



Try It

Displaying the Developer Tab on the Ribbon

Note: The course data files are available at www.onlc.com/data and should be installed prior to beginning the course exercises. This course is designed to access the data files from a folder on the C: drive named Data.

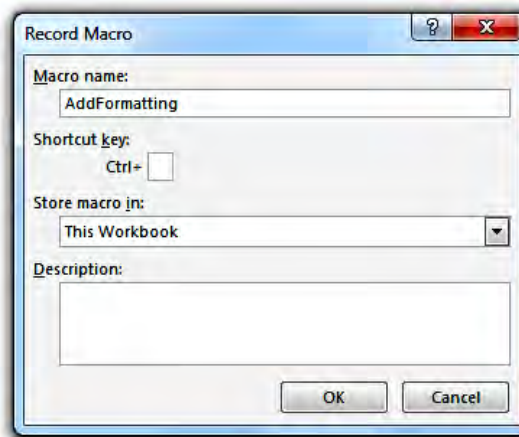
1. Open Excel.
2. If the Excel Start screen displays, click Blank Workbook.
3. If necessary, use the steps shown above to display the Developer tab on the Ribbon.

Recording a Macro

When you record a macro, it is saved as a series of commands in the Visual Basic programming language. You can record a sequence of editing, formatting, or other commands and replay those actions by running the macro. Recording a macro and examining the code that is generated is a useful way to gain some understanding of how Visual Basic works.

Macros can be stored locally in a workbook and can be shared with other workbooks. Excel also provides for macros to be made available globally by saving them in the Personal Macro workbook. This is a hidden workbook that automatically opens when you open Excel.





The following illustration displays the Record Macro dialog box:



Recording a Macro



STEPS

1. Do one of the following to start the recorder:
 - Click the Developer tab and locate the Code group; click  **Record Macro**.
 - On the Status bar, click .
2. Type the desired name in the Macro Name text box.
3. Make the desired selection from the *Store Macro In* drop-down list.
4. Type a description, if desired.
5. Click OK.
6. Perform the actions to be recorded.
7. To end the recording, perform one of the following:
 - On the Developer tab, click  **Stop Recording**.
 - On the Status Bar, click .



Macro names must begin with a letter and may contain letters and numbers and the underscore character; spaces and other special characters are not allowed.

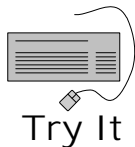
Information typed into the Description text box of the Record Macro dialog box will appear in the Visual Basic Editor as commented code within the procedure containing your macro.

To assign a macro to a keyboard shortcut at the time the macro is created, enter the desired character in the text box under *Shortcut Key* in the Record Macro dialog box.




The Personal Macro Workbook is named *Personal.xlsb* and is stored in *C:\Users\AppData\Roaming\Microsoft\Excel\XLSTART* folder. The workbook does not exist until you store a macro in it and save it.



If two macros share a shortcut key, the first one that appears in the Run Macro list will run when you press the shortcut.



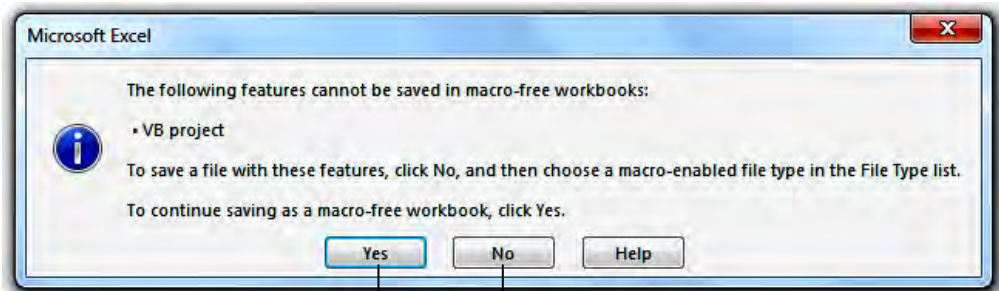
Recording a Macro

1. Open **Inventory.xlsx** located in the Data folder.
2. Click the Developer tab.
3. Locate the Code group and click  **Record Macro**.
4. Type `AddFormatting` in the Macro name text box.
5. Note the choices in the Store Macro In drop-down list and then make sure *This Workbook* is selected; click OK.
6. Perform the following actions:
 - Select cell A1;
 - Click the Home tab; in the Cells group open the Insert Cells drop-down and select *Insert Sheet Rows*;
 - Starting in cell A1, type the column headers as follows:
 VIN, Year, Make, Model, Classification, Color,
 Dealer Cost, MSRP
 - Select all of row 1 and change the formatting to Bold and Centered;
 - Select columns G and H;
 - On the Home tab, in the Number group, click  Accounting Number Format to apply the style to the selected columns;
 - Select columns A through H and AutoFit them.
7. Click the Developer tab and then click  **Stop Recording**.

Saving a Macro-Enabled Workbook

The file extension for an Excel workbook is **.xlsx**. For security purposes, macros cannot be saved in workbooks with this extension. To save a file containing macros, you need to save it to a special macro-enabled format. The extension for a macro-enabled workbook is **.xlsm**.

The illustration below shows the message that appears when you attempt to save an existing **.xlsx** file to which you have added macros:



Click **Yes** to save the file and retain its macro-free state. Macros that you have added to the file will be discarded.

Click **No** to open the Save As dialog box and save the file as macro-enabled by selecting the **.xlsm** file type in the Save As Type list.

Saving a Macro-Enabled Workbook

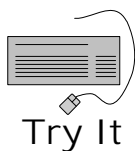


STEPS


1. In a workbook to be saved as macro-enabled, click the File tab and then click Save As.
2. Navigate to the location to save the file.
3. In the Save As dialog box, enter a name for the file.
4. Open the *Save As Type* drop-down list and select *Excel Macro-Enabled Workbook*.
5. Click Save.



If you add macros to an existing **.xlsx** file and attempt to save it, you will receive the message shown in the illustration shown above. If you click “Yes” in the message, the file will remain as an **.xlsx** and your macros will be discarded when you close the file.



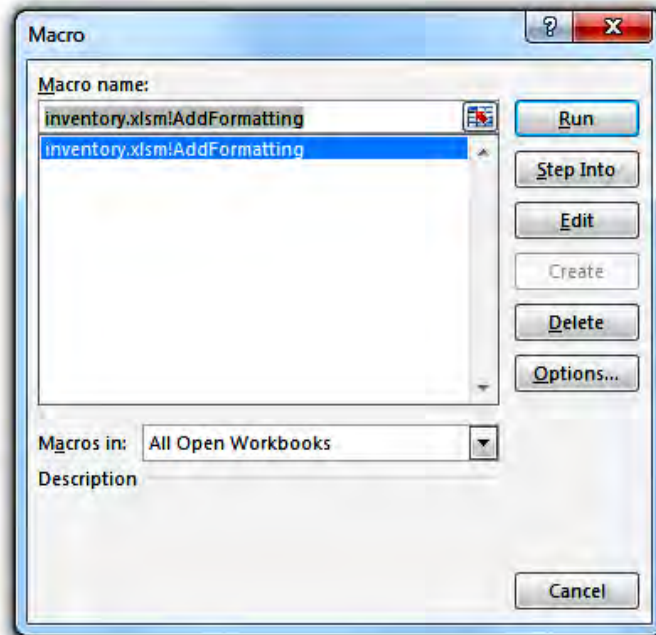
Saving a Macro-Enabled Workbook

1. In the **Inventory** workbook, click  Save.
 2. Read the message that appears and then click No.
 3. Under Current Folder, select the *Data* folder.
 4. In the Save As dialog box, open the *Save As Type* list and select *Excel Macro-Enabled Workbook*.
 5. Click Save.
-

Running a Macro

When you run a macro, it will execute the steps that were recorded. A macro can be run using the Macro dialog box, a keystroke combination, or a tool that is added to the Quick Access toolbar. The Macro dialog box provides a list of the available macros in all of the open workbooks.

Below is an illustration of the Macro dialog box:



Running a Macro



STEPS

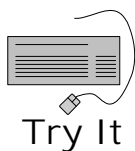
1. Open the Macro dialog box using one of the following methods:
 - Click the Developer tab; in the Code group, click Macros.
 - Press **[Alt] – [F8]**.
2. Select the desired macro from the Macro Name list.
3. Click Run.



Macro names without a workbook name in front of them indicate that they belong to the active workbook.

To assign a keyboard shortcut to an existing macro, select a macro from the list in the Macro dialog box and click Options; enter the desired character in the text box under Shortcut Key.

Click Step Into in the Macro dialog box to begin to run a macro in a mode that executes code statements one line at a time. Once the Visual Basic Editor displays, press **[F8]** to highlight the first executable code statement; press **[F8]** again to execute the statement. Continue pressing **[F8]** to step through the code. When in this mode, it is useful to display both the Excel and VB Editor windows in order to see the results of code execution.



Running a Macro

1. Open **Inventory(2).xlsx**.
 2. Click the Developer tab.
 3. In the Code group, click Macros.
 4. Select *All Open Workbooks* from the *Macros in* drop down list, if necessary.
 5. Select *Inventory.xlsm!AddFormatting*.
 6. Click Run.
-

Editing a Macro in the Visual Basic Editor

When you record a macro, the series of recorded instructions are inserted into a unit called a procedure whose beginning and end are denoted with the keywords **Sub** and **End Sub**. This procedure is stored within a *module*, which is a structure that can contain many procedures. You can create your own modules to organize your code and copy the procedures containing your macros into them.

The code that is generated when a macro is recorded can be modified or augmented to provide more custom functionality. Once you are familiar with the Visual Basic language you can start to create macros by entering code directly into procedures in the Visual Basic Editor's Code window, which functions in a similar manner as a word processor.


The following shows the code generated by the macro recorder for the AddFormatting macro:



Editing a Macro in the Visual Basic Editor




STEPS

1. Display the Macro dialog box.
2. Make a selection from the *Macros in* drop-down list.
3. Select the macro to be edited from the list under *Macro Name*.
4. Click Edit.
5. Make the desired changes in the Visual Basic Editor.
6. Click  Save.
7. Close the Visual Basic Editor window.



Press **[Alt] – [F8]** on the Status bar to quickly display the Macros dialog box.

Click  View Microsoft Excel from within the Visual Basic Editor to return to Excel.

Macros are saved in numbered modules. The first macro that you record in a workbook is stored in a module named *Module1*. Until you close the workbook, each subsequent macro that you record is stored there as well. When you reopen the workbook once it has been closed, any new macros that you record will be saved to a new module that is named using the next number in the sequence.

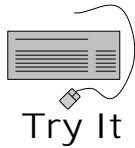


Press **[Alt] – [F11]** to quickly open the Visual Basic Editor from within Excel; this key combination can also be used to move back and forth between Excel and the Editor once the Editor is running.

You can also switch between Excel and the Visual Basic Editor using the taskbar.

Click the Developer tab and then click Visual Basic in the Code group to display the Visual Basic Editor from within Excel.

If you do not want to edit your macro, you can record the macro again using the same name. You will be prompted to replace the existing macro.





Editing a Macro in the Visual Basic Editor

1. Close **Inventory(2)** without saving changes and then open it again.
2. Press **[Alt] – [F8]** to display the Macro dialog box.
3. Select *Inventory.xlsm!AddFormatting* from the Macro Name list, if necessary.
4. Click Edit. The code for the AddFormatting macro you created earlier will appear in the Code window.
5. Scroll down in the Code window and find the portion of the code shown below. Delete the unshaded lines of code from your macro. (These lines are unnecessary because they set properties to default values. Removing them will have no effect on the execution of the macro.)

```
Rows("1:1").Select
Selection.Font.Bold = True
With Selection
    .HorizontalAlignment = xlCenter
    .VerticalAlignment = xlBottom
    .WrapText = False
    .Orientation = 0
    .AddIndent = False
    .IndentLevel = 0
    .ShrinkToFit = False
    .ReadingOrder = xlContext
    .MergeCells = False
End With
Columns("G:H").Select
Selection.Style = "Currency"
Columns("A:H").Select
Columns("A:H").EntireColumn.AutoFit
End Sub
```

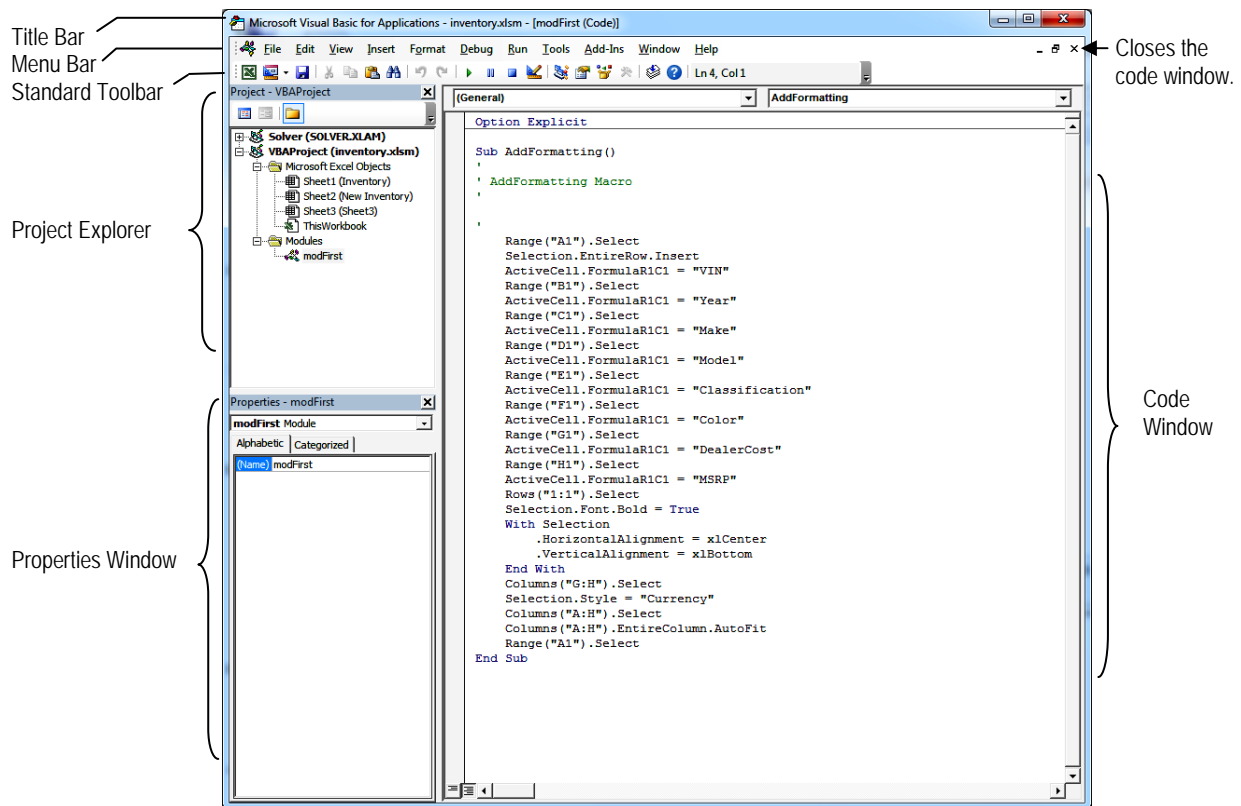
6. Immediately before the End Sub statement, insert the line of code shown in the box below. An editing tool called the *Auto List* will appear. Ignore it for now, it will be discussed later.

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

7. Click  Save.
8. Click  View Microsoft Excel in the Visual Basic Editor.
8. Run the *AddFormatting* macro. Notice the position of the cell pointer.
9. Close and save **Inventory(2)**.

Understanding the Development Environment

The Visual Basic desktop consists of several windows inside a unified development environment. The development environment is displayed below:



The windows shown in the illustration above may be used at various times when creating a project. A brief explanation of each window follows:

Title bar, menu bar, and Standard toolbar

are the center of the Visual Basic development environment. Closing this window closes the program. The menu bar and toolbar can be hidden, and you can customize each if you wish.

Project Explorer

provides an organized view of the files and components belonging to your project. When hidden, the Project Explorer can be displayed by pressing **Ctrl** - **R**.

Properties Window

provides a way to change and set the attributes of your forms and controls (such as their name, color, etc.). If hidden, the Properties window can quickly be displayed by pressing **F4**.

Code Window

is a window for editing the Visual Basic code. Pressing **F7** will open the code window for an object that is selected in the Project Explorer. When the Code window is maximized, the Close button that appears on the menu bar will close the window.

Understanding the Development Environment, continued:

The Menu Bar

The menu bar contains the commands used to perform most of the development tasks in Visual Basic for Applications. General characteristics of the menus found in the menu bar are listed below:

File	provides options for saving, printing, and closing the Visual Basic Editor as well as importing and exporting files.
Edit	provides cut, copy and paste options, as well as search commands and options for managing other features of the Code window.
View	provides selections to control the display, such as showing or hiding the many different windows.
Insert	provides commands for adding new procedures, modules, and classes to the project. Can be used to insert existing code text into a procedure.
Format	provides commands for positioning and manipulating controls placed on UserForms.
Debug	provides commands for eliminating errors in your code.
Run	provides commands used to test run the current procedure.
Tools	provides options to add references to other applications, set development environment options, and access project properties.
Add-Ins	provides access to the Add-In Manager dialog box.
Window	provides commands for controlling the orientation of the windows in the development environment.
Help	provides selections to access the Help system.












Right-clicking many of the objects within the development environment will display a shortcut menu giving you access to context-sensitive menu selections for the object.

Understanding the Development Environment, continued:

The Toolbar

The toolbar icons provide a graphical shortcut for accessing many of the commands found in the menu system. The Standard toolbar is visible when you open the Visual Basic Editor. Additional toolbars with icons for debugging, editing, and working with forms are also available.

Below are explanations for the pertinent icons on the Standard toolbar:

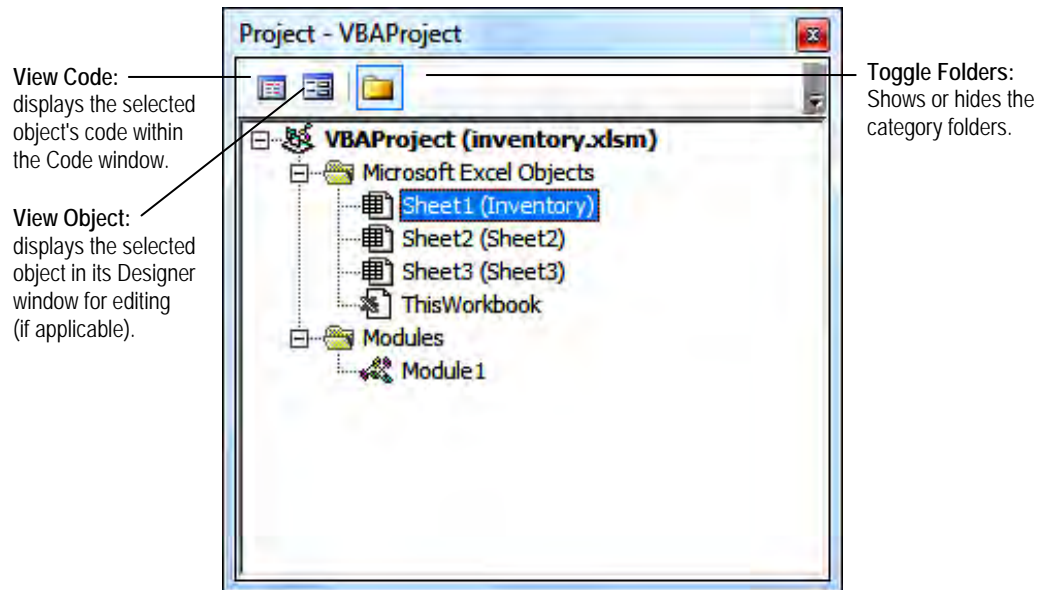
	View Microsoft Excel	returns to Excel.
	Insert <i><object></i>	adds a new module, class module, or procedure to the project. Click the drop-down arrow to select from the different objects that can be added. The name of the button changes to reflect the last object chosen.
	Run Macro	executes a procedure and continues program execution after entering break mode.
	Break	stops a running application and switches to break mode.
	Reset	terminates a running application.
	Design Mode	acts as a toggle switch for Design Mode.
	Project Explorer	displays or activates the Project Explorer.
	Properties Window	displays or activates the Properties window.
	Object Browser	displays or activates the Object Browser.
	Toolbox	acts as a toggle switch for the Toolbox.
	Microsoft Visual Basic Help	displays the Excel Help window.

Understanding the Development Environment, continued:

The Project Explorer

The Project Explorer lists all of the objects that have been created in, or added to, the current project. In Excel, a project and its objects are stored within a workbook file. Examples of objects include forms, worksheets, and code modules. The objects are listed under the project name by category, and then alphabetically within each category.

Below is an illustration of the Project Explorer:



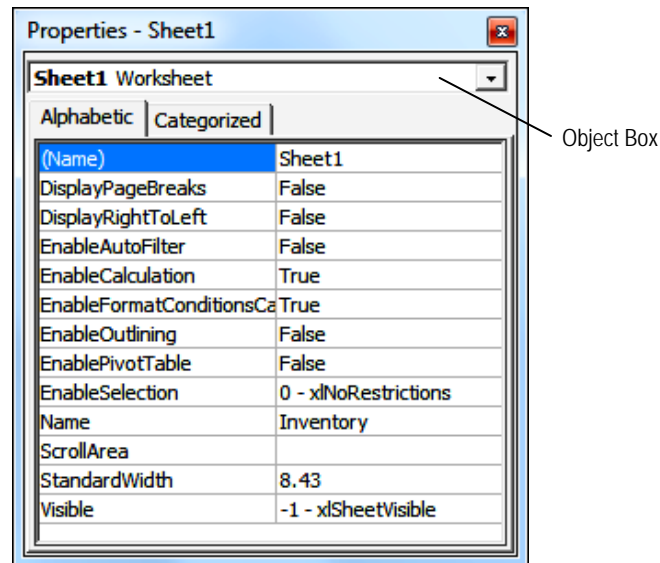
To expand a folder in the Project Explorer, click ☐ next to the folder; click ☐ to collapse a folder.

Understanding the Development Environment, continued:

The Properties Window

The Properties window in the Visual Basic Editor is used to display the properties for the selected object and their values. The properties are listed either alphabetically or by category by selecting the appropriate tab. The drop-down list at the top of the Properties window is used to select other elements contained within the selected object.

The Properties window with properties listed alphabetically is shown below:



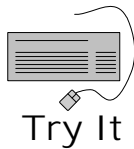
To expand a category in the Properties window, click next to the category. Click to collapse a category.

Using the Properties Window



STEPS

1. Display the Visual Basic Editor.
2. Select an object in the Project Explorer window.
3. If necessary, use one of the following methods to display the Properties window:
 - Open the View menu and select *Properties Window*.
 - Click Properties Window on the Standard toolbar.
 - Press .
4. To change the value of a property:
 - Select the name of the property in the left column.
 - Begin typing to enter the new property value in the right column or select the value from the drop-down list.

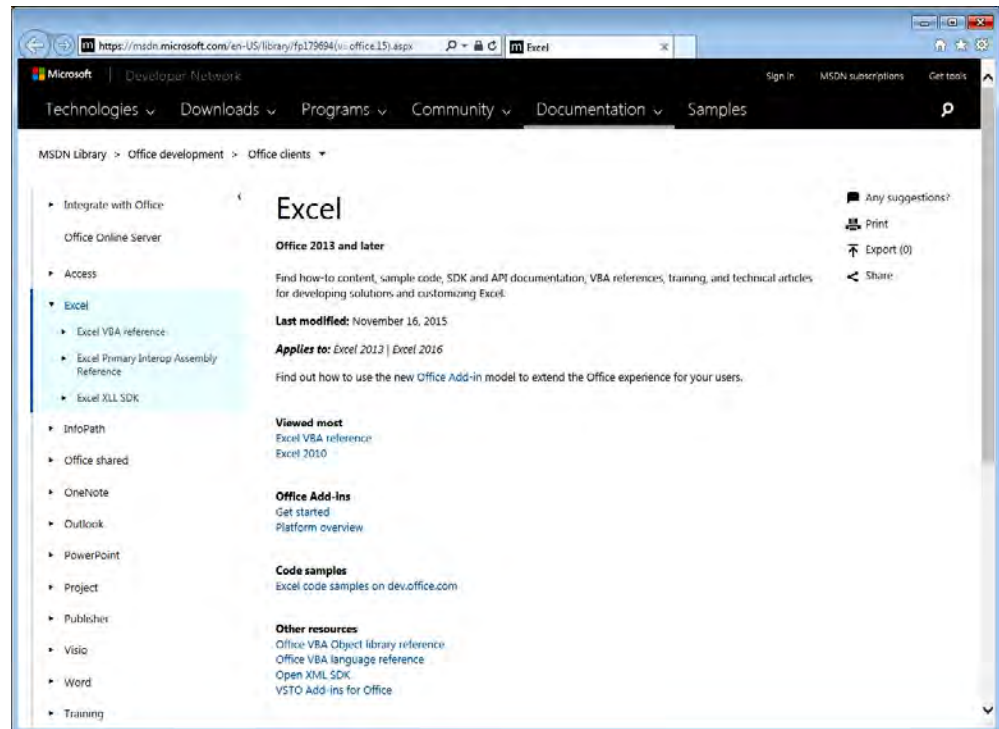


Understanding the Development Environment

1. Use the taskbar to switch to the Visual Basic Editor.
 2. Close the Code window.
 3. In the Project Explorer window, expand the Modules folder in the **Inventory.xlsm** project, if necessary.
 4. Select *Module1*.
 5. In the Properties window, change the Name property of Module1 to `modFirst`.
 6. In the Project Explorer window, right-click *Sheet1 (Inventory)* and select *View Object*. The worksheet will display in the Excel window.
 7. Return to the Visual Basic Editor.
 8. View the properties for Sheet1 that appear in the Properties window. Notice that there are two *Name* properties, one at the top in parentheses and another farther down the list.
 9. Select *Sheet2* in the Project Explorer.
 10. In the Properties window, change the Name property without parentheses to `New Inventory`.
 11. Press **[Alt] – [F11]** to return to the Excel window. Note that Sheet2 has been renamed.
 12. Press **[Alt] – [F11]** again to return to the Visual Basic Editor.
-

Using Visual Basic Help

Help for working with Excel VBA is available online from Microsoft's MSDN library. The site offers various resources including the Excel VBA reference and code samples. The picture below shows the page that displays when you access help from within the Visual Basic Editor:



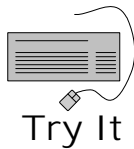


In the Visual Basic Editor, click  Microsoft Visual Basic for Applications Help on the Standard toolbar to open the MSDN library web page.





Several help pages include sample code to illustrate the use of the keywords and other elements used in Visual Basic. These examples can be copied and then modified for use in your code.

To find VBA help content not specific to Excel, search for *Visual Basic for Applications language reference*.



Using Visual Basic Help

1. In the Visual Basic Editor, open the Help menu and select *Microsoft Visual Basic for Applications Help*.
 2. In the content links on the left of the page, click Excel VBA reference.
 3. In the content links on the left, click Object model.
 4. In the content links on the left, scroll through the list of objects.
 5. Click the link for the Application Object and briefly review the information.
 6. At the top of the page, click .
 7. In the search text box, type `vba language reference` and then click  again.
 8. In the search results, click the link for the *VBA language reference*.
 9. In the content links on the left, expand *Visual Basic Conceptual Topics*.
 10. In the content links, select *Understanding Objects, Properties, Methods, and Events*.
 11. Briefly review the information on the page and then close the browser window.
 12. In the Project window in the Visual Basic Editor, right-click `modFirst` and select *View Code*.
 13. In the module that is displayed in the Code window, select the word “Sub”.
 14. Press **F1** and briefly review the help information.
 15. Follow the procedure link on the Sub Statement page. Scroll down in the VBE Glossary and read the definition for *procedure*.
 16. Close the browser window when you are finished.
-


Closing the Visual Basic Editor

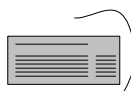
When you are finished using the Visual Basic Editor, you can close or minimize it. If you will not be using it again, it is best to close it. If you choose to minimize it, the Editor will be shut down when you close Excel.



STEPS

Closing the Visual Basic Editor

1. To close the Visual Basic Editor, perform one of the following:
 - Open the File menu; select *Close and Return to Microsoft Excel*.
 - Press **Alt** - **Q**.
 - Click  Close on the title bar.



Try It

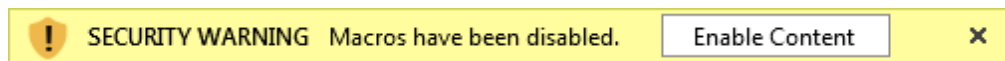
Closing the Visual Basic Editor

1. Use the desired method to close the VB Editor.
2. Close the **Inventory** workbook. If prompted, save the changes.

Understanding Macro Security

Because macros and other active content can potentially contain malicious code, it is important to control the circumstances under which macros are allowed to run on your computer. Macro security settings are handled in Microsoft Office 2013 via the Trust Center. In the Trust Center, you can designate trusted locations, view trusted publishers, and set other security options. Macros that are digitally signed by a trusted publisher will automatically be enabled. Likewise, trusted documents or files with macros that are stored in a trusted location will be able to run without a prompt for permission.

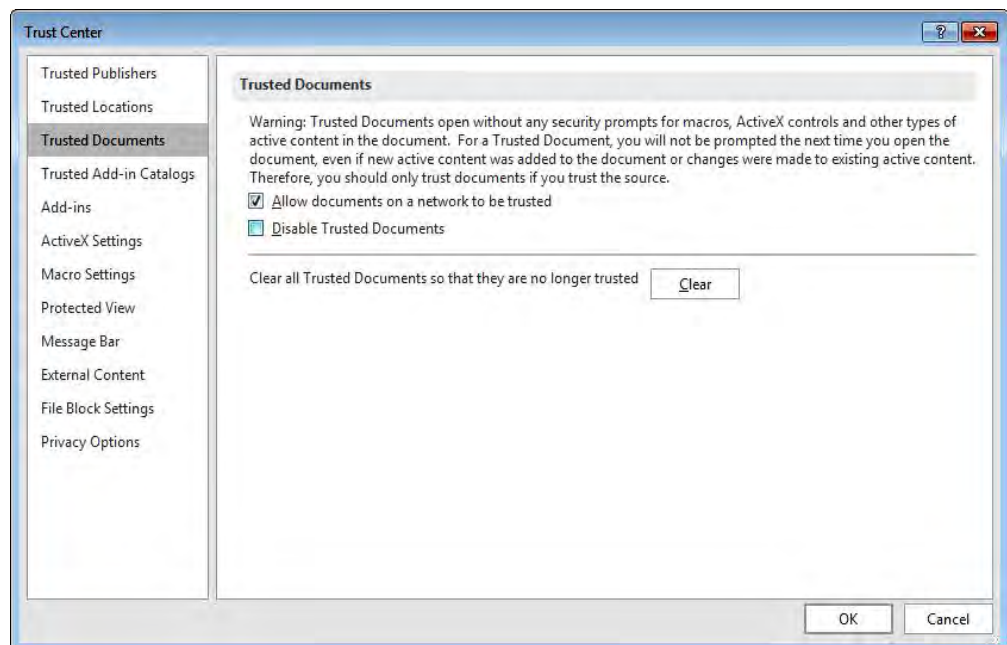
Under the default settings, opening an Excel file containing macros that is not a trusted document, or in a trusted location, or from a trusted source will trigger a security warning that appears above the Excel formula bar:



Trusted Documents

By clicking Enable Content on the message bar in a file with active content, you designate the file as a *Trusted Document*. The next time you open the file, you will not have to enable the content using the message bar. You should only trust documents that you know are from a reliable source.

The Trust Center provides options for disabling the Trusted Documents feature or clearing all Trusted Documents. The Trusted Documents category of the Trust Center is shown in the picture below:



Creating a Trusted Document



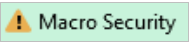
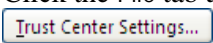
STEPS

1. To create a Trusted Document, open the document containing macros or other active content and do one of the following:
 - On the Message Bar, click Enable Content.
 - Click the File tab. In Backstage view, on the Info tab, click Enable Content and then select *Enable All Content*.
2. To enable active content in an open document for the current session only:
 - Click the File tab;
 - On the Info tab in Backstage view, click Enable Content and then select *Advanced Options*;
 - In the Microsoft Office Security Options box, select *Enable content for this session* and then click OK.

Opening the Trust Center



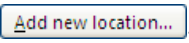
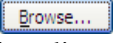
STEPS

1. To open the Trust Center, choose one of the following options:
 - Click the Developer tab and then click .
 - Click the File tab and then click Options; click Trust Center and then click .

Creating a Trusted Location in the Trust Center

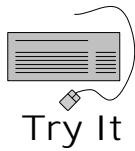


STEPS

1. Open the Trust Center.
2. Click Trusted Locations.
3. Click .
4. Click  and then navigate to the location to add to the Trusted Locations list; click OK.
5. If desired, select *Subfolders of this location are also trusted*.
6. In the Microsoft Office Trusted Location dialog box, click OK.
7. In the Trust Center, click OK.
8. Close the Excel Options dialog box, if necessary.



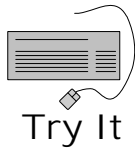
Make sure that any location that you designate as trusted is secure. Adding a network location as a trusted location is not recommended.



Creating a Trusted Location - Optional


Warning: It is important that folders to be designated as Trusted Locations are secure.

1. If appropriate for your classroom environment, use the steps shown above to add the folder where the course exercise files are located as a trusted location. (In a typical setup, the class files are located in C:\Data.)
 2. Make sure that you select the *Subfolders of this location are also trusted* option in the Microsoft Office Trusted Location dialog box.
-



Introduction to the Course Exercises

In this introductory exercise, we'll look at two workbooks that represent the finished files that we'll be creating throughout the course. In the course exercises, we'll add the procedures and other functionality that we'll review here.

1. Open the **Inventory** workbook that is located in the C:\Data\Finished folder.
 2. If you have not designated the student data folder as a trusted location, locate the Message Bar and click Enable Content. (Continue to enable content in this way as workbooks are opened throughout the remainder of the course.)
 3. Note the number of rows on the Inventory worksheet.
 4. Select the New Inventory worksheet.
 5. Click Yes at the prompt to run the GetNewInventory procedure.
 6. Click OK to accept the name and path of the new inventory file.
 7. Click Yes to append the inventory.
 8. Display the Inventory sheet and note the number of rows.
 9. Close the **Inventory** workbook without saving it.
 10. Open the **Fiscal Year Sales** workbook from the Data\Finished folder. Briefly look at the data in the workbook. It contains a worksheet for each day of sales for an imaginary car dealership.
 11. On the Quick Access Toolbar, click  Launch Reports.
 12. In the Run Reports dialog box, select *Month* under Period.
 13. Select *Classification* under Sales By.
 14. Select *Aug* from the drop-down list next to For Which Month.
 15. Click Display.
 16. If necessary, click Delete when prompted.
 17. A workbook named **Reports** is now open which contains a PivotTable. Briefly review the workbook and then close it without saving.
 18. Close the **Fiscal Year Sales** workbook. If prompted, do not save changes.
-



Working with Procedures and Functions

Understanding modules

Creating a standard module

Understanding procedures

Creating sub procedures and functions

Calling procedures

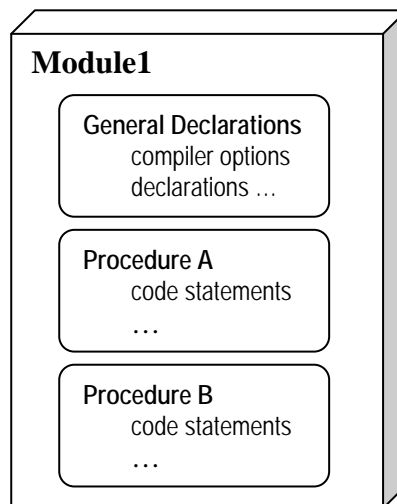
Using the Immediate window to call procedures

Naming procedures

Working with the Code Editor

Understanding Modules

Procedure is a general term that refers to a unit of code that is created to perform a specific task. In Excel, procedures are stored in objects called *modules*. In addition to being a container for procedures, modules also provide an area in which compiler options and various declarations may be stored. This area is located at the very top of a module and is referred to as the *General Declarations* section. The first procedure encountered within the module marks the end of the General Declarations section. From that point on, the module will only contains procedures.



The types of modules are defined below:

Standard modules	contain procedures that can be called from anywhere in the project, as well as other applications.
Built-in Class modules	are associated and stored within an object such as a form or worksheet. They may contain procedures used to respond to events for that form or worksheet. They may also include general procedures that perform tasks in support of the form or worksheet.


Creating a Standard Module

Standard modules can be used to store procedures that are available to all forms, worksheets, and other modules. These procedures are usually generic in nature and can be called by any other procedure while the workbook is open. You can create as many standard modules as necessary within the project, storing related procedures together within the same module. In addition, standard modules are used to declare global variables and constants.

Creating a Standard Module

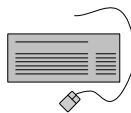


STEPS

1. To create a standard module in the Visual Basic Editor, perform one of the following:
 - Open the Insert menu; select *Module*.
 - On the Standard toolbar, click the down arrow next to  Insert *Object* and select *Module*.



In the Visual Basic Editor, expand the Modules folder in the Project window to display the standard modules for the workbook. Double-click a module to open it in the Code window.



Try It

Creating a Standard Module

1. Create a new workbook.
2. Click the Developer tab. In the Code group, click Visual Basic to display the Visual Basic Editor.
3. In the VB Editor, open the Insert menu and select *Module*.
4. Display the Properties window, if necessary.
5. In the Properties window, change the name of the module to `modPractice`.

Understanding Procedures

A procedure is a named set of statements that does something useful within your application. Every line of executable code must be stored within a procedure. You execute the code stored in a procedure by referring to that procedure by name from within another procedure; this is known as *calling* a procedure. When a procedure is done executing, it returns control to the procedure from which it was called.

Visual Basic provides two general types of procedures:

Sub procedures	perform a task and return control to the calling procedure.
Function procedures	perform a task and return a value, as well as control, to the calling procedure.

Both sub and function procedures may define argument variables in order to accept data from the calling procedure. This data is used by the procedure to perform its task. Arguments may be defined as required or optional.

Creating a Sub Procedure

Most Excel tasks can be automated by creating procedures, either by recording a macro or by entering them directly into the Visual Basic Editor's Code window. Some tasks accept arguments but do not return values. Sub procedures are used to perform actions when a return value is not needed.

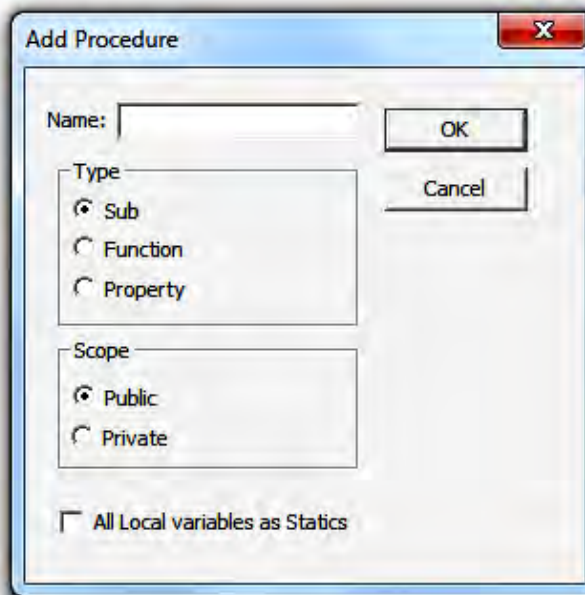
Below is the syntax for a sub procedure:

```
[Public/Private] Sub ProcedureName ([argument list])  
    statement block  
End Sub
```

Public indicates that the procedure can be called by procedures stored within other modules. It is the default setting if not explicitly declared.

Private indicates that the procedure is available only to other procedures contained within the same module.


The **Sub...End Sub** structure can be typed directly into the Code window to start a new procedure. The structure may also be inserted using the Add Procedure dialog box shown below:



Creating a Sub Procedure



STEPS

1. Create or display the module to contain the sub procedure.
2. Click within the Code window.
3. To display the Add Procedure dialog box, perform one of the following:
 - Open the Insert menu; select *Procedure*.
 - On the Standard toolbar, click the down arrow next to  Insert *Object* and choose *Procedure*.
4. Type the name of the procedure in the *Name* text box.
5. Select *Sub* under *Type*, if necessary.
6. Make the desired selection under *Scope*.
7. Click OK.



VBA automatically places parentheses after the sub procedure name in the Code window. These are necessary and must enclose any arguments that the procedure is designed to receive.

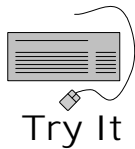
Public function and sub procedures within standard modules should have unique names within the current workbook.

Sub procedures can be placed anywhere after the General Declarations section.

The order in which procedures appear in a module has no effect upon the program's operation.




You can create a sub procedure by simply typing the keyword **Sub** followed by a space and the name of the procedure on an empty line in the Code window. Press **Enter** and the Visual Basic Editor will place parentheses after the name and insert the **End Sub** keywords to mark the end of the procedure.




Creating a Sub Procedure

In this exercise we will create a sub procedure that displays a message box with text that we specify.

1. Check the information in the title bar to verify that the code for modPractice is displayed in the Code window. Alternatively, double-click modPractice in the Project window.
2. Click in the Code window, if necessary.
3. Open the Insert menu and choose *Procedure*.
4. Type `Message` in the *Name* text box. Make sure that *Sub* is selected under *Type* and click OK.
5. Press  to indent and then enter code so that the procedure resembles the following:

```
Public Sub Message()  
    MsgBox "Hello World!"  
End Sub
```

6. Click  Save.
 7. Type `Practice` as the name for the workbook.
 8. If necessary, navigate to the Data folder as the location to save the file.
 9. Open the *Save as type* list and select *Excel Macro-Enabled Workbook*.
 10. Click Save in the Save As dialog box.
-

Calling Procedures

You call a sub procedure or function at the point in another procedure where you want its code to execute. The sub procedure or function that is called must be accessible to the calling procedure, meaning that it must be in the same module or declared publicly in another module within the same project. VBA will find the procedure within the current workbook, regardless of which standard module contains the code, as long as the procedure was not declared privately.

Passing Values to Arguments

A call consists of naming the sub procedure or function and providing the necessary or desired values for the arguments. The argument list of the function or sub procedure names the set of variables that receive the data passed to it by the calling statement. The values you name in the calling statement do not have to match the argument names used in the function or sub procedure declaration, but they must match the data types. A passed value can be a literal, variable, constant or other expression. A call to a function procedure is typically found on the right side of an assignment statement so that the return value is stored to a variable.

Below are examples of calls to sub procedures:

```
AddFormatting  
PrintReport "EmployeeReport"
```

Below are examples of calls to function procedures:

```
Volume = CubeVolume(3,7,8)  
NameString = FixName(txtFirstName, txtLastName)
```

Auto Quick Info is a feature of the Visual Basic environment that displays a syntax box when you type a procedure or function name showing the list of arguments that the procedure will accept. The illustration below shows the Auto Quick Info tip for the built-in Message Box function of Visual Basic:

```
Public Sub Message()  
    MsgBox |  
End MsgBox(Prompt, [Buttons As VbMsgBoxStyle = vbOKOnly], [Title], [HelpFile], [Context])  
    As VbMsgBoxResult
```



When passing multiple arguments, separate them with commas and pass them in the same order in which they are listed in the procedure syntax. If you skip an argument, use a comma as a placeholder before entering the next value in the list.

Enclose arguments in parentheses when calling a function from which you will use the return value.

Arguments that appear in the syntax box in square brackets are optional.

Values that are passed to procedure arguments are sometimes referred to as parameters.





Using the Immediate Window to Call Procedures

The Immediate window is an interactive debugging feature of the Visual Basic environment that can be used to enter commands and evaluate expressions. In a similar manner, you can execute the code stored in a sub or function procedure by calling the procedure from the Immediate window.



STEPS


Using the Immediate Window to Call Procedures

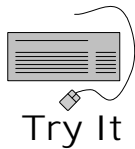
1. To open the Immediate window, perform one of the following:
 - Open the View menu; select *Immediate Window*.
 - Click  Immediate Window on the Debug toolbar.
 - Press **Ctrl** – **G**.
2. To execute a sub procedure:
 - Type: `SubProcedureName [argument list]`
 - Press **Enter** .
3. To execute a function procedure and print its return value in the window:
 - Type: `? FunctionName([argument list])`
 - Press **Enter** .
4. To evaluate an expression:
 - Type: `? expression`
 - Press **Enter** .



Within code, use the **Debug.Print** statement to display values in the Immediate window, while the code is executing. The Immediate window must be open in order to see the results of **Debug.Print** statements.



To re-execute a statement in the Immediate window, place the insertion point anywhere in the statement, edit as necessary, and press **Enter** .



Using the Immediate Window to Call Procedures

1. Open the View menu and select *Immediate Window*.
2. In the Immediate window, type: `message`
3. Press `Enter`.
4. Click OK to close the message box.
5. In the Code window below the Message procedure, type `Sub NewMessage` and press `Enter`.
6. Modify the procedure to match the code below:

```
Sub NewMessage(msgText)
    MsgBox msgText
End Sub
```

7. In the Immediate window, call the new procedure and supply the `msgText` argument by typing the following and pressing `Enter`:

```
NewMessage "Hello World!"
```

8. Click OK to close the message box.
 9. Call the `NewMessage` procedure again supplying a different text message; then close the message box.
 10. In the Immediate window, type `NewMessage` and press `Enter` to call the procedure without supplying an argument.
 11. Click OK to close the error Message Box.
-

Creating a Function Procedure

Function procedures are similar to *intrinsic* or built-in functions such as Sum() or Average() and are sometimes called *user-defined* functions. A function is distinguished from a sub procedure in that it returns a value to the procedure that calls it. The value that the function generates is assigned to the name of the function within the code of the function procedure.

The following is the syntax for a function procedure followed by an example:

```
[Public/Private] Function FunctionName [(argument list)] [As <Type>]  
[statement block]  
FunctionName = <expression>  
End Function
```

```
Private Function CubeVolume(Width, Length, Height) as Double  
    CubeVolume = Width * Length * Height  
End Function
```

Public indicates that the procedure can be called by procedures stored within other modules. It is the default setting if not explicitly declared.

Private indicates that the procedure is available only to other procedures contained within the same module.

The **As** clause sets the data type of the function's return value.

An **As** clause can also be included after each argument to set its data type. Data types for arguments in sub procedures are also defined in this same manner. The example below shows the **As** clause being used to set the data type for each argument of the function as well as the function's return value:

```
Function CubeVolume(Width As Double, Length As Double , Height As Double) As Double  
    CubeVolume = Width * Length * Height  
End Function
```

When calling a function you pass the values that the function needs to perform its task. The values passed must match the data types of the receiving arguments. To call the CubeVolume function shown in the previous example, you might use a statement similar to the following:

```
dblVolume = CubeVolume(dblWidth, dblLength, dblHeight)
```

Variable name to hold
result of function


Function name

Required values for the function arguments


Creating a Function Procedure



STEPS

1. Create or display the module to contain the function procedure.
2. Click in the Code window.
3. To display the Insert Procedure dialog box, perform one of the following:
 - Open the Insert menu; select *Procedure*.
 - Click the Insert down arrow next to  Insert *Object* and choose *Procedure*.
4. Type the name of the procedure in the *Name* text box.
5. Select *Function* under *Type*, if necessary.
6. Change other procedure options as desired.
7. Click OK.




You can call a public function that you create in a standard module from the Excel window using the Insert Function dialog box. In the workbook containing the function, click  Insert Function next to the Formula bar. In the Insert Function dialog box, select *User Defined* from the *Or Select a Category* drop-down list.

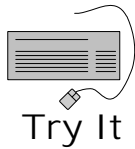
VBA automatically places parentheses after the function name in the module window. These are necessary and must enclose any arguments that the procedure is designed to receive.

Public function and sub procedures within standard modules should have unique names within the current project.

A value that is passed to an argument may be a literal, constant, variable, or other expression that represents a value. A literal is an actual value, a constant represents a value that doesn't change, and a variable represents a value that can change.



You can also create a function procedure by typing the keyword **Function** followed by a space and the name of the function on an empty line in the Code window. Then press  and the Visual Basic Editor will place parentheses after the name and insert the **End Function** keywords to mark the end of the procedure.



Creating a Function Procedure

In this exercise we will create a function that accepts a number as its argument and returns that number squared. Then we'll create a procedure that calls the function.

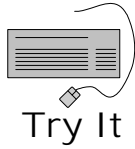
1. Click within the Code window for modPractice, if necessary.
2. Open the Insert menu and select *Procedure*.
3. Type `Squared` in the *Name* text box.
4. Select *Function* under *Type* and click OK.
5. Modify the procedure so it matches the code shown below:

```
Public Function Squared(Number)
    Squared = Number ^ 2
End Function
```


6. Save the module.
7. In the Immediate Window, type `? Squared(7)` and press Enter to call the function.
8. In the Code window, select the `NewMessage` procedure, make a copy of the code and paste it below the `Squared` function.
9. Modify the copied procedure to match the code below:

```
Sub SquareMessage(SquareIt)
    MsgBox Squared(SquareIt)
End Sub
```

10. Call the `SquareMessage` procedure in the Immediate window and supply a numeric argument.
 11. Close the message box.
-



Bonus Exercise - Calling a User-Defined Function from Excel

1. Switch to the Excel window.
 2. On Sheet1, enter a three-digit number in cell A1.
 3. Move to cell A2.
 4. Click  Insert Function on the Formula bar.
 5. Select *User Defined* from the *Or select a category* drop-down list.
 6. Select *Squared* in the *Select a function* list and click OK.
 7. In the Function Arguments dialog box, enter A1 in the text box next to Number and click OK.
 8. Note the result of the function.
-

Naming Procedures

What follows are the rules and conventions for naming procedures in Visual Basic. While you must follow the rules or you will generate an error, following a convention is not required. A convention is a standard or guideline that may help to make your code easier for you or someone else to understand.

Adhere to the following *rules* when naming procedures:

- The maximum length for a procedure name is 255 characters.
- The first character must be a letter.
- Procedure names cannot contain spaces or any of the following characters:
., @ & \$ # () !
- Procedure names must be unique within a given module.

Consider these naming *conventions* when naming procedures:

- As procedures are generally used to carry out some action, begin their names with a verb.
- Use proper case for each word within the name of a procedure.
- When two or more procedures are related, attempt to place the words that vary at the end of the name.

Following the conventions listed above would create procedure names such as:

```
PrintInventoryList()  
GetSelectedDateBegin()  
GetSelectedDateEnd()
```

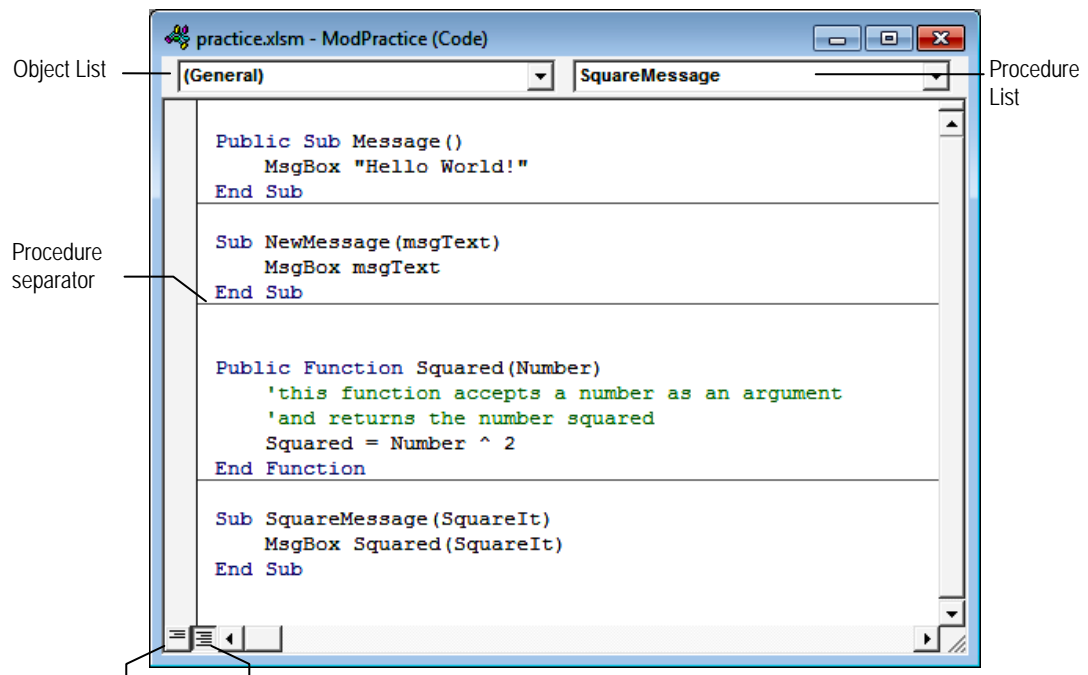
Additional information to consider when naming procedures is shown below:

- The names of public procedures stored within standard modules should be unique throughout the entire application. This prevents detection of ambiguous references and makes your program easier to understand.
- Likewise, avoid using names that Visual Basic uses for its many public functions, methods, and keywords.
- Visual Basic is not case-sensitive, but the case used in the declaration of a procedure name is preserved.

Working with the Code Editor

The Code window portion of the environment is used to edit Visual Basic code. It contains two drop-down lists that can be used to display different procedures within a standard module or various objects' event procedures within a class module.

Below is an illustration of the Code window followed by a description of the Object and Procedure drop-down lists:



Procedure View:
displays procedures
one at a time.

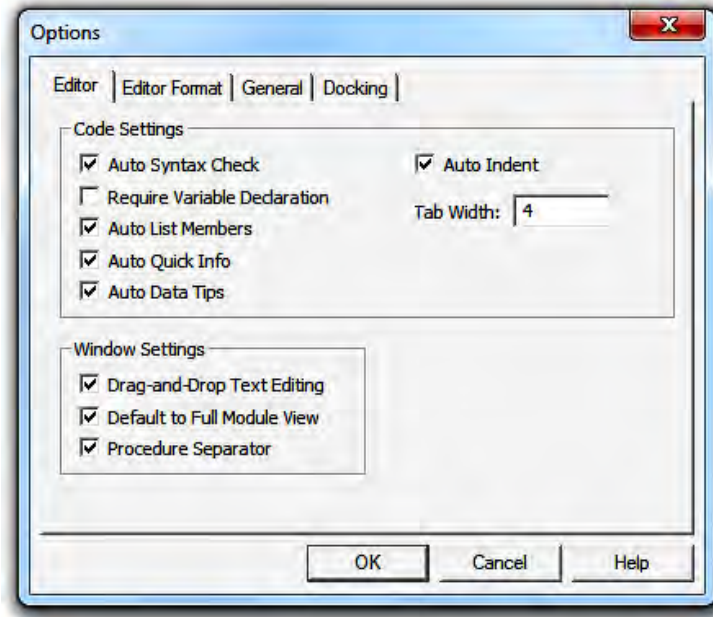
Full Module View:
displays all the procedures in the
module one after the other.

Object List	displays a list of objects contained in the current module.
Procedure List	displays a list of general procedures contained in the current module when General is selected in the Object list. When an object is selected in the Object list, a list of events associated with the object is displayed.

Working with the Code Editor, continued:

Setting Code Editor Options

Settings for the Code window can be changed on the Editor tab of the Options dialog box, illustrated below:



The following is an explanation of the Code Settings selections:

<i>Auto Syntax Check</i>	automatically displays a help message when a syntax error is detected. The message appears when you move off the line of code containing the error.
<i>Require Variable Declaration</i>	adds the line <i>Option Explicit</i> to all newly created modules, requiring all variables to be explicitly declared before they are used in a statement.
<i>Auto List Members</i>	displays a list box under your insertion point after you type an identifiable object or member name. The list shows all members of the object's class (as can be seen in the Object Browser) or items that would complete the next part of the statement. An item selected from the list can be inserted into your code by pressing Tab .
<i>Auto Quick Info</i>	displays a syntax box when you type a method, procedure, or function name showing a list of arguments.
<i>Auto Data Tips</i>	displays the value of a variable when you point to it with the mouse during break mode. This is useful when debugging.
<i>Auto Indent</i>	indents the specified amount when you press Tab and indents all subsequent lines at the same level.

Working with the Code Editor, continued:

The Window Settings selections are explained below:

<i>Drag-and-Drop Text Editing</i>	allows you to drag and drop code around the Code window and into other windows, like the Immediate window.
<i>Default to Full Module View</i>	displays all module procedures in one list with optional separator lines between each procedure. The alternative is to show one procedure at a time, as selected through the Procedure list.
<i>Procedure Separator</i>	displays a gray separator line between procedures if Module view is selected.

Editing Guidelines

Try to observe the following guidelines when editing code:

- When entering a statement, you may carry it over to the next line by typing a space and an underscore (`_`) character at the end of the line. This also works for comments. Strings that are continued require a closing quote, an ampersand (&), and a space before the underscore. This is called *command line continuation*.
- You should indent text within control structures for readability. To indent text, select one or more lines, and press `Tab`. To remove an indent, press `⇧ Shift` – `Tab`. There are buttons on the Edit toolbar for indenting and outdenting selected lines of code.
- Complete statements by pressing `Enter` at the end of the line or by switching focus away from the line by clicking somewhere else with the mouse or pressing an arrow key.



When you press `Enter` moving the focus off a line of code, the code formatter automatically places keywords in proper case, adjusts spacing, adds punctuation, and standardizes variable capitalization.

Commenting Code

It is usually a good idea to include comments in your code to document what is happening in your project. It can be very helpful if you have to revise it later. A good commenting practice is to comment what isn't obvious and stop there. When testing and debugging, you may also want to comment out a line of code so that it does not execute.

You can include comment lines in your code by starting the line with an apostrophe (`'`) or by typing the keyword **Rem** (for remark). When you use the apostrophe to include a comment, you can place that comment at the end of a line containing another statement without causing a syntax error. Tools for commenting and uncommenting code can be found on the Edit toolbar.

Working with the Code Editor, continued:

Navigating in the Code window

While working in the Code window, you will often find it necessary to display other procedures in the module.



STEPS

Navigating in the Code Window

1. Open the desired module.
2. To display a specific object:
 - Open the Object drop-down list;
 - Select the desired object.
3. To display a specific procedure:
 - Open the Procedure drop-down list;
 - Select the desired procedure.



You can print a procedure from the Code window. Select the desired procedure, open the File menu and select *Print*. Choose *Selection* under Range and click OK.



Set code options such as text color and font on the Editor Format tab of the Options dialog box.

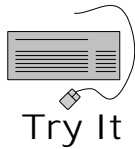
The Find and Replace feature can be used to search for text in the Code window.

Select the name of a function or sub procedure within a calling procedure and press **Shift** – **F2** to quickly display its code. Use **Ctrl** – **Shift** – **F2** to return to the code of the calling procedure.






Press **Ctrl** – **Spacebar** at any point in the Code window to turn on the Complete Word feature. This will provide an alphabetically sorted list of items available to your current line of code. Scroll or continue to type to navigate in the list. Press **Tab** to insert the selection into the code.









Switch between views by clicking either  Full Module view or  Procedure View in the lower-left corner of the Code window.



Working with the Code Editor

1. Return to the Visual Basic Editor.
2. Click  Procedure View in the lower-left corner of the Code window.
3. Open the Procedure drop-down list and select *Message*.
4. Display the SquareMessage procedure.
5. In the MsgBox statement in the SquareMessage procedure, select the word Squared that refers to the function procedure and press  – .
6. In the Squared function procedure, enter the commented text as shown below:

```
Public Function Squared(Number)
    ' this function accepts a number as an argument
    ' and returns the number squared
    Squared = Number ^ 2
End Function
```

7. Click  Full Module View.
8. Display the Edit toolbar.
9. Select the Message and NewMessage sub procedures.
10. Click  Comment Block.
11. Click  Procedure View if you would prefer to work in that view.
12. To practice using the Auto List Members and Complete Word features do the following:
 - If necessary, display the Immediate window.
 - Type ? followed by a space. Type the letter w and then press  – .
 - Use the down arrow key to select *Worksheets* from the list and then press  to insert it.
 - Type a period and then select *Count* from the list. Press .
 - Press . (The Immediate window returns the value of the expression.)
 - Type ? followed by a space; use the features to enter the following expression into the Immediate window: `Activeworkbook.ActiveSheet.Name`
13. Close the Code window for modPractice.
14. Save the **Practice** workbook.

3

Understanding Objects

Understanding classes and objects

Navigating the Excel Object Hierarchy

Understanding Collections

Using the Object Browser

Working with properties, methods, and events

Using the With statement

Understanding Objects

An object is an element of an application that can be accessed and manipulated programmatically using Visual Basic. Examples of objects in Excel are worksheets, charts, and ranges. Objects are defined by lists of *properties* and *methods*, and many allow for custom sub procedures to be executed in response to various *events*.

The term *class* refers to the general structure of an object. You can think of a class as a template that defines the elements that all the objects within that class share.

Properties

Properties are attributes or characteristics of an object. The data values assigned to properties describe a specific instance of an object. When you create a new workbook in Excel you are creating an instance of a Workbook object based on the Workbook class. Some properties that would help to define an instance of a Workbook object include its name, whether it has a password, its path, etc.

Methods

While properties describe characteristics, methods represent procedures that perform actions. Printing a worksheet, saving a workbook, and selecting a range are all examples of actions that can be executed using a method.

Events

Many objects can recognize and respond to events. For each event that an object can recognize, you have an opportunity to write a sub procedure that will be executed when the specified event occurs. For example, a workbook can recognize the Open event. Code inserted into the Open event procedure for a workbook would run whenever the workbook was opened. If you do not write any code for an object's event, when the event occurs the object will not react to that event. Events may be initiated by users, other objects, or code statements. Many objects are designed to respond to one or more events.

Navigating the Excel Object Hierarchy

The Excel object model is the set of objects that Excel exposes to the development environment. Many of the objects within the Excel object model are contained within other objects; that is, there is a hierarchical or parent-child relationship between the objects. The Application object represents the application itself; all other objects are below it and accessible through it. It is by referencing these objects in code that we are able to control Excel and create the desired functionality within our projects. Objects along with their properties and methods are referred to in code using the “dot” operator as illustrated in the example code below:

Application.ActiveWorkbook.SaveAs "Inventory.xlsx"

Parent Object

Child Object

Method of the Child Object

Argument for the Method

Some of the objects in Excel are considered global, meaning that they are at the top of the hierarchy and can be referenced directly. The Workbook object is a child object of the Excel Application object. Because the Workbook object is global, it is not necessary to specify the Application object when referring to it. Therefore, the following two statements are equivalent:

```
Application.ActiveWorkbook.SaveAs "Inventory.xlsx"  
ActiveWorkbook.SaveAs "Inventory.xlsx"
```

Several objects in the Excel object model actually represent a *collection* of objects. A collection is a set of objects, typically of the same type. The Workbooks collection in Excel represents the set of all open workbooks. An item in a collection is referenced using an index number or its name.



To access help for all Excel objects, open Help from within the Visual Basic Editor. From the navigation links on the left, click the *Excel for VBA reference* link and then click the *Object model* link. Use the navigation links to display the help content for a particular object.

Understanding Collections

A collection is a set of similar objects such as all open workbooks, all worksheets in a specified workbook, or all charts in a workbook.

Many Excel collections have the following properties:

Property	Description
Application	refers to the application that contains the collection.
Count	an integer value representing the number of items in the collection.
Item	refers to a specific member of the collection identified by its name or its position. In some collections, Item is a method rather than a property. Item is typically the default property or method of a collection.
Parent	refers to the object that contains the collection.

Some collections provide methods similar to the following:

Method	Description
Add	allows you to add an item to the collection.
Delete	allows you to remove an item from the collection by identifying it by name or position.

Referencing Objects in a Collection

A large part of programming in Excel is a matter of first identifying or referencing the desired object, then manipulating that object by changing its properties or using its methods. In order to reference a specific object, you may first have to identify the collection in which it is contained. You can reference an item in a collection by its position or by its unique name within the collection. When you reference an object by position, you provide an index number that identifies the position of the item within the collection.

The following syntax references an object within its collection by using its position. Since the Item property is the default property of a collection, its inclusion in the syntax is not required.

CollectionName(Object Index Number)

Workbooks.Item(1)

Workbooks(1)

Charts(intCount)

Understanding Collections, continued:

The following syntax references an object within its collection by using the object name. Here again, it is not necessary to include the Item property in the syntax.

CollectionName(ObjectName)

Workbooks("Inventory.xlsx")

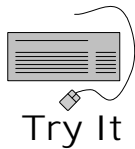
Worksheets("YTD Sales")

Charts("Sales by Month")



There may be times when you must navigate through the object hierarchy to reference an object that is two or three levels deep. Like any default property, if a collection is the default property of an object, it is not necessary to name it in the reference.

When referring to an item in a collection, the index number or item name can be a literal, variable, or other expression so long as it evaluates to the name of an item or its position within the collection.



Understanding Collections

For this exercise, we'll want to tile the Excel and the Visual Basic Editor windows so we can see them all at the same time. If you have other programs running, close them before doing this exercise. The **Practice** workbook we created in the last section should still be open.

1. Switch to Excel and open **Inventory.xlsm** and **Inventory(2).xlsx** from the student Data folder.
2. Switch to the Visual Basic Editor.
3. Close the Properties Window and the Project Explorer.
4. Right-click the taskbar and select *Show windows side by side*.
5. To make **Practice** the active workbook, click cell A1. Notice that a cell pointer is not visible in the other Excel windows.
6. In the Immediate window, type the following code and press **Enter** (↵):

```
Workbooks("Inventory.xlsm").Activate
```

7. Note the cell highlighting in the window for the Inventory workbook.
8. Enter the following lines of code into the Immediate window pressing **Enter** (↵) after each. Note the result of each statement in the Excel windows.

```
Worksheets(2).Activate  
Worksheets("Sheet3").Activate  
Workbooks("practice.xlsm").Activate  
Workbooks.Add  
Workbooks.Close
```

9. Maximize the Excel and the Visual Basic Editor windows.
-

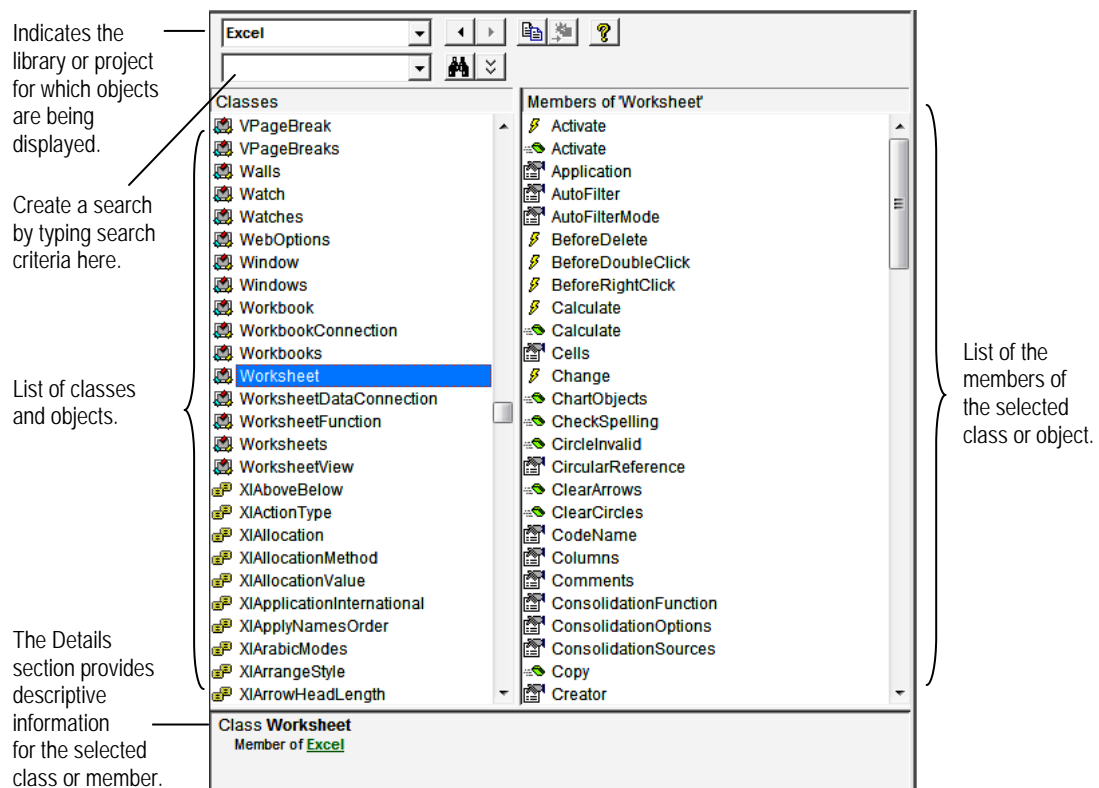
Using the Object Browser

You use the Object Browser to examine the hierarchy and contents of the various classes and modules defined in Excel and in your project. The same kind of information is available for other component libraries that may be referenced by your project, such as Visual Basic for Applications, Office, etc.

The Object Browser is often the best tool to use when you are in search of information about an object, helping you to answer questions such as:








- What is the name of that object, property, method, or event?
- Does an object have a certain property, method, or event?
- What arguments are required by a given method?
- Where does a certain object fit within an object hierarchy?

Below is an illustration of the Object Browser:



Using the Object Browser, continued:

The following icons and terms are used in the Object Browser:


	Class	indicates a class module.
	Property	is a variable or procedure representing an attribute of a class.
	Method	is a procedure of a class module accessible to other modules.
	Event	indicates an event which the class generates.
	Constant	is a variable with a permanent value assigned to it.
	Enum	is a set of constants.
	Module	is a standard module.

The Library drop-down list allows you to focus your search on a specific component, called a library, which contains class definitions and other information.

Using the Object Browser






STEPS

1. In the Visual Basic Editor, perform one of the following to open the Object Browser:
 - Open the View menu and select *Object Browser*.
 - Press **[F2]**.
 - On the Standard toolbar, click  Object Browser.
2. Select the desired library from the Project/Library drop-down list.
3. Select the class object from the *Classes* list.
4. Scroll through the *Members* list to display information about the object's properties, methods, and events.

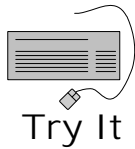


Right-click in the Members pane and select *Group Members* to display the members of an object grouped in the categories of properties, methods, events, etc.




Select an object or member in the Object Browser and press **[F1]** or click  Help to display the Visual Basic Help topic for the item selected.

To search in the Object Browser, type the desired search criteria in the Search text box and click . Click  to close the search pane.

A list of the global properties and methods can be displayed in the Object Browser by selecting *<globals>* from the Classes list.



Using the Object Browser

1. Open the **Practice** workbook.
2. Switch to the Visual Basic Editor. Maximize the editor, if necessary.
3. Click  Object Browser on the Standard toolbar in the Visual Basic Editor. (Adjust or move the windows on the desktop as needed to get a better view of it.)
4. Select *Excel* from the Library drop-down list.
5. Select *Application* in the Classes pane.
6. Right-click in the Members pane and check *Group Members* if the option is not active.
7. Scroll down the member list and select the *Goto* method. (Methods are preceded by the  icon.)
8. Press **[F1]** to open the Help window and briefly review the information. Close Help.
9. In the Object Browser, select the *Workbook* object from the Classes pane.
10. Use the space below to record the names of three events that the Workbook object can recognize: (Events are preceded by the  icon.)





11. Select *VBA* from the Project/Library drop-down list.
12. Select *<globals>* at the top of the Classes pane.
13. Scroll down and select the *Mid* method in the Members pane.
14. Use the information about the Mid method in the Details section, which is the gray pane at the bottom of the Object Browser window, to answer the following questions:
 - Is the Mid method a sub or a function procedure?

 - Name the *required* arguments for the Mid method?

15. Make sure *<globals>* is still selected in the Classes pane; select the *MsgBox* method in the Members pane.

continued on next page

Using the Object Browser, continued

16. View the information in the Details section. Click [VbMsgBoxStyle](#).
 17. Select one of the constants in the Members pane and click  Help.
 18. Close the Help page after reviewing the information.
 19. Select *Excel* from the Project/Library drop-down list.
 20. In the Object Browser, enter `activate` in the Search Text box; click  Search.
 21. Scroll through the list of search results.
 22. Name three objects that have the Activate method as a member:
 23. Make sure one of the Activate method results is selected and click  Help to view the information about the Activate method.
 24. When finished, close the Help page.
 25. Click  to close the Search Results pane.
 26. Select *Worksheet* in the Classes pane. Select the *Change* event in the Members pane.
 27. Press **[F1]** to display Visual Basic Help.
 28. Briefly review the information and close the Help page when finished.
 29. Close the Object Browser (if maximized, use the Close button on the right side of the menu bar).
-

Working with Properties

Most objects in Excel have an associated set of properties. During program execution, your code can read property values and in most cases, change them as well. To read a property value, you simply reference the object and the desired property within a statement.

The syntax to read an object's property follows:

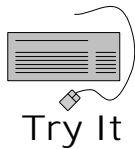
ObjectReference.PropertyName

ActiveSheet.Name

The syntax to change an object's property is as follows:

ObjectReference.PropertyName = expression

ActiveSheet.Name = "New Inventory"




Working with Properties

1. In the **Practice** workbook, enter each of the lines of code shown below into the Immediate window to return the value of different properties. Make sure to use the Auto List feature.

```
? Workbooks("practice.xlsm").FullName  
?  
? Application.Caption  
?  
? Application.OperatingSystem  
?  
? Application.StandardFont
```

2. Display the Code window for modPractice.
3. Create a new sub procedure named AddColumnFormatting.
4. Type the following code into the procedure:

```
Sub AddColumnFormatting()  
    Columns("A").Select  
    Selection.ColumnWidth = 12  
    Selection.Style = "Currency"  
End Sub
```

5. Click  Save. (We'll run the procedure in the next exercise.)
-

Using the With Statement

You can use the With statement to work with several properties or methods belonging to a single object without having to type the entire object reference on each line. The With statement can help optimize your code because every time Visual Basic must read a line with “dots” it slows down execution. So, the fewer “dots” on a line, the better.

The syntax for the With statement is shown below:

With ObjectName

<statements>

End With

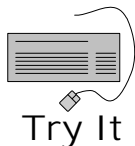
```
With ActiveWorkbook
    .PrintOut
    .Save
    .Close
End With
```



You can nest With statements if needed.



Make sure your code does not jump out of the With block before the End With statement is executed. This can lead to unexpected results.



Using the With Statement

1. Modify the AddColumnFormatting procedure to match what is shown below:

```
Sub AddColumnFormatting()
    Columns("A").Select
    With Selection
        .ColumnWidth = 12
        .Style = "Currency"
    End With
End Sub
```

2. Save the changes to modPractice.
3. Display the Excel window.
4. Make sure that Sheet1 of the **Practice** workbook is displayed.
5. Run the AddColumnFormatting procedure.
6. Note the formatting that has been applied to column A.
7. Close and save **Practice**.

Working with Methods

Many Excel objects provide a set of public sub and function procedures that are callable from outside of the object using a reference in your Visual Basic code. These procedures are called methods, an object-oriented term that describes the actions an object can perform. Some methods require arguments that must be supplied when using the method.

The syntax to invoke an object method is as follows:

ObjectReference.Method [arguments]

```
Workbooks.Open "Inventory.xlsx"
```

```
Range("A1", "B12").Select
```

```
Selection.Clear
```

Named versus Positional Arguments

When you call any procedure or method that has arguments, you have two choices of how to list the argument values to be sent. We've seen how you can pass values by positioning them in the same order as the argument list. You can also pass values by naming each argument along with the value you wish to pass. When you use named arguments it is not necessary to match the argument order or to insert commas as placeholders in the list for optional arguments.

A colon followed by an equal sign is the syntax for using named arguments:

ArgumentName:=value

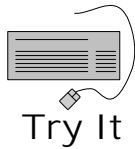
By way of example, consider the PrintOut method which has the arguments shown in the syntax below:

```
Sub PrintOut([From], [To], [Copies], [Preview], [ActivePrinter], [PrintToFile],  
[Collate], [PrToFileName], [IgnorePrintAreas])
```

The two statements below show both ways of passing values when calling the PrintOut method. The first example passes the values by position; the second by naming the arguments.


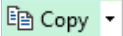

```
Workbooks("Inventory.xlsx").PrintOut 1, 2, 2, , , True
```

```
Workbooks("Inventory.xlsx").PrintOut From:=1, To:=2, Copies:=2, Collate:=True
```



Working with Methods

In this exercise we'll create two sub procedures that contain several calls to methods. We'll record the first one and create the second one in the Code window.

1. In the Excel window, open the **Inventory.xlsm** workbook.
2. Display the New Inventory sheet.
3. On the Developer tab, click .
4. Make sure that *This Workbook* is selected under Store Macro In.
5. Name the macro `GetReceivedVehicles` and click OK.
6. Open the **Received Vehicles** workbook from the student data location.
7. Make sure Sheet1 is the active sheet.
8. Click the Home tab. In the Editing group, click Find & Select and then choose *Go To Special*.
9. In the Go To Special dialog box, select *Current Region* and click OK. The range containing the data on Sheet1 is selected.
10. Click  in the Clipboard group.
11. Switch to the **Inventory** workbook.
12. Select cell A1 on the New Inventory worksheet. Display the Home tab and click  Paste.
13. Switch to the **Received Vehicles** workbook and close it.
14. Stop the macro recorder.
15. Display the Visual Basic Editor and open the module containing the new macro.
16. In the Properties window, change the name of the module containing the macro to `modNewInventory`.
17. Use the space below to name the methods in the `GetReceivedVehicles` procedure.


18. In `modNewInventory`, create a new sub procedure named `MoveCells`.

continued on next page

Working with Methods, continued

19. In the new procedure, enter the code below to select column B and then cut the selection.

```
Columns("B").Select  
Selection.Cut
```

20. On the next line in the procedure, enter the code that will select column G.
21. As the last line of code in the MoveCells procedure, call the Paste method of the ActiveSheet object to paste the data that was cut from column B to column G.
22. Click  Save.
23. Return to Excel and make sure that New Inventory is the active sheet.
24. Display the Macro dialog box and run the MoveCells procedure.
25. Select and then delete the contents of the New Inventory worksheet.
26. Return to the Visual Basic Editor when you are finished.
27. Display the AddFormatting procedure in modFirst.
28. Edit the procedure so that it will work in the New Inventory sheet by commenting or deleting the line of code that inserts a row:

```
Range("A1").Select  
' Selection.EntireRow.Insert
```

29. Display modNewInventory in the Code window and create a new procedure named GetNewInventory that will call each of these procedures:

```
Sub GetNewInventory()  
    GetReceivedVehicles  
    MoveCells  
    AddFormatting  
End Sub
```

30. Save your work.
31. Return to the New Inventory sheet in the Excel window.
32. Run GetNewInventory.
33. When finished, select and delete the contents of the worksheet.
34. Return to the Visual Basic Editor.
-

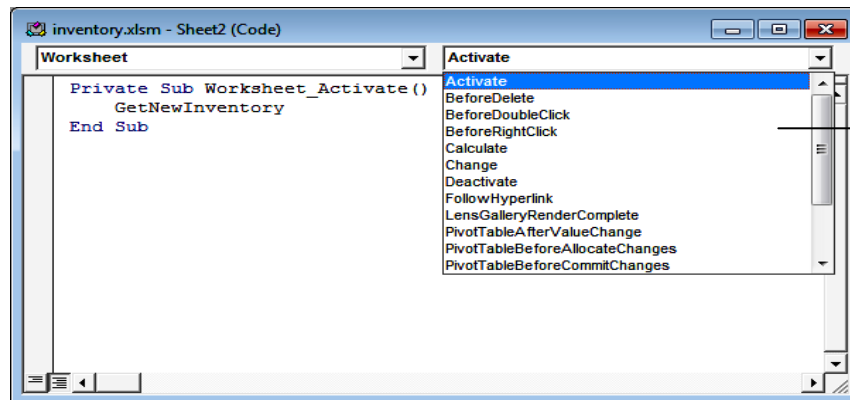
Creating an Event Procedure

An event procedure is a sub procedure that you create to run in response to an event associated with an object. Event procedure names are created automatically and consist of the object name, followed by an underscore and the event name. These names cannot be changed. Event procedures are stored in the class module associated with the object for which it is written.

The name syntax of the Activate event procedure of the Worksheet object is shown here:

```
Private Sub Worksheet_Activate()
```

When you select an object from the Object drop-down list in the Code window, a list of its corresponding events will become available from the Procedure drop-down list. The following is an illustration of the Code window showing an event procedure for the Activate event of the Worksheet Object.



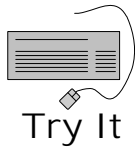
Procedure drop-down list shows all events for the selected object.

Creating an Event Procedure



STEPS


1. Display the code window for the appropriate class module.
2. Select the object from the Object drop-down list.
3. Select the event from the Procedure drop-down list.
4. Enter the desired code in the event procedure.



Working with Events

1. In the Visual Basic Editor, double-click Sheet 2(New Inventory) in the Project window to display its Code window.
2. Open the Object drop-down list at the top of the Code window and select *Worksheet*.
3. Select *Activate* from the Procedure drop-down list. (If desired, delete the structure that is inserted for the `Worksheet_SelectionChange` event.)
4. To have the `GetNewInventory` procedure run when the New Inventory sheet is activated, call the procedure from within the `Worksheet_Activate` procedure:

```
Private Sub Worksheet_Activate()  
    GetNewInventory  
End Sub
```

5. Click  Save.
 6. Display the Excel window.
 7. If the New Inventory sheet is displayed, display another worksheet and then return to the New Inventory sheet to activate it and run the `GetNewInventory` macro automatically.
 8. Delete the inserted data and return to the Visual Basic Editor.
-

4

Using Expressions, Variables, and Intrinsic Functions

Understanding expressions and statements

Declaring and setting variables

Understanding data types

Working with Variable Scope and Lifetime

Using intrinsic functions and constants

Creating and using message boxes

Creating and using input boxes

Understanding Expressions and Statements

The foundation of any programming language rests on its expressions and the statements that put those expressions to use.

Expressions

An expression is defined as any language element, alone or in combination, that represents a value. The following table lists the different expression types in Visual Basic:

String	evaluates to a sequence of characters.
Numeric	evaluates to anything that can be interpreted as a number.
Date	evaluates to a date.
Boolean	evaluates to True or False.
Object	evaluates to an object reference.

Expressions can be represented by any one or a combination of the following language elements:

Literal	is the actual value, explicitly stated.
Constant	represents a value that cannot change during program execution.
Variable	represents a value that can change during program execution.
Function/Method /Property	performs a procedure and represents the resulting value. This category includes user-defined functions.
Operator	allows for the combination of expression elements.

Understanding Expressions and Statements, continued:

The following table shows examples of expressions:

Element	String	Numeric	Date/Time	Boolean
Literal	"John"	1045	#05/05/2002#	TRUE
Constant	vbNullString	Pi	DateIncorporated	Editing
Variable	strCompanyName	lngTotal	datInvoiceDate	blnSelected
Function	Left(strCompany,4)	Sqr(9)	DateSerial(2002,5,5)	IsDate("05/32/02")
Method	fzmCustomers.GetName()	MyCollection.Count()	MyTask.GetOriginalDate()	MyTask.Updated()
Property	txtLastName.Text	Worksheets.Count	fzmReports.GetDateStart	txtLastName.Visible
Operators	&, LIKE	+, -, *, /, \, ^, MOD	+, -	AND, OR, NOT, ()

Understanding Expressions and Statements, continued:

Statements

A statement is a syntactically complete unit of execution that results in a definition, declaration, or action. An individual statement is generally entered one per line and may not span more than one line unless the line continuation character (_) is used.

Statements generally combine the language's keywords with expressions to control the flow of program execution and get things done in the application.

Examples of statements are shown below:

```
ActiveSheet.Name = "New Inventory"
```

```
strLabel = ActiveCell.Value
```

```
curPrice = curPrice * 1.1
```

Declaring Variables

A variable is a name that is used to represent a value. Variables are particularly useful in representing values likely to change within a procedure or an entire application. A variable name identifies a unique location in memory where a value may be stored temporarily. When a variable name is evaluated within a program, the value currently stored within the variable's unique memory location is returned.

Variables are created by a declaration statement. A variable declaration establishes a variable's name, data type, scope, and lifetime.

The general syntax of a variable declaration is:

Dim/Public/Private/Static VariableName [As <type>]

Dim strSsn As String

Private intCounter As Integer

Public datTodaysDate As Date

Naming Variables

When you declare a variable, you give it a name. Visual Basic will associate the name with a location in memory where the variable will be stored.

Variable names are subject to the following limitations:

- must start with a letter.
- may include letters, numbers, and underscore characters.
- may not exceed 255 characters in length.
- may not be a reserved word such as TRUE or FALSE.

Assigning Values to Variables

An *assignment* statement is used to set the value of a variable. The variable name is placed on the left side of the equal sign. The right side of the assignment statement can be any expression that evaluates to the appropriate data type:

VariableName = expression

intCounter = intCounter + 1

lngTotal = lngTotal + ActiveCell.Value

Declaring Variables, continued:

Implicit versus Explicit Variable Declaration

Another way of declaring variables is to simply assign a value to a valid variable name. This is known as *implicit* declaration, while the use of the Dim, Static, Private, and Public declaration statements results in *explicit* variable declarations.

Within the Excel development environment, you can take advantage of variable name syntax checking by requiring that all variables within a module be explicitly declared. This is accomplished by placing the **Option Explicit** statement in the Declarations section of each module in which you would like the check performed. Upon compiling or execution of the application, VBA will inform you of any references to undeclared variables. The resulting benefit is that you will be alerted to any code that stores a value to, or retrieves a value from, an undeclared variable.

Understanding Compiling

Compiling prepares your code for execution by translating your high-level source code into a more efficient language. Depending on the compiler, code is translated either into pseudo language (P-code) or machine language. The compiler maintains a list of variables and controls the usage of memory required by your program. Compiling your code flags variables that are not declared, as well as variables that you have declared but are misspelled in your code.

Declaring Variables Explicitly

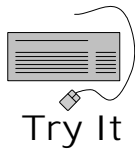


STEPS

1. To force explicit variable declaration within an existing module:
 - Open the desired form module or standard module;
 - Select (*General*) from the Object list box;
 - Select (*Declarations*) from the Procedure list box;
 - Type `Option Explicit` in the code window and press `Enter`.
2. To force explicit variable declaration for all new modules:
 - Open the Tools menu and select *Options*;
 - On the Editor tab, select *Require Variable Declaration* and click OK.



If the option has not been enabled, you must type `Option Explicit` in an existing module to force explicit variable declaration within that module.



Declaring Variables Explicitly

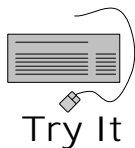
1. If necessary, open **Inventory.xlsm** from the Data folder.
 2. In the Visual Basic Editor, display the Code window for modNewInventory.
 3. Move to the top of the module.
 4. If the keywords “Option Explicit” do not appear, enter them on a line before the first procedure.
 5. Open the Tools menu and select *Options*.
 6. On the Editor tab, select *Require Variable Declaration*, if necessary.
 7. Click OK.
-




STEPS

Displaying the Compile Project Command Button

1. Open the View menu, point to *Toolbars*, and select *Customize*.
2. In the Customize dialog box, click the Commands tab.
3. Select *Debug* from the Categories list.
4. Drag Compile Project from the Commands list to the Standard toolbar.
5. Click Close.



Displaying the Compile Project Command Button

1. Use the steps in the procedure above to display the Compile Project button on the Standard toolbar of the Visual Basic Editor.
 2. Throughout the course, click  Compile VBAProject to compile your projects after you add code to them.
-

Understanding Data Types

When you declare a variable, you can specify its data type. Your choice of data type impacts your program's accuracy, memory usage, efficiency, and vulnerability to errors.

Data types determine the following:

- the structure or length of the memory storage unit that will hold the variable.
- the kind and range of values the variable can contain. For example, you can only store a specified range of integers to an Integer data type; you cannot store character strings or fractions to it.
- the operations that can be performed with the variable such as add, subtract, concatenate, etc.

If the data type is omitted from the variable declaration statement, a generic type called *Variant* is used as the default.

Numeric Data Types

Numeric data types are provided to allocate memory appropriate for storing and operating on numbers. Generally, you should select the smallest type that will hold the intended data to conserve memory and to speed execution. Numeric operations are performed according to the standard order of operator precedence with operations inside parentheses being performed first.

The following numeric operations, shown in order of precedence, can be used with numeric data types:

Exponentiation (^)	raises a number to the power of an exponent.
Negation (-)	indicates a negative operand.
Multiplication, Division (* /)	multiplies, divides with floating point result.
Integer Division (\)	divides rounded operands with integer result.
Modulus (Mod)	divides two numbers and returns the remainder.
Addition, Subtraction (+ -)	adds, subtracts operands.

String Data Types

The String data type is used to store one or more characters.

The following operators can be used with strings:

Concatenation (&)	combines two string operands. If an operand is numeric, it is first converted to a string-type Variant.
Like LikePattern	provides pattern matching of strings.

Understanding Data Types, continued:

VBA supports the following fundamental data types:

Type	Size/ Description	Prefix	Value Range
Boolean	2 byte	bln	True/False (-1 / 0)
Byte	1 byte integer	byt	0 to 255
Integer	2 byte integer	int	-32,768 to 32,767
Long	4 byte integer	lng	-2,147,483,648 to 2,147,483,647
Single	4 byte floating point	sng	approximate range -3.40×10^{38} to 3.40×10^{38}
Double	8 byte floating point	dbl	approximate range -1.798E308 to -4.941E-324 (neg) approximate range 4.941E-324 to 1.798E308 (pos)
Currency	8 byte fixed point	cur	-922,337,203,658,477.5808 to 922,337,203,658,477.5807
Fixed Length String	character string	str	0 to approximately 65,400 characters
Variable Length String	character string 10 bytes+	str	0 to approximately 2 billion characters
Date	8 byte floating point	dat	numbers that represent dates from: 1/1/100 to 12/31/9999
Variant	all numeric data types, datetime, variable length string, empty, null, error, array, objects and user-defined types 16 bytes – numeric 22 bytes + – string	var	
Decimal	12 byte number (only used within a Variant)	dec	number of digits to the right of the decimal point: 0-28
Object	4 bytes		an address reference to an object

Understanding Data Types, continued:



Use the prefix for the data type when naming variables so that you and others will know what kind of data is stored in your variables.

For monetary values with up to 4 decimal places, use the Currency data type. Single and Double data types are subject to small rounding errors.

A numeric variable of any type may be stored to a numeric variable of another type. The fractional part of a variable typed as Single or Double will be rounded off when stored to an Integer typed variable.



Excessive use of the Variant data type will make an application slow because of the memory Variants consume and the need for value and type checks.

VBA code that was written prior to the release of Office 2010 may cause errors when run in a 64-bit version of Office because of data type incompatibilities. For more information, find the topic entitled *64-Bit Visual Basic for Applications Overview* in the MSDN Library.

Working with Variable Scope

Declaring Variable Scope

The keyword used in the declaration statement—Dim, Static, Public, or Private—defines the scope of the variable. The scope of a variable determines which procedures and modules can reference the variable.

Procedure-level Variables

A Procedure-level variable is only visible within the procedure in which it is declared. You might think of Procedure-level variables as Local variables. Procedure-level variables are declared using the keyword Dim or Static within a procedure. Argument variables also have procedure-level scope.

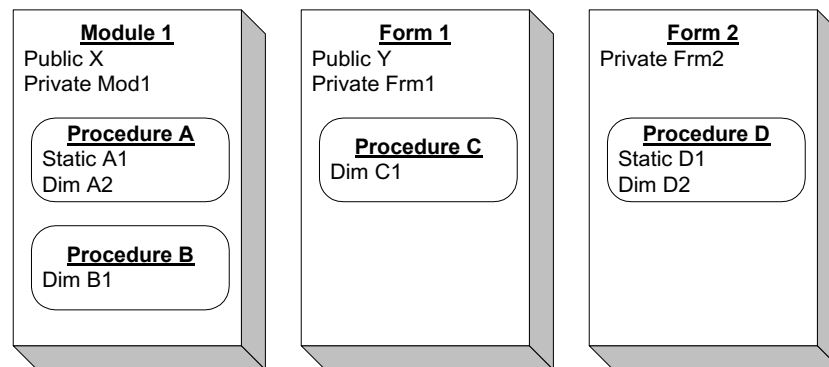
Module-level Variables

Module-level variables are declared with the Dim or Private keyword within the Declarations section of a module. Module-level variables can be referenced by any procedure contained in the module. The Private keyword can only be used in the Declarations section.

Public Variables

The Public keyword is used to declare a variable that may be referenced anywhere within the application. The Public keyword can only be used in the Declarations section of a module.

The diagram below demonstrates the accessibility of variables across procedures, modules, and forms based upon the scope of each variable:



Visibility of the variables to each of the procedures is outlined as follows:

Procedure A can see: A1, A2, Mod1, X, Y

Procedure B can see: B1, Mod1, X, Y

Procedure C can see: C1, Frm1, X, Y

Procedure D can see: D1, D2, Frm2, X, Y

Working with Variable Scope, continued:

The order of processing for variables is:

1. Local (Dim, Static)
2. Module-level (Private, Dim)
3. Public (Public)

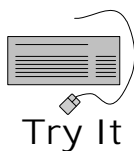
The same variable name may be used to declare separate variables within different scope levels. Whenever a variable is referenced, the most locally scoped variable will be evaluated.



Public variables declared within a Form module become properties of that form; they must be referenced by specifying the form name, followed by a period, followed by the variable name. The same applies to procedures in a form.

To avoid “Ambiguous name detected” errors, all public variables declared in standard modules should have unique names.

Avoid the excessive use of public variables declared in standard modules. It is quite easy for two or more sections of an application to get into a tug-of-war over the value of these variables. Public variables declared within a *class* module are somewhat safer as various parts of an application can declare and use their own objects. Passing values and variables from one procedure to another in the form of arguments is safer too. In general, use the most local scope as practical when declaring variables.




Declaring Variables

1. Display the Excel window and open the **Vin Numbers** workbook from the student Data folder.
2. Display the Visual Basic Editor.
3. In the Project window, select the **Vin Numbers.xlsx** project.
4. Open the Insert menu and select *Module*.
5. Create a new sub procedure named ParseVin.
6. Enter the code shown below to declare a procedure-level variable named strVinNum that has a data type of string:

```
Dim strVinNum as String
```

7. Using the names below, declare six more procedure-level string variables.

strYear
strMake
strModel
strColor
strCountry
strClassification

8. Click  Save.
 9. At the prompt, click No to open the Save As dialog box.
 10. Save **Vin Numbers** as a Macro-Enabled workbook.
-

Using Intrinsic Functions

An intrinsic function is similar to a function procedure in that it performs a specific task or calculation and returns a value. There are many intrinsic functions that can be used to manipulate text strings or dates, convert one data type to another, or perform mathematical calculations.

Type Conversion Functions

Listed below are some common type conversion functions and their results:

CCur(expr)	returns the currency equivalent of any valid expression.
CDBl(expr)	returns the double equivalent of any valid expression.
CDate(expr)	returns the date equivalent of any valid expression.
CInt(expr)	returns the integer equivalent of any valid expression.
CStr(expr)	returns the string equivalent of any valid expression.
CSng(expr)	returns the single equivalent of any valid expression.
CVar(expr)	returns the Variant equivalent of any valid expression.
CLng(expr)	returns the long equivalent of any valid expression.
Val(strexp)	returns the numeric equivalent of a character string.

Evaluation Functions

Several intrinsic functions used to provide information about an expression are described below:

IsNumeric(expr)	returns a Boolean value indicating whether <i>expr</i> can be interpreted as a numeric value.
IsNull(expr)	returns a Boolean value indicating whether <i>expr</i> is Null.
IsDate(expr)	returns a Boolean value indicating whether <i>expr</i> can be converted to a date value.
IsEmpty(expr)	returns a Boolean value indicating whether <i>expr</i> , a Variant variable, has been initialized.
IsMissing(argname)	returns a Boolean value indicating whether an optional Variant argument, <i>argname</i> , has been passed to a procedure.

Using Intrinsic Functions, continued:

String Functions

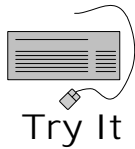
The following intrinsic functions are used to manipulate text strings:

FormatDateTime(DateExpr [, NamedFormat])	returns the expression with the specified date or time formatting.
Format(numexpr strexpr1, [, strexpr2])	returns a Variant containing the numeric or character value formatted using the format mask in <i>strexpr2</i> .
LCase(strexpr)	returns a Variant containing <i>strexpr</i> converted to all lower case characters.
UCase(strexpr)	returns a Variant containing <i>strexpr</i> converted to all upper case characters.
Trim(strexpr)	returns a Variant containing <i>strexpr</i> with all leading and trailing spaces removed.
LTrim(strexpr)	returns a Variant containing <i>strexpr</i> with all leading spaces removed.
RTrim(strexpr)	returns a Variant containing <i>strexpr</i> with all trailing spaces removed.
Left(strexpr, numexpr)	returns a Variant with the first <i>numexpr</i> characters from <i>strexpr</i> .
Mid(strexpr, numexpr1, [, numexpr2])	returns a Variant that contains characters <i>numexpr1</i> through <i>numexpr2</i> of <i>strexpr</i> . If <i>numexpr</i> is not included, the remainder of <i>strexpr</i> is returned.
Right(strexpr, [, numexpr])	returns a Variant with the last <i>numexpr</i> characters from <i>strexpr</i> .
InStr([numexpr], [, strexpr1, strexpr2])	returns the numeric starting position of <i>strexpr2</i> within <i>strexpr1</i> . <i>Numexpr</i> is an optional starting position for the search.
Replace(strexpr, find, [, replace[, start[, count[, compare]]]])	finds the substring, <i>find</i> , within <i>strexpr</i> and replaces it with the value in <i>replace</i> .
StrConv(strexpr, conversion [, LCID])	converts <i>strexpr</i> based on the value in <i>conversion</i> .



Intrinsic functions appear as methods in the Object Browser. To view them, select VBA from the Project/Library drop-down list and select <globals> in the Classes pane.

To find help on a particular function, click Microsoft Visual Basic for Applications Help on the Standard toolbar in the Visual Basic Editor. On the MSDN library website, click the *Office VBA language reference* link under Other Resources. In the navigation at the left, click *Visual Basic language reference* and then click the *Functions* link. Select the desired function from the list.



Using Intrinsic Functions

A Vehicle Information Number (VIN) is a sequence of seventeen character used to uniquely identify a motor vehicle. VINs encode information about a vehicle, such as its make, model, country of origin, and year of manufacture. In this exercise we are going to create a procedure that uses the Left, Right, and Mid intrinsic functions to extract the characters in a VIN which represent specific information, as shown in the table below. The completed code follows the exercise.

Note: The examples of VINs used in this manual are only loosely based on the actual VIN format. This data is being used to illustrate the use of intrinsic functions not to accurately identify vehicles.

The following characters are used in this exercise:

Character Position	Information Yielded
1 st	Country of Origin
2 nd	Make of the vehicle
3 rd and 4 th	Model name
11 th	Year of manufacture
12 th	Color
17 th	Classification

1. In the ParseVIN procedure, enter the following code to select cell A2 and set the value of strVinNum to the value in the cell:

```
Range("A2").Select
strVinNum = ActiveCell.Value
```

2. On the next line enter the following code to use the Mid function to return the character from the VIN representing the year and store it in the strYear variable:

```
strYear = Mid(strVinNum, 11, 1)
```

3. Using the information in the table above, create five more statements using the Left, Right, and Mid functions to extract vehicle information and store it to the appropriate variable.
4. Enter the following code to set up a With...End With structure and place the value of strYear in the cell next to the active cell using the Offset method:

```
With ActiveCell
    .Offset(, 1).Value = strYear
End With
```


continued on next page

Using Intrinsic Functions, continued

5. Enter five more similar statements within the **With** structure to place the values of the variables shown in the table below into the appropriate cells. Note that the offset is always figured from the active cell (A2).

strMake
StrModel
strColor
strCountry
strClassification

	A	B	C	D	E	F	G
1	Vin #	Year	Make	Model	Color	Country	Classification
2	1448331649w271961						
3							

6. Save your work.
7. Click  Compile VBAProject and then fix any errors that the compiler identifies.
8. Return to the Excel window and test the ParseVIN procedure. Compare your result with the illustration below.

	A	B	C	D	E	F	G
1	Vin #	Year	Make	Model	Color	Country	Classification
2	1448331649a271961	a	4	48	2	1	1
3							

9. Close the workbook when you are finished, saving the changes.
-

Exercise Code - Using Intrinsic Functions

Option Explicit

Sub ParseVin()

Dim strVinNum As String

Dim strYear As String

Dim strMake As String

Dim strModel As String

Dim strColor As String

Dim strCountry As String

Dim strClassification As String

Range("A2").Select

strVinNum = ActiveCell.Value

strYear = Mid(strVinNum, 11, 1)

strMake = Mid(strVinNum, 2, 1)

strModel = Mid(strVinNum, 3, 2)

strColor = Mid(strVinNum, 12, 1)

strCountry = Left(strVinNum, 1)

strClassification = Right(strVinNum, 1)

With ActiveCell

.Offset(, 1).Value = strYear

.Offset(, 2).Value = strMake

.Offset(, 3).Value = strModel

.Offset(, 4).Value = strColor

.Offset(, 5).Value = strCountry

.Offset(, 6).Value = strClassification

End With

End Sub

Understanding Constants

A constant is a special kind of variable that receives an initial data value that doesn't change during program execution. Constants can be used to assign a meaningful name to a value or expression. This is useful in situations where a value is hard to remember, appears over and over, or whose meaning is not obvious. The use of constants can make code more readable.

While it sounds contradictory, constants are a subset of variables and the word variable is used to refer to any named memory storage unit. Declaring a constant is similar to declaring a variable in that a name and a data type are specified. The value of the constant is also set in the declaration statement. Constants are privately scoped by default unless the `Public` keyword is included in the declaration statement to create a publicly scoped constant.

The syntax of a constant declaration is as follows:

```
[Public] [Private]Const ConstantName [<As type>] = <ConstantExpr>
```

```
Const Pi As Double = 3.14159265358979
```

```
Const Pi As Double = 22/7
```

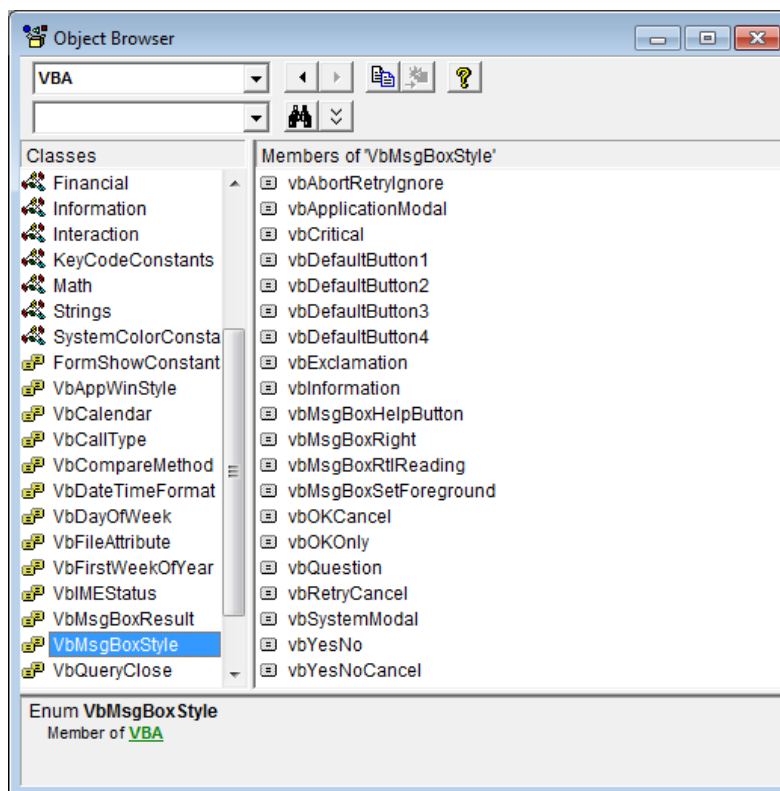
```
Public Const DQ As String = " " "
```

```
Public Const SQ As String = ""
```


Using Intrinsic Constants

Visual Basic for Applications has many built-in constants that can be used in expressions. VBA constants begin with the letters *vb* while constants belonging to the Excel object library begin with the letters *xl*. Intrinsic constants can be viewed in the Object Browser and also appear in the Auto List when appropriate.

Below is an illustration of the Object Browser showing the constants that can be used to define the Visual Basic Message Box style:



Some other useful Visual Basic constants are listed below:

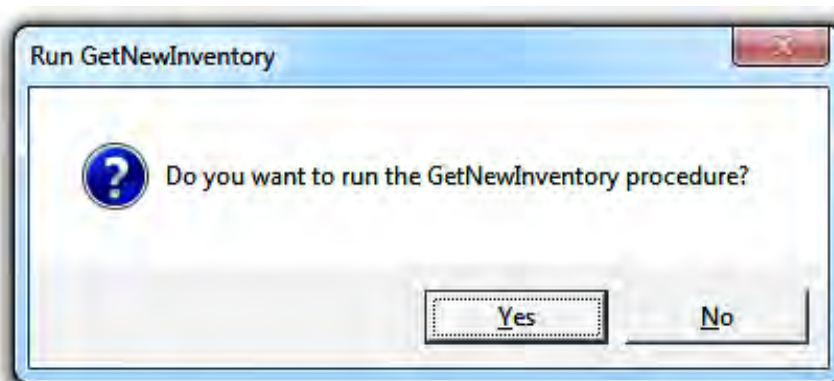
Constant	Equivalent to:	Same as pressing:
vbCr	Carriage return	Enter ↵
vbTab	Tab character	Tab ⇥
vbLf	Soft return or linefeed	⇧ Shift – Enter ↵
vbCrLf	Combination of carriage return and linefeed	
vbBack	Backspace character	← Bksp
vbNullString	Zero length string	""

Using Message Boxes

The MsgBox function can be used to display messages on the screen and prompt for a user's response.

The MsgBox function can display a variety of buttons, icons, and custom title bar text. In addition, the MsgBox function can be used to return a constant value that represents the button clicked by the user.

A sample of a message box with two buttons, a text message, an icon, and a title is shown below:



The MsgBox function syntax is:

```
variable=MsgBox(prompt[, buttons][, title][, helpfile][, context] )
```

```
intReturnVal= MsgBox("Do you want to run the GetNewInventory procedure?", _  
36, " Run GetNewInventory ")
```

```
intReturnVal= MsgBox("Do you want to run the GetNewInventory procedure?", _  
vbYesNo + vbQuestion, "Run GetNewInventory")
```

Using Message Boxes, continued:

The components of the MsgBox function are described below:





prompt	<p>is a string expression displayed as the message in the dialog box.</p> <p>The MsgBox message wraps automatically at the right edge of the box. To set line breaks, place a linefeed (vbLf) or a carriage return, (vbCr) within the message string expression.</p>
buttons	<p>is an optional numeric expression that defines the set of command buttons to display, the style of icon to use, the default button, and the modality of the message box. The desired configuration can be specified by entering a VB constant, a combination of constants, or the actual numeric value of the constant or sum of constants.</p>
title	<p>is an optional string expression that will appear in the title bar. If you omit the title, "Microsoft Excel" is the default title while running in the Excel development environment.</p>
helpfile	<p>an optional string expression that identifies the file name of the Help file to use for the message box.</p>
context	<p>an optional numeric expression that identifies the appropriate topic in the Help file related to the message box.</p>

Using Message Boxes, continued:

The values and constant names for creating buttons are explained below:

Value	Constant	Command Button
0	vbOKOnly	OK button only.
1	vbOKCancel	OK and Cancel buttons.
2	vbAbortRetryIgnore	Abort, Retry, and Ignore buttons.
3	vbYesNoCancel	Yes, No, and Cancel buttons.
4	vbYesNo	Yes and No buttons.
5	vbRetryCancel	Retry and Cancel buttons.
16384	vbMsgBoxHelpButton	displays a Help button in the dialog box.

The values for creating the icons are explained below:

Value	Constant	Icons
16	vbCritical	displays the Stop icon. 
32	vbQuestion	displays the Question icon. 
48	vbExclamation	displays the Exclamation icon. 
64	vbInformation	displays the Information icon. 

The values for setting the default command button are explained below:

Value	Constant	Default Button
0	vbDefaultButton1	sets the first button in the set as default.
256	vbDefaultButton2	sets the second button in the set as default.
512	vbDefaultButton3	sets the third button in the set as default.
768	vbDefaultButton4	sets the fourth button in the set as default.

Using Message Boxes, continued:

The values for controlling the modality of the message box are explained below:

Value	Constant	Modality
0	vbApplicationModal	suspends work in the current application until a response is made in the message box.
4096	vbSystemModal	suspends work in the current application until a response is made in the message box. In addition, the MsgBox window will be displayed as the top-most window on the system even if the user interacts with other applications.

To achieve a combination of settings, you can sum the desired values. For example, to display OK and Cancel buttons with the Stop icon and Cancel (the second button) set as the default, the type argument would be 273 (1 + 16 + 256). Although, it is preferable to sum the constants for the corresponding values rather than using the actual values themselves: vbOKCancel + vbCritical + vbDefaultButton2.

When adding numbers or combining constants for the Buttons argument, select only one value from each of the previously listed groups.

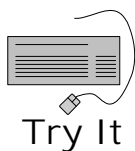
Return Value

The MsgBox function returns the value of the button that is clicked. You can reference this value by number or by its corresponding constant. The return values and corresponding constants are as follows:

Button Selected	Constant	Value Returned
OK	vbOK	1
Cancel	vbCancel	2
Abort	vbAbort	3
Retry	vbRetry	4
Ignore	vbIgnore	5
Yes	vbYes	6
No	vbNo	7

The return value is of no interest when the MsgBox only displays the OK button. In this case, it may be useful to call the MsgBox Function with the syntax used to call a sub procedure as shown in the following example:

```
MsgBox "You must enter a numeric value.", vbOKOnly, "Attention"
```



Using Message Boxes

1. If necessary, open the **Inventory** workbook and display the Visual Basic Editor.
2. Display the Code window for the New Inventory worksheet.
3. Modify the code as shown below to have a message box display before the macro runs when the worksheet is activated. (Remember to put a space before each underscore when continuing a statement on the next line.)

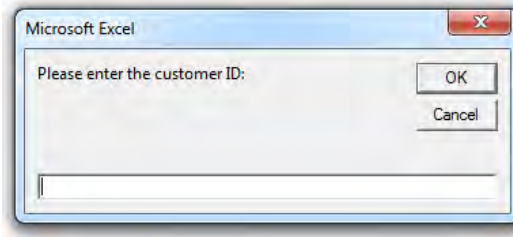
```
Private Sub Worksheet_Activate()  
    Dim intResponse As Integer  
    intResponse = MsgBox _  
        ("Do you want to run the GetNewInventory procedure?", _  
        vbYesNo, "Run GetNewInventory")  
    If intResponse = vbYes Then GetNewInventory  
End Sub
```

4. Save and compile the project.
 5. Display the Excel window.
 6. If the New Inventory sheet is already displayed, move to another sheet and then reactivate it.
 7. In the message box, click No.
 8. Return to the Visual Basic Editor.
-

Using Input Boxes

The Input Box function prompts the user for one piece of information and returns it as a string.

The following is an illustration of a dialog box displayed by the InputBox function:



The syntax of the InputBox function is shown below. In the example, the return value of the function is being stored in a variable named strCustomerID.

```
InputBox(prompt [, title] [, default][, xpos, ypos ][, helpfile, context])  
strCustomerID = InputBox("Please enter the customer ID:")
```

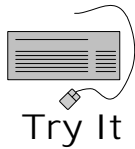
The arguments for the InputBox function are as follows:

prompt	string expression to display in the box. The maximum length is approximately 1024 characters.
title	string expression displayed in the title bar. If you omit the title, "Microsoft Excel" is the default title while running in the Excel development environment.
default	string expression displayed in the text box as the default response. If you omit default, the text box is empty.
xpos	numeric expression that specifies the horizontal distance, in twips, from the left edge of the screen to the left edge of the box. If you omit xpos, you must also omit ypos.
ypos	numeric expression that specifies, in twips, the vertical distance from the top of the screen to the top of the box.
helpfile	optional string naming the help file associated with the input box.
context	an optional numeric expression that identifies the appropriate topic in the Help file related to the message box.



If OK is clicked, the function returns the contents of the text box or a zero-length string if no value was entered. If the user clicks Cancel, the function returns a zero-length string ("").

A twip is equal to 1/20th of a point.



Using Input Boxes

1. In the Visual Basic Editor, display the GetReceivedVehicles procedure in modNewInventory.
2. Declare two procedure-level string variables named strFileLocation and strFile.
3. After the variable declarations, enter the following to set the strFileLocation variable to the path of the Inventory workbook. (For the sake of simplicity, we'll always assume that the Received Vehicles workbook is located in the same folder as the Inventory workbook.)

```
strFileLocation = ThisWorkbook.Path
```

4. On the next line, enter the following code to display an Input Box that prompts for the file name:

```
strFile = InputBox( _  
    Prompt:="What is the name of the new inventory file?", _  
    Title:="Inventory File", _  
    Default:="Received Vehicles.xlsx")
```

5. On the next line, enter the following code to concatenate the file location with the file name and then store the information in the strFile variable.

```
strFile = strFileLocation & "\" & strFile
```

6. If the existing code contains the following statement, select and then delete it.

```
ChDir "C:\data"
```

7. Modify the line of code that calls the Open method of the Workbooks object supplying the strFile variable for the argument instead of the actual file name.

```
Workbooks.Open Filename:=strFile
```

8. Modify the line of code that activates the Received Vehicles.xlsx workbook, supplying the proper Index value for the Workbooks collection.

```
Workbooks(2).Activate
```

9. Save and compile your work.
10. Switch to the Excel window and activate the New Inventory worksheet.
11. Click Yes in the message box to run the GetNewInventory procedure.
12. Click OK to accept the default text in the input box.
13. Clear the data from the worksheet; then close and save the **Inventory** workbook.

Declaring and Using Object Variables

In addition to storing string, numeric, date, and Boolean data to variables, you also use variables to reference objects in order to work with their properties, methods, and events. Any Excel object, such as a worksheet, range, or chart, can be represented and accessed using a variable name.

The following is the syntax to declare an object variable:

```
Dim/Public/Private/Static variablename [As <objecttype>]
```

```
Public wsSheet As Worksheet
```

```
Dim myRange As Range
```

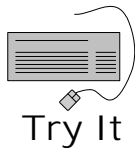
Assigning values to object variables is somewhat different from other variables and requires the use of the keyword **Set**:

```
Set VariableName = ObjectName
```

```
Set wsSheet = Worksheets("YTD Sales")
```

```
Set myRange = ActiveSheet.Range("A1", "F12")
```

Once you assign an object to an object variable, the object can be referenced by its variable name. Object variables can be and often are used to avoid typing lengthy object references.



Declaring and Using Object Variables

In the following example we will create a procedure that sums column I of a worksheet and places the total two rows below the last entry in the column. The Fiscal Year Sales workbook contains multiple sheets representing individual days. Since the number of sales for each day varies, we need to locate the last cell in the data range programmatically rather than manually selecting it. When finished, we'll step through the completed code one line at a time. In a later exercise, we will edit this procedure so that it adds totals for every sheet in the workbook.

1. Open the **Fiscal Year Sales** workbook.
2. Display the Visual Basic Editor and select the Fiscal Year Sales project in the Project Explorer.
3. Open the Insert menu and select *Module*.
4. Add the following procedure into Module1 of the Fiscal Year Sales project. (It is not necessary to break the line that sets the strTotalFormula variable.)

```
Sub AddTotals()  
    Dim LastCell As Range  
    Dim strTotalFormula As String  
    Set LastCell = Range("I2").End(xlDown)  
    LastCell.Select  
    ActiveCell.Offset(2).Select  
    strTotalFormula = _  
        "= sum(I2:" & LastCell.Address(False, False) & ")"  
    ActiveCell.Formula = strTotalFormula  
End Sub
```

5. Save and compile your work.
 6. Use the taskbar to tile the Visual Basic Editor and the Excel window side by side. Make sure that **Fiscal Year Sales** is the active workbook. If necessary, close the Project Explorer and the Properties window to see the Code window.
 7. In the Excel window, display the Macro dialog box. Select *AddTotals* and click Step Into.
 8. Press **[F8]** twice to begin to step into the code. The first line of code to execute will look for the last non-empty cell in Column I and store its range in the variable LastCell.
 9. Press **[F8]** four times observing what each line of code does as it executes.
 10. In the VB Editor, position the insertion point over the word "strTotalFormula" and note the value of the variable that appears in the pop-up window.
 11. Press **[F8]** again to finish executing the sub procedure.
 12. Save and close the workbook.
-

5

Controlling Program Execution

Understanding control-of-flow structures

Working with Boolean expressions

Using the If...Then...Else...End If structure

Using the Select Case...End Select structure

Using the For...Next structure

Using the For Each...Next structure

Using the Do...Loop structure

Guidelines for use of branching structures

Understanding Control-of-Flow Structures

When a procedure runs, the code executes from top to bottom in the order in which it appears. However, only the simplest of programs execute line after line of code in this manner. Most programs incorporate logic to control which lines of code to execute. Control-of-flow structures provide the logic used to branch, loop, and make decisions within code.

The following provides an overview of control-of-flow structures:

Sequential	Each line of code is executed in order from top to bottom.
Unconditional Branching	A statement directs the flow of program execution to another location in the program without condition. Calling a sub or function procedure or using the Goto statement are examples of unconditional branching.
Conditional Branching	The code to be executed is based on the outcome of a Boolean expression. Decision structures that are used to implement conditional branching include the If and Select Case structures.
Looping	A block of code is executed repeatedly as long as a certain condition exists. The For...Next and the Do...Loop are examples of looping structures.
Halt Statements	Commands are used to stop executing code. The Stop command suspends execution while retaining variables in memory. The End command terminates the application.

Working with Boolean Expressions

A *Boolean* expression is an expression that results in a value of either True or False. Also referred to as conditional expressions, they are used in control-of-flow structures to evaluate information and direct program control. Many Boolean expressions take the form of two expressions on either side of a comparison operator. If the result of the comparison is true, the condition is met and the structure passes control to the code to be executed.

The following are examples of Boolean expressions:

```
txtLastName = "Jones"  
YearExpired > 2010  
Sheets.Count < 3
```

The following comparison operators can be used in Boolean expressions:

=	Equal to
<>	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
Is	Compares object variables
Like	Compares string expressions

You can test for more than one condition by joining Boolean expressions with a Boolean or *logical operator*. When testing for more than two conditions, parentheses can be used around expressions joined by a logical operator to establish the combined logic and order in which expressions will be evaluated.

The following is a list and explanation of logical operators:

And	Each expression must evaluate to True for the condition to be true.
Or	One of the expressions must evaluate to True for the condition to be true.
Not	The expression must evaluate to False for the condition to be true.

Working with Boolean Expressions, continued:

The following are examples of multiple conditions joined by a logical operator:

```
CreditRating > 800 AND AnnualIncome > 30000
```

```
YearExpired = 2012 OR AmountDue > 0
```



A null expression will be treated as a false expression.

In Visual Basic, the Boolean data type is based upon the Integer data type. False is equivalent to 0 (zero); True is equivalent to -1. The value of all Boolean expressions is either 0 or -1. However, if a non-zero number is supplied in place of a Boolean expression, it will be treated as true.

Using the If...End If Decision Structures

The **If...End If** construct is used to execute one or more statements depending upon a test condition. If the condition tested is true, the statement or statements following the test will be executed; if the test result is false, execution resumes with the code immediately following the decision structure. There are four forms of the **If** construct. The first contains the condition and the statement to be executed in the same line:

If *<condition>* **Then** *<statement>*

If txtDept = "Sales" Then txtLevel = 5

The block form is used when several statements are to be executed based on the result of the test condition:

If *<condition>* **Then**
 <statement block>
End If

If txtCountry = "USA" Then
 txtAccount = "Domestic"
 curRate = 0.40
End If

Like the **If...Then** structure, the **If...Then...Else** structure passes control to the statement block following the **Then** keyword when the result of the condition is true; however, when the test condition is false, the statement block that follows the **Else** keyword will execute.

If *<condition>* **Then**
 <statement block>
Else
 <statement block>
End If

If txtCountry = "USA" Then
 txtAccount = "Domestic"
 curRate = 0.40
Else
 txtAccount = "Foreign"
 curRate = 1.2
End If

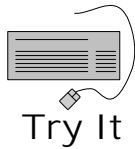
Using the If...End If Decision Structures, continued:

By modifying the basic structure and inserting **ElseIf** statements, you can create an **If...Then...Else** block that tests multiple conditions. During execution, the conditions are evaluated in the order that they appear until a condition is true or until each condition has been tested. If a true condition is found, the statement block following the condition is performed; execution then continues with the first line of code following the **End If** statement. If no conditions are true, execution will continue with the **End If** statement. An optional **Else** clause at the end of the block will catch the cases that do not meet any of conditions.

```
If <condition_1> Then  
    statementblock 1  
[ElseIf <condition_2>  
    [statement block 2]  
[ElseIf <condition_3>  
    [statement block 3]  
[Else <condition_n>  
    [statement block_n]  
End If  
  
If txtCountry = "USA" Then  
    txtAccount = "Domestic"  
    curRate = 0.40  
ElseIf txtCountry = "Canada" Then  
    txtAccount = "Domestic"  
    curRate = 0.65  
ElseIf txtCountry = "Mexico" Then  
    txtAccount = "Domestic"  
    curRate = 0.85  
Else  
    txtAccount = "Foreign"  
    curRate = 1.20  
End If
```



When creating the code for block structures, it is a good idea to outline the logic using the keywords first and enter the closing statement.




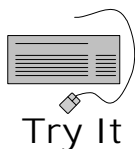
Working with the If...Then...End If Structure

In this exercise, we'll return to the code that we started building in the Inventory workbook. We'll add an **If...Then...End If** structure to the existing code in the `Worksheet_Activate` event procedure of the New Inventory worksheet that will test whether cell A1 contains data. If it does not contain data, we'll assume the sheet is blank and therefore ready to run the `GetNewInventory` procedure; control will be passed to the statement that displays the message box. Otherwise the sheet already contains information so it is not necessary to run `GetNewInventory` and the event procedure ends.

1. Open **Inventory.xlsm** from the Data folder.
2. Display the Visual Basic Editor.
3. Display the Code window for the New Inventory worksheet.
4. In the `Worksheet_Activate` event procedure, modify the code to match what is shown below:

```
Private Sub Worksheet_Activate()  
    Dim intResponse as Integer  
    If ActiveSheet.Range("A1").Value = "" Then  
        intResponse = MsgBox _  
            (Do you want to run the GetNewInventory procedure?", _  
            vbYesNo, "Run GetNewInventory")  
        If intResponse = vbYes Then  
            GetNewInventory  
        End If  
    End If  
End Sub
```

5. Click  Save.
 6. Display the Excel window.
 7. Activate the New Inventory sheet to test the procedure.
 8. Click No in the message box.
 9. Enter a value into cell A1.
 10. Reactivate the New Inventory worksheet and note that the message box does not appear.
 11. Delete the contents of cell A1.
 12. Return to the Visual Basic Editor.
-



Using the If...Then...Else...End If Structure

In this exercise, we'll create two functions that use the **If...Then...Else...End If** structure. The first function determines the markup on a vehicle by evaluating the character in the VIN which represents the model year of the vehicle. The necessary information is provided in the table below:

Character	Model Year	Markup
a	Previous year	10%
b	Current year	20%

The second function will evaluate the same character in the VIN to determine the model year of the vehicle. We'll modify the GetNewInventory procedure to call both of these functions and place their return values in the appropriate columns in the New Inventory worksheet.

1. In modNewInventory of the Inventory workbook, insert a new function named GetMSRP that will accept two arguments, DealerCost and VinNum.
2. Insert the following code to create an If...Then...Else structure that will evaluate the character in the VIN which represents the model year and then mark the cost up based on whether the car is the current or previous model year:

```
Function GetMSRP(DealerCost, VinNum)
    If Mid(VinNum, 11, 1) = "b" Then
        GetMSRP = DealerCost * 1.2 'vehicle is current model year
    Else
        GetMSRP = DealerCost * 1.1 'vehicle is previous model year
    End If
End Function
```

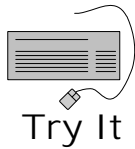
3. Display the GetNewInventory procedure. Using the information below, declare two procedure-level variables within the procedure:

Variable Name	Data Type
sngDealerCost	Single
strVinNumber	String

4. Below the last line of code in the procedure, enter a line of code to select cell A2.
5. On the next line, enter code to set the value of strVinNumber to the value in the active cell (A2).
6. On the next line, enter the following code to assign the Dealer Cost value in cell G2 to the variable sngDealerCost:

```
sngDealerCost = ActiveCell.Offset( , 6).Value
```

continued on next page



Using the If...Then...Else...End If Structure

7. To call the GetMSRP function and put its return value in cell H2, enter the following code on the next line in the procedure:

```
ActiveCell.Offset(, 7).Value = GetMSRP(sngDealerCost, strVinNumber)
```

8. Save your work.
9. Display the Excel Window and test the GetNewInventory procedure by activating the New Inventory worksheet and clicking Yes in the Message box.
10. Note that the column containing the MSRP amount isn't wide enough because the AddFormatting procedure was called before an amount was entered into the column.
11. Delete the contents of the New Inventory worksheet.
12. Return to the Visual Basic Editor.
13. In the GetNewInventory procedure, move the statement that calls the AddFormatting procedure to a line just before the End Sub statement.
14. In modNewInventory, create a new function procedure named GetYear that accepts VinNum as an argument.
15. Declare a procedure-level string variable in GetYear named strYearMarker.
16. On the next line in the procedure, use the Mid function to set the value of strYearMarker to the eleventh character of the VinNum argument.
17. Next, use the following information to create an If...Then...Else structure that evaluates strYearMarker and sets the value of the function accordingly.

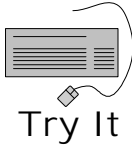
If strYearMarker is	Value of GetYear is
a	2016
b	2017

18. In the GetNewInventory procedure, locate the line of code that calls the GetMSRP function. Immediately before it, enter a similar line of code to call the GetYear function and place its return value in cell B2.
19. Save your work.
20. Return to the Excel window and test the procedure. Note the width of the column containing the MSRP amount.
21. Delete the contents of the New Inventory worksheet when you are finished.
22. Return to the Visual Basic Editor.

Exercise Code – Using the If...Then...Else...End If Structure

```
Sub GetNewInventory()  
  
    Dim sngDealerCost As Single  
    Dim strVinNumber As String  
  
    GetReceivedVehicles  
    MoveCells  
    Range("A2").Select  
    strVinNumber = ActiveCell.Value  
    sngDealerCost = ActiveCell.Offset(, 6).Value  
    ActiveCell.Offset(, 1).Value = GetYear(strVinNumber)  
    ActiveCell.Offset(, 7).Value = GetMsrp(sngDealerCost, strVinNumber)  
    AddFormatting  
End Sub
```

```
Function GetYear(VinNum)  
  
    Dim strYearMarker As String  
  
    strYearMarker = Mid(VinNum, 11, 1)  
    If strYearMarker = "a" Then  
        GetYear = "2016"  
    Else  
        GetYear = "2017"  
    End If  
End Function
```



Using the If...Then...ElseIf...End If Structure with ElseIf

In this exercise, we'll use an **If...Then...ElseIf...End If** structure in a function to evaluate the character representing the vehicle class. We'll call the new function from the GetNewInventory procedure and place its return value in the appropriate column in the New Inventory worksheet.

1. In modNewInventory, create a new function procedure named GetClassification that accepts an argument named VinNum.
2. Declare a procedure-level string variable named strClassMarker.
3. As the first line of code within the procedure, enter a code statement that will set the value of strClassMarker to the last character in VinNum. (Hint: use the Right function.)
4. Enter the following code to begin an **If...Then...ElseIf** structure that will evaluate the strClassMarker variable and assign the appropriate classification to the value of the function:

```
If strClassMarker = 1 Then
    GetClassification = "Car"
ElseIf strClassMarker = 2 Then
    GetClassification = "Truck"
End If
```

5. Using the information below, insert two more ElseIf statements before the End If statement to complete the structure:

Classification Marker	Vehicle Type
3	Van/Minivan
4	SUV

6. Display the GetNewInventory procedure in the Code window and modify the existing code to match what is shown below:

```
Range("a2").Select
strVinNumber = ActiveCell.Value
sngDealerCost = .Offset(, 6).Value
With ActiveCell
    .Offset(, 1).Value = GetYear(strVinNumber)
    .Offset(, 4).Value = GetClassification(strVinNumber)
    .Offset(, 7).Value = GetMsrp(sngDealerCost, strVinNumber)
End With
```

7. Save your work.
8. Return to the Excel window and activate the New Inventory worksheet to test the code. Delete the contents of the worksheet and return to the Visual Basic Editor when you are finished.

Using the Select Case...End Select Structure

The **Select Case** statement is often used in place of complex **If** statements. The advantage of using this construct rather than an **If...Then...Else** block with several **ElseIf** statements is that your code will be more readable and efficient. However, it is only useful when all tests are to be compared against just one value.

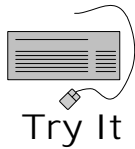
The **Select Case** structure contains the test expression in the first line of the block. Each **Case** statement in the **Select Case** structure is then compared against the test expression, and if a match is found, the statements accompanying the **Case** are executed. Only the statement block of the first matching **Case** expression is executed. A **Case Else** statement may be added at the end of the structure to add code that will execute if none of the **Case** expressions evaluate to True.

The syntax of the **Select Case** structure, followed by two examples, is shown below:

```
Select Case <TestExpression>
  Case <ExpressionList1>
    <StatementBlock1>
  Case <ExpressionList2>
    <StatementBlock2>
  Case <expressionListN>
    <StatementBlockN>
  [Case Else]
    <StatementBlock>
End Select
```

```
Select Case txtCountry
  Case "USA"
    txtAccount = "Domestic"
    curRate = 0.40
  Case "Canada"
    txtAccount = "Domestic"
    curRate = 0.65
  Case "Mexico"
    txtAccount = "Domestic"
    curRate = 0.85
  Case Else
    txtAccount = "Foreign"
    curRate = 1.2
End Select
```

```
Select Case txtScore
  Case 0 to 50
    txtResult = "Below Average"
  Case Is > 50
    txtResult = "Above Average"
  Case Else
    txtResult = "Irregular Score"
End Select
```



Using the Select Case...End Select Structure

In this exercise, we'll use the **Select Case...End Select** in a function that returns the color of the vehicle based on a character in the VIN. We'll create a similar structure to determine the model. We'll call both functions from the GetNewInventory procedure.

1. In modNewInventory, create a new function named GetColor that accepts an argument named VinNum.
2. Within the function procedure, declare a procedure-level string variable named strColorMarker.
3. Use the Mid function to set the value of strColorMarker to the twelfth character of VinNum.
4. Enter the following Select Case code to interpret the Color Marker and set the value of the function.

```
Select Case strColorMarker
  Case 0
    GetColor = "Black"
  Case 1
    GetColor = "White"
  Case 2
    GetColor = "Silver"
  Case 3
    GetColor = "Green"
  Case 4
    GetColor = "Blue"
  Case 5
    GetColor = "Red"
  Case 6
    GetColor = "Yellow"
  Case Else
    GetColor = "Unknown"
End Select
```

5. Save your work.
6. In the GetNewInventory procedure on the line after the code that calls the GetClassification procedure, insert a line of code that will put the return value of the GetColor function into cell F2.

continued on next page

Using the Select Case...End Select Structure, continued

7. Still within modNewInventory, use the information below to create another function called GetMake that:
 - Accepts an argument named VinNum;
 - Has a procedure-level string variable named strMakeMarker;
 - Sets the value of strMakeMarker to the second character in VinNum;
 - Contains a Select Case structure that evaluates strMakeMarker and sets the value of the function to the make name according to the following table:

Make Marker	Make Name
1	Buick
2	Cadillac
3	Chevrolet
4	Oldsmobile
5	Pontiac
6	GMC

8. Save your work.
9. Insert a line of code into the GetNewInventory procedure that will call the GetMake function and place its return value into cell C2.
10. To insert a function procedure that will extract the model from the VIN, place the insertion point at the end of modNewInventory and do the following:
 - Open the Insert menu and select *File*;
 - Navigate to the Data\InsertText folder; select **GetModel.txt** and click Open.
11. Briefly review the GetModel function.
12. Modify the GetNewInventory procedure to call the GetModel function and place its return value in cell D2.
13. Save and compile your work.
14. Test the GetNewInventory procedure. Compare your results with the illustration below.
15. Delete the information from the New Inventory worksheet when you are finished.

	A	B	C	D	E	F	G	H
1	VIN	Year	Make	Model	Classification	Color	DealerCost	MSRP
2	1556132350b086551	2017	Pontiac	Grand Prix	Car	Black	\$ 21,789.00	\$26,146.80
3	1216566410b452461						\$ 25,452.00	
4	1216542761b627911						\$ 31,960.00	
5	1559279965a240523						\$ 17,639.00	
6	1660501871a330824						\$ 23,682.00	
7	1335350326a341692						\$ 10,907.00	
8	1220881327b547474						\$ 32,913.00	
9	1553112010a210901						\$ 20,857.00	
10	1114752242a146471						\$ 10,285.00	
11	1557424187a541501						\$ 18,959.00	

Using the Do...Loop Structure

The **Do...Loop** structure controls the repetitive execution of code based upon the test of a conditional value. There are two variations of the **Do...Loop** structure: **Do While** and **Do Until**. The **Do While** structure will execute a code block as long as a given condition is true. The **Do Until** structure will execute a code block up to the point when a given condition becomes true or as long as the condition is false. The condition is any conditional expression that can be evaluated to true or false.

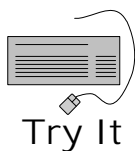
The **Exit Do** is optional and may be included to quit the **Do** statements and resume execution with the statement following the Loop. Multiple Exit Do statements can be placed anywhere within the Loop construct.

Use one of the following constructs to perform the statement block zero or more times:

```
Do While <condition>  
    <statement block>  
[Exit Do]  
Loop  
  
Do Until <condition>  
    <statement block>  
[Exit Do]  
Loop  
  
Do While ActiveCell.Value <> ""  
    ActiveCell.Value = ActiveCell.Value * 1.5  
    ActiveCell.Offset(1).Select  
Loop
```

To perform the statement block at least once, use one of the following:

```
Do  
    <statement block>  
[Exit Do]  
Loop While <condition>  
  
Do  
    <statement block>  
[Exit Do]  
Loop Until <condition>  
  
Do  
    intCount = intCount + 1  
Loop Until MsgBox("Continue?", vbYesNo) = vbNo
```



Using the Do...Loop Structure

So far in this chapter, we've created functions that will extract specific information from the VIN located in cell A2. In this exercise, we'll modify the GetNewInventory procedure to include a **Do...Loop** structure that will perform our set of functions on each of the VINs on the New Inventory sheet.

1. Edit GetNewInventory so that it matches the code shown below:

```
Sub GetNewInventory()
    Dim sngDealerCost As Single
    Dim strVinNumber As String

    GetReceivedVehicles
    MoveCells
    Range("a2").Select
    strVinNumber = ActiveCell.Value
    Do While strVinNumber <> ""
        sngDealerCost = ActiveCell.Offset(, 6).Value
        With ActiveCell
            .Offset(, 1).Value = GetYear(strVinNumber)
            .Offset(, 2).Value = GetMake(strVinNumber)
            .Offset(, 3).Value = GetModel(strVinNumber)
            .Offset(, 4).Value = GetClassification(strVinNumber)
            .Offset(, 7).Value = GetMSRP(sngDealerCost, strVinNumber)
            .Offset(1).Select
            strVinNumber = ActiveCell.Value
        End With
    Loop
    AddFormatting
End Sub
```

2. Save the project.
3. Return to Excel and test the procedure. When you are finished, leave the data in the worksheet and return to the Visual Basic Editor.

	A	B	C	D	E	F	G	H
1	VIN	Year	Make	Model	Classification	Color	DealerCost	MSRP
2	1556132350b086551	2017	Pontiac	Grand Prix	Car	Black	\$ 21,789.00	\$26,146.80
3	1216566410b452461	2017	Cadillac	Catera	Car	Blue	\$ 25,452.00	\$30,542.40
4	1216542761b627911	2017	Cadillac	Catera	Car	Yellow	\$ 31,960.00	\$38,352.00
5	1559279965a240523	2016	Pontiac	Trans Sport	Van/Minivan	Silver	\$ 17,639.00	\$19,402.90
6	1660501871a330824	2016	GMC	Envoy	SUV	Green	\$ 23,682.00	\$26,050.20
7	1335350326a341692	2016	Chevrolet	3500 Pickup	Truck	Green	\$ 10,907.00	\$11,997.70
8	1220881327b547474	2017	Cadillac	Escalade	SUV	Red	\$ 32,913.00	\$39,495.60
9	1553112010a210901	2016	Pontiac	Bonneville	Car	Silver	\$ 20,857.00	\$22,942.70
10	1114752242a146471	2016	Buick	Regal	Car	White	\$ 10,285.00	\$11,313.50
11	1557424187a541501	2016	Pontiac	Sun Fire	Car	Red	\$ 18,959.00	\$20,854.90

Finishing the GetNewInventory Procedure

1. In the Code window, place the insertion point on a line at the end of `modNewInventory`.
2. Open the Insert menu and select *File*.
3. Navigate to the student directory and select **AppendInventory.txt**; click Open.
4. Review the code in the `AppendInventory` procedure. This code moves the data from the New Inventory sheet to the Inventory sheet leaving the headings.
5. Tile the Excel window and the Visual Basic Editor side by side.
6. Place the insertion point anywhere within the `AppendInventory` procedure and press `F8` to step into the code. Continue to click `F8` and click Yes when the message box displays.
7. Step through the code and watch the results in the Excel window.
8. After the procedure has run, delete the appended inventory from the Inventory worksheet. (Delete rows 106 – 115.)
9. Maximize the Visual Basic Editor and create a new sub procedure named `ClearNewInventorySheet` at the end of `modNewInventory`.
10. Enter the following code to clear the remaining contents of the New Inventory sheet once the `AppendInventory` procedure has run:

```
Sub ClearNewInventorySheet()  
    Worksheets("New Inventory").Select  
    Cells.Select  
    Selection.Clear  
    Range("A1").Select  
End Sub
```

11. Return to Excel and maximize the window, if necessary.
12. Run the `ClearNewInventorySheet` procedure.
13. Return to the Visual Basic Editor.
14. Add the following line of code before the `End Sub` line in `GetNewInventory`:

```
AppendInventory
```

15. Display the `AppendInventory` procedure.
16. In the **If** statement block, just before the code that saves the workbook, call the `ClearNewInventorySheet` procedure.
17. Save your work; then test the `GetNewInventory` procedure in the Excel window.
18. Close the workbook when you are finished, saving the changes.

Using the For...To...Next Structure

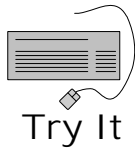
The **For...To...Next** structure executes a block of statements a specific number of times using a counter that increases or decreases in value.

Beginning with the start value, the counter is increased or decreased by the increment. The default increment is 1. You can count backwards by specifying a step increment of -1.

The **Exit For** statement is optional and may be included to quit the **For** construct and resume execution with the statement following **Next**.

Below is the syntax of the **For...To...Next** statement:

```
For <counter> = <start> To <end> [Step <increment>]  
    <statements>  
    [Exit For]  
Next [<counter>]  
  
Dim intIndex as Integer  
For intIndex = 1 to Worksheets.Count  
    Worksheets(intIndex).Select  
    Range("A1").Select  
    Selection.CurrentRegion.Select  
    Selection.Columns.AutoFit  
Next intIndex
```



Using the For...To...Next Structure

In an earlier exercise we created a sub procedure called `AddTotals` which placed the total sales for the day at the bottom of a sheet in the Fiscal Year Sales workbook. In this exercise, we'll use the For...Next structure in a sub procedure that automatically totals all of the worksheets in the workbook by repeatedly calling the `AddTotals` procedure.

1. Open the workbook **Fiscal Year Sales**.
2. Type the following sub procedure into `Module1` of the Fiscal Year Sales project:

```
Sub AllTotals()
    Dim i as Integer
    For i = 1 to Worksheets.Count
        Worksheets(i).Select
        AddTotals
    Next i
End Sub
```

3. Compile and save the project.
4. Switch to the Excel window and run `AllTotals`.
5. At some point during execution you will encounter a run-time error. On sheets when there were no sales, the procedure `AddTotals` doesn't work correctly.
6. Click End.
7. Close the workbook without saving changes and open it again.
8. Add the unshaded code below to the `AddTotals` sub procedure:

```
Sub AddTotals()
    Dim LastCell As Range
    Dim strTotalFormula As String
    If Range("I2").Value <> 0 Then
        Set LastCell = Range("I2").End(xlDown)
        LastCell.Select
        ActiveCell.Offset(2, 0).Select
        strTotalFormula = "= sum(I2:" & LastCell.Address & ")"
        ActiveCell.Formula = strTotalFormula
    Else
        Range("A3").value = "No Sales Today"
    End If
End Sub
```

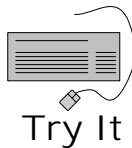
9. Save the workbook. We'll test the revised `AddTotals` procedure by calling it in the next exercise.

Using the For Each...Next Structure

The **For Each...Next** structure is used primarily to loop through a collection of objects. With each loop, it stores a reference to a given object within the collection to a variable. This variable can be used by your code to access that object's properties and methods. By default, it will loop through all of the objects in a collection. The **Exit For** statement may be included to quit the **For Each** construct and resume execution with the statement following the **Next** statement.

The syntax is shown here:

```
For Each <element> In <CollectionReference>  
    <statement block>  
    [Exit For]  
    <statement block>  
Next <element>  
  
For Each ws In Worksheets  
    ws.Select  
    Cells.Select  
    Selection.Clear  
Next ws
```



Using the For Each...Next Structure

In this exercise, we'll create a new AllTotals procedure that accomplishes the same result as the last exercise by using the For Each...Next structure to loop through each of the sheets in the Worksheets collection.

1. Comment out the AllTotals procedure.
2. Create another sub procedure named AllTotals.
3. Enter the following code into the new AllTotals procedure:

```
Sub AllTotals()  
Dim wsSheet As Worksheet  
For Each wsSheet In Worksheets  
    wsSheet.Select  
    AddTotals  
Next wsSheet  
End Sub
```

4. Save your work and then switch to Excel and run the AllTotals procedure.
5. View the 4-Jul-16 worksheet.
6. Save the workbook.

Guidelines for use of Control-of-Flow Structures

Use the following as a guide for choosing the appropriate Decision structure:

Need to...	Use...
Execute one statement based on the result of one condition.	If...Then or If...Then...End If
Execute a block of statements based on the result of one condition.	If...Then...End If
Execute 1 of 2 statement blocks based on the result of one condition.	If...Then...Else...End If
Execute 1 of 2 or more statement blocks based on 2 or more conditions with all conditions evaluated against 1 expression.	Select Case...End Select
Execute 1 of 2 or more statement blocks based on 2 or more conditions with conditions evaluated against 2 or more expressions.	If...Then...ElseIf...End If

Use the following as a guide for choosing the appropriate Looping structure:

Need to...	Use...
Repeat a statement block a specific number of times. The number is known or can be calculated once at the beginning of the loop and will not change.	For...To...Next
Repeat a statement block for each element in a collection or array.	For...Each or For...To...Next
Repeat a statement block while iterating through a list when the number of list items is not known or is likely to change.	Do...Loop
Repeat a statement block while a condition is met.	Do...Loop

6

Working with Forms and Controls

Understanding UserForms

Using the toolbox

Working with UserForm properties, events, and methods

Understanding controls

Setting control properties in the Properties window

Working with controls

Setting the tab order

Adding code to controls

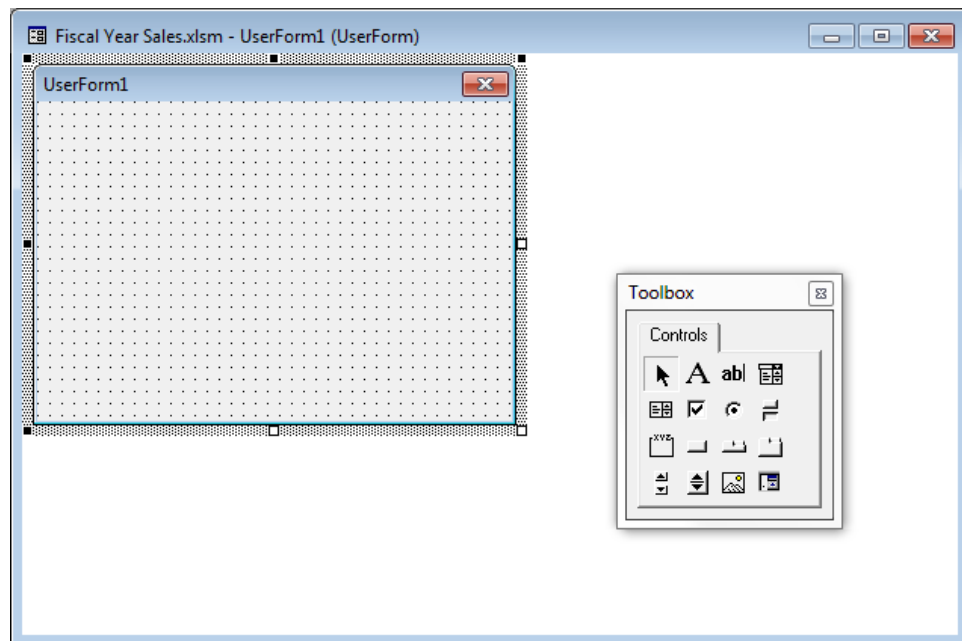
Understanding UserForms

Dialog boxes are commonly used in applications to interface with the user. With VBA, you can create your own custom dialog boxes that can be used to display information to or retrieve information from the person using your project. These custom dialog boxes are referred to in VBA as UserForms or simply Forms. A UserForm not only provides the visual interface, but also stores some or all of the code required for the functionality of the form.

Typically, a UserForm serves as a container for control objects, such as labels, command buttons, combo boxes, and so on. The controls that you use will depend on what kind of functionality you want in the form and what type of information needs to be communicated to and from the user.





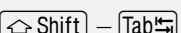


When you add a UserForm to your project, the UserForm window will appear with a blank form, as well as a toolbox containing the available controls. Controls are added by dragging icons from the toolbox onto the UserForm. The currently selected control on the form appears in a rectangle with eight handles. The handles can be used for sizing the control. The grid of dots on the form background can help you more easily place and align controls. There is an option for snapping controls to this grid; or you can choose to place controls freely on the form.

The illustration below shows a blank UserForm and the Toolbox:



Understanding UserForms, continued:


The following is a list of useful keystrokes and key combinations that can be used in the UserForm window:

	displays the code window for the selected form.
	displays the Properties window for the selected object.
	deletes the selected control.
	moves through the controls on the form in tab order.
	moves through the controls in reverse tab order.
	selects multiple controls.
	removes a selected control from a set of selected controls.


Adding a UserForm to a Project

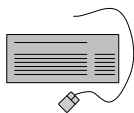


STEPS

1. In the Visual Basic Editor, select the desired project name in the Project Explorer.
2. To insert a form, perform one of the following:
 - Open the Insert menu and select *UserForm*.
 - Right-click the project name, point to *Insert* and then choose *UserForm*.
 - On the Standard toolbar, click the down arrow next to  Insert *Object* and select *UserForm*.



To display an existing form, select the form name in the Project Explorer and click  View Object; you can also right-click the UserForm object and select *View Object* from the Shortcut menu.



Try It














Adding a UserForm to a Project

1. Open **Fiscal Year Sales**, if necessary.
2. Display the Visual Basic Editor.
3. Select the *Fiscal Year Sales* project in the Project window, if necessary.
4. Open the Insert menu and select *UserForm*.

Using the Toolbox

The working set of controls that you can add to your forms is displayed in the toolbox along with the Select Objects tool used to select them. Controls are added to forms to build the desired interface and add functionality. When you are working with the toolbox and you select another window in the Visual Basic Editor, the toolbox becomes hidden; select the form and the toolbox will display again.

The default set of controls in the toolbox are shown and described below:

	Select Objects	makes the mouse behave as a pointer for selecting a control on the form.
	Label	creates a box for static text.
	Text Box	creates a box for text input and display.
	Combo Box	creates a combination of a drop-down list and text box. The user can choose an option or type the choice.
	List Box	creates a scrollable list of choices.
	Check Box	creates a logical check box.
	Option Button	creates an option button that allows exclusive choice from a set of options.
	Toggle Button	creates a toggle button that when selected indicates a <i>yes</i> , <i>true</i> , or <i>on</i> status.
	Frame	creates a visual or functional border.
	Command Button	creates a standard command button.
	Tab Strip	creates a collection of tabs which can be used to display different sets of similar information.
	MultiPage	creates a collection of pages. Unlike the Tab Strip, each page can have a unique layout.
	Scroll Bar	creates a tool that can set or return a value for a different control according to the positioning of the scroll box on the scroll bar.
	Spin Button	creates a tool that increments numbers.
	Image	creates an area to display a graphical image from a bitmap, icon, or metafile on your form.
	RefEdit	displays the address of a range of cells selected on one or more worksheets.



After clicking an icon in the toolbox and drawing it on the form, the toolbox selection will revert back to the Select Objects tool. If you double-click a toolbox icon, the icon will remain selected, allowing you to draw multiple controls.

Working with UserForm Properties, Events, and Methods

Every UserForm in a project has its own set of properties, events, and methods. You can set properties in both the Properties window and through code in the Code window.

Properties

All forms share the same basic set of properties. So, to begin with, every form will be the same. As you alter the form visually in the UserForm window, keep in mind that you are also changing its properties. For example, if you resize a form window by dragging its borders, you are changing the Height and Width properties.

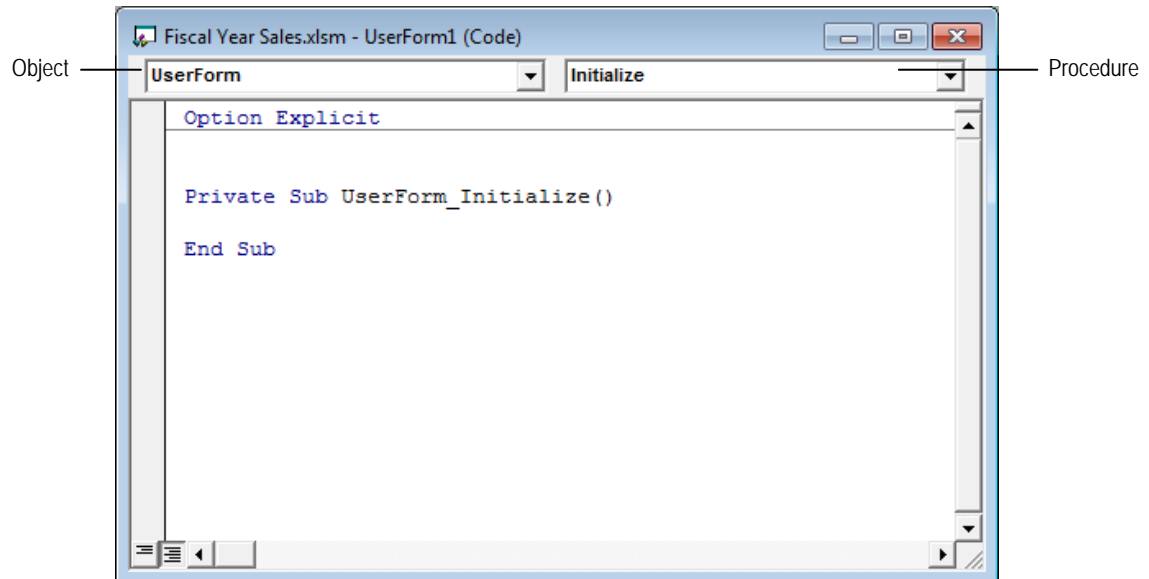
The following list describes some of the more commonly used properties of a UserForm:

Property	Description
BackColor	sets the background color of a form.
BorderStyle	sets the border style for the form.
Caption	sets the form's title in the title bar.
Enabled	determines whether the form can respond to user-generated events.
Height	sets the height of the form.
HelpContextID	associates a context-sensitive Help topic with the form.
MousePointer	sets the shape of the mouse pointer when the mouse is positioned over the form.
Picture	specifies graphic to display in the form by naming a bitmap, icon, or metafile.
StartPosition	sets the area of the screen where the form will be displayed.
Width	sets the width of the form.

Working with Form Properties, Events, and Methods, continued:

Events

All UserForms share a set of events that they can recognize and to which they can respond by executing a procedure. You create code to execute for a form event in the same way you create other event procedures: display the Code window for the form, select the UserForm object, and then select the event from the Procedure list.



Methods

UserForms also share similar methods that can be used to execute built-in procedures. Methods are normally used to perform an action on the form.

The three most useful methods of a UserForm for our purposes in this course are explained below:

Show	displays the form; can be used to load a form if it is not already loaded.
Hide	hides the form without unloading it from memory.
Unload	removes the form from memory.

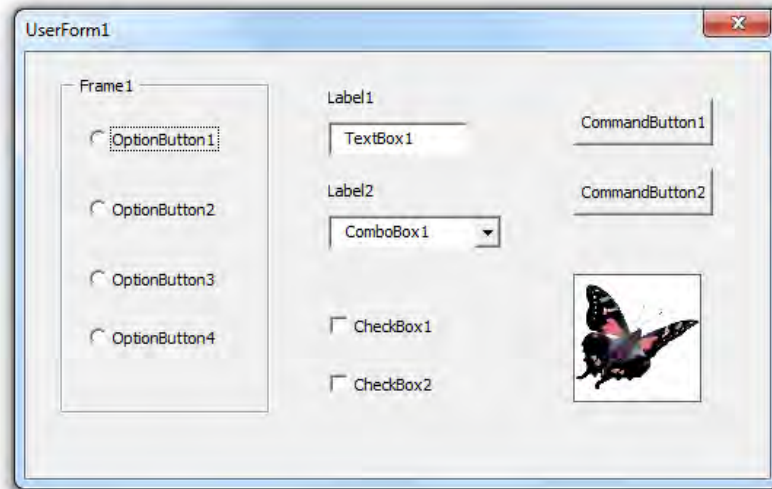


The keyword **Me** is used to reference the active form. Use **Me** in a UserForm's code module instead of its name to refer to the form and access its properties and methods.

Understanding Controls


A control is an object that is placed on a form to enable user interaction. Some controls accept user input and some display output. Like other objects, controls are defined by their properties, methods, and events.

A sample window with commonly used controls is shown below:



Although each type of control is unique, many share similar attributes and behaviors. Most control properties can be viewed and assigned manually via the Properties window at design time as well as referenced and assigned in code during execution. Before we look at specific controls, we will first discuss their shared characteristics.

The following list contains properties that are common among several controls:

Property	Description
ControlTipText	specifies a string to be displayed when the mouse pointer is paused over the control.
Enabled	determines if the user can access the control.
Font	sets the control text typeface and size.
Height	sets the height of the control.
MousePointer	sets the shape of the mouse pointer when the mouse is positioned over an object.
TabIndex	determines the position of the control within the tab order of the controls on the form.
TabStop	determines whether a control can be accessed via the  key.
Visible	determines whether the object is visible.
Width	sets the width of the control.

Understanding Controls, continued:

All controls have a *default* property that can be referred to by simply referencing the name of the control. For instance, the Caption property is the default property of the Label control making the two statements below equivalent:

```
Label1 = "First Name:"
Label1.Caption = "First Name:"
```

Many controls can respond to the same system events. As is the case with forms, if you want to write a procedure that executes when a control detects a particular event, you write the code in the Code window, having selected the control's name from the Object drop-down list.

The following is an explanation of a few of the most common events that many controls can detect and react to:

Click	occurs when the user clicks the mouse button on a control while the pointer is on it.
GotFocus	occurs when a control receives the focus.
LostFocus	occurs when a control loses the focus.
MouseMove	occurs when a user moves the mouse pointer over a control.

Naming Conventions for Controls

You should follow naming conventions when creating code to aid in the design and debugging stages. It is also valuable during later project maintenance when you or someone else has to look at your work and figure out what is going on.

A good practice is to assign a control a name that describes the particular use of the control with a prefix that identifies the type of control.

Below is a list of several control object name prefix conventions:

Object	Prefix
Check box	chk
Combo box	cbo
Command button	cmd
Frame	fra
Image	img
Label	lbl
List box	lst
Option button	opt
Text box	txt

Setting Control Properties in the Properties Window

Each type of control has a set of properties that describes the attributes of that control. These properties can be set in the design environment using the Properties window.


The control name and type are listed in the drop-down list at the top of the properties window. Select a desired object from this list to view its properties. Each individual property may present an edit box or a drop-down list for setting values.

The categories for the property list vary by object. Frequently used categories include appearance, behavior, font, position, and miscellaneous.



STEPS

Setting Control Properties in the Properties Window

1. Display the Properties window using one of the following methods:
 - Open the View menu and select *Properties Window*.
 - Press **F4**.
 - Click  Properties Window.
2. To display the properties in alphabetical order:
 - Click the Alphabetic tab.
3. To display the properties by category:
 - Click the Categorized tab.
4. To change a property setting:
 - Select the desired control in the UserForm window or from the drop-down list in the Properties window;
 - Scroll to the desired property and use the appropriate method to change the setting in the value column.



Try It


Using the Properties Window

1. If necessary, display the Properties window.
2. Make sure the new form is selected and set its properties as shown below:

Name	frmRunReports
Caption	Run Reports
Height	200
Width	330

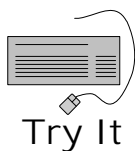
Working with the Label Control

The Label control is used to display text on a form. Label text cannot be modified directly by the user; however, it can be modified within a procedure by assigning a new value to its Caption property.

The Label control is represented in the toolbox by the  icon.

Below are some important and unique properties related to a Label control:



Property	Description
TextAlign	determines the alignment of the text inside the label.
AutoSize	determines if the dimensions of the label will be automatically sized to fit its caption.
Caption	sets the displayed text for the label.
WordWrap	determines if a label expands horizontally or vertically as text is added. Used in conjunction with the AutoSize property.



Working with the Label Control

After this exercise your form should look similar to the illustration below. As you perform this and other exercises in this chapter, do not worry about the spacing or alignment of items that you add to the form.

The screenshot shows a form window titled "Run Reports" with a close button in the top right corner. The form area has a light blue grid background. In the bottom left corner of the grid, there are two labels: "Start Date:" and "End Date:", one above the other.


1. On the Standard toolbar, click  to open the Toolbox, if necessary.
2. Click  Label in the Controls toolbox.
3. Click in the bottom left corner of the form.
4. In the Properties Window, set the following properties:

AutoSize	True
Caption	End Date:
WordWrap	False

5. Create the Start Date Label and position it as shown above. Use the caption Start Date: and change the other properties as you did with End Date.

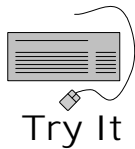
Working with the Text Box Control

A Text Box control allows the user to add or edit text. Both string and numeric values may be stored in the Text property of the control.

The Text Box control is represented in the toolbox by the  icon.

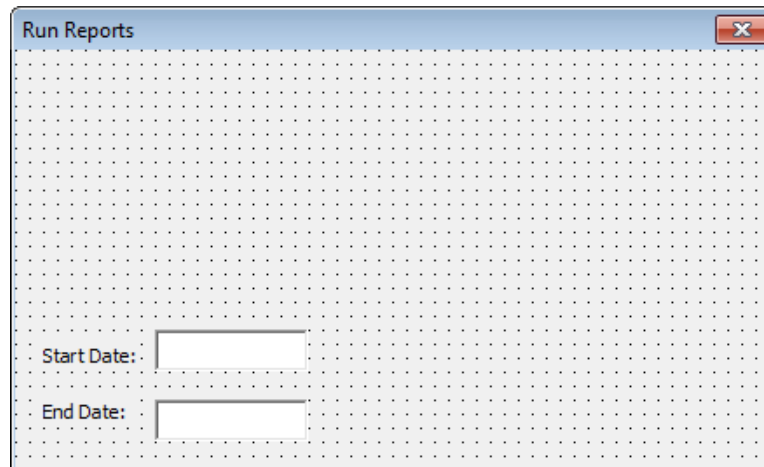
Below are some important properties related to a text box:


Property	Description
MaxLength	specifies the maximum number of characters that can be typed into a text box. The default is 0 which indicates no limit.
MultiLine	indicates whether a text box can contain more than one line.
ScrollBars	determines whether a multi-line text box has horizontal and/or vertical scroll bars.
Text	contains the string displayed in the text box.



Working with the Text Box Control

After this exercise, your form should look similar to the illustration below:



1. Click  TextBox.
2. Click to the right of the Start Date label to create a default sized control.
3. Set the following properties to the values displayed below:

Name	txtStartDate
Enabled	False
Width	50

4. Select the text box.
 5. Make a copy of the text box and position it directly below the original as shown in the above illustration.
 6. Change the Name property of the new text box to `txtEndDate`. Note that the other property changes were carried forward to the new instance of the text box.
-

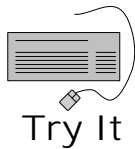
Working with the Command Button Control

Command buttons are used to get feedback from the user. In an event-driven environment, command buttons are among the most important controls for initiating event procedures. As a result, the most heavily used event associated with the Command Button control is the Click event.

The Command Button control is represented in the toolbox by the  icon.


Below are two unique properties related to a Command button control:

Property	Description
Cancel	allows the [Esc] key to “click” a command button. This property can only be set for one command button per form.
Default	allows the [Enter↵] key to “click” a command button. This property can only be set for one command button per form.



Working with the Command Button Control

After this exercise, your form should look similar to the illustration below:

1. Click  CommandButton.
2. Click to create a command button in the upper right hand corner of the form.
3. Set the following properties for the control:

Name	cmdDisplay
Caption	Display
Default	True


4. Create another command button below the first as shown.
5. Set the following properties for the control:

Name	cmdCancel
Caption	Cancel
Cancel	True
Default	False

6. Save the workbook.

Working with the Combo Box Control

A Combo Box control allows you to display a list of items in a drop-down list box. The user can select a choice from the list or type an entry. The items appearing in the list are added in code using the control's `AddItem` method.

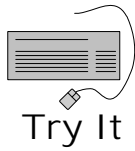
The Combo Box control is represented in the toolbox by the  icon.

Below are some important combo box control properties:

Property	Description
ListRows	sets the number of rows that will display in the list.
MatchRequired	determines whether the user can enter a value that is not on the list.
Text	returns or sets the text of the selected row in the list.


A few important methods that belong to the Combo Box control are explained below:

AddItem <i>item_name, index</i>	adds the specified item to the bottom of the list. If you specify an index number after the named item, you add the item to that position in the list.
RemoveItem <i>index</i>	removes the item referred to by its index number.
Clear	clears the entire list.



Using the Combo Box

After this exercise, your form should look similar to the illustration below:

1. Click  ComboBox and create the control in the lower right corner of the form.
2. Set the following properties:

Name	cboMonth
ListRows	12
MatchRequired	True

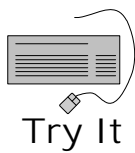
3. Create a label to the left of the combo box.
4. Set the following properties for the label:

Name	lblMonth
Caption	For Which Month?

Working with the Frame Control

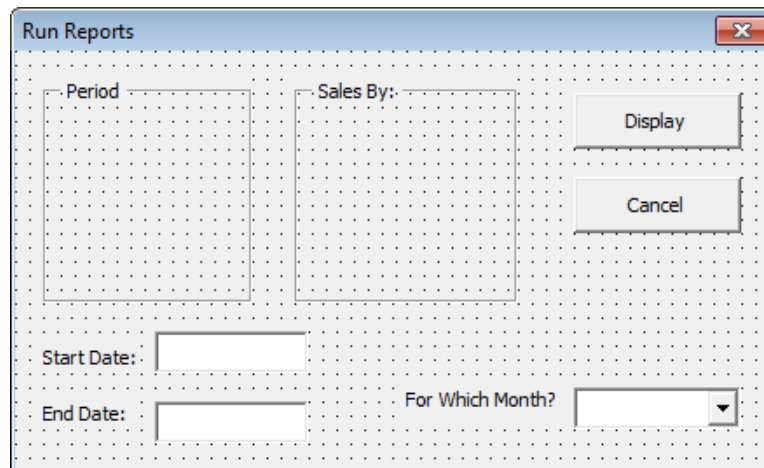
Use a Frame control to group a set of controls either functionally or logically within an area of a UserForm. When option button controls are placed within a frame, they are related logically; that is, setting the value of one affects the values of the others in the group. Option buttons in a frame are mutually exclusive, meaning that when one is set to True the others will be set to False. When you want to display a group of controls together because their content is related, a frame can be used to group the controls functionally or conceptually. In that case, changing the value of one control typically has no effect on the value of the others.


The Frame control is represented in the toolbox by the  icon.



Working with the Frame Control

After this exercise, your form should look similar to the illustration below:



1. Click  Frame and draw a Frame control in the upper left-hand corner of the form.
2. If necessary, manually resize the frame so that it looks like the illustration above.
3. Set the following properties for the frame:

Name	fraPeriod
Caption	Period


4. Add a second frame to the right of the Period frame.
5. Set the following properties for the frame:

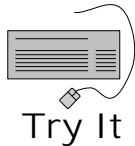
Name	fraSalesBy
Caption	Sales By:

6. If necessary, resize the frame so it appears similar to the one in the illustration.

Working with Option Button Controls

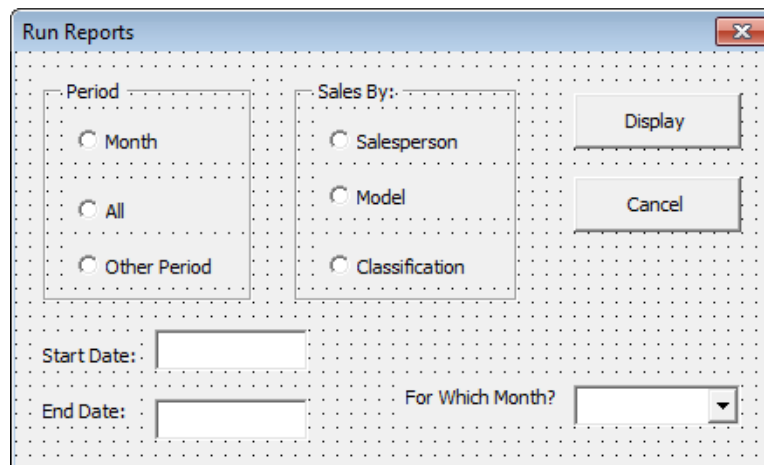
An option button control displays a button that can be set to on or off. Option buttons are typically presented within a group in which only one button may be selected at a time. The Value property of the option button control indicates the on or off state of the button.


The option button control is represented in the toolbox by the  icon.



Working with Option Button Controls

We will now add option buttons to the frame control. After this exercise, your form should look like the illustration below:



1. Click , move the mouse pointer within the Period frame control and click the left mouse button.
2. Create two more option buttons within the Period frame.
3. Create three option buttons in the Sales By frame.
4. Set the names and captions for the option buttons as follows:

Name	Caption
optMonth	Month
optAll	All
optOther	Other Period
optSalesperson	Salesperson
optModel	Model
optClassification	Classification

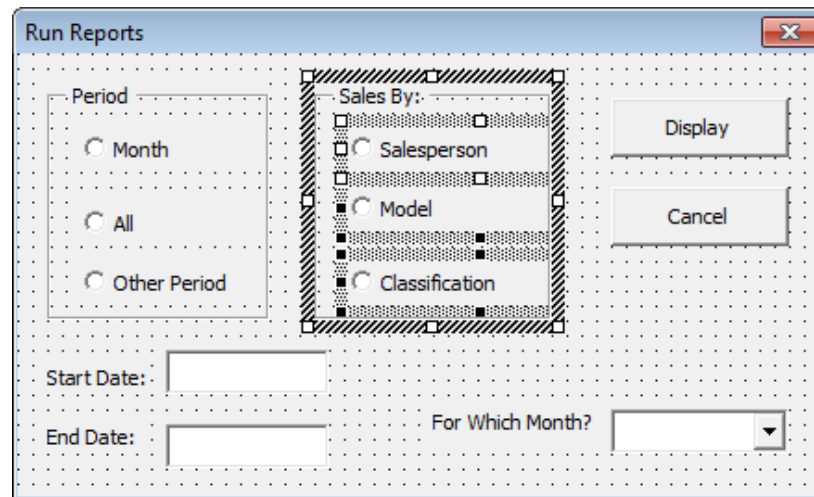
5. Drag one of the frame controls to see how the option buttons move with the frame. Drag the frame back into its original position when finished.
6. Save the project.

Working with Control Appearance

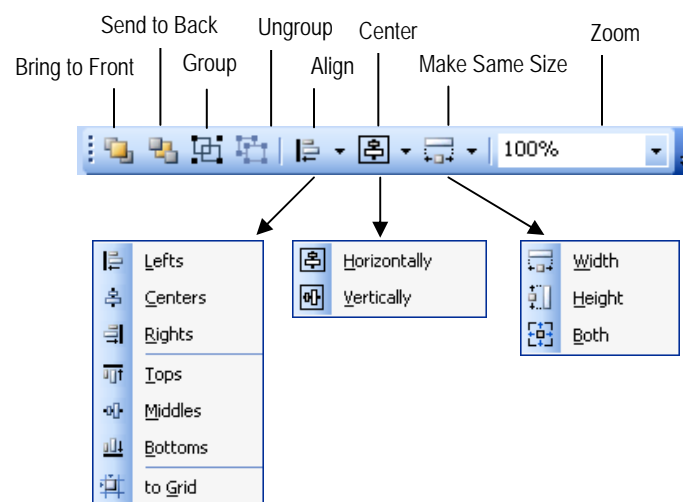
The UserForm toolbar provides several tools that can be used to manipulate the appearance of the controls on your form. The same functions provided by the UserForm toolbar can also be accessed from the Format Menu. These functions include grouping, aligning, sizing, and positioning controls.

To use many of the tools on the UserForm toolbar, it is necessary to first select multiple controls. To select multiple controls, click the first control then hold down the **Shift** key and click any additional controls. Controls will be aligned or sized according to the first control selected. You will be able to tell which control was selected first because its selection handles will be white.

The following is an illustration of a UserForm with multiple controls selected:








The following is an illustration of the UserForm toolbar followed by the options associated with the Align, Center, and Make Same Size tools respectively:



Working with Control Appearance



STEPS

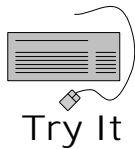
1. Display the UserForm toolbar, if necessary.
2. Select the desired controls.
3. To align the selected controls:
 - Click the down arrow of the  Align tool; make the desired selection.
4. To group the selected controls:
 - Click  Group on the Form Design toolbar.
5. To ungroup the selected controls:
 - Click  Ungroup on the Form Design toolbar.
6. To make controls the same height, width, or size:
 - Click the down arrow of the  Make Same Size tool; make the desired selection.
7. To center controls on the form:
 - Click the down arrow of the  Center tool; make the desired selection.



The Center tool can be used to center controls within a frame.

You can select multiple controls by dragging a bounding box around them.

The Format menu contains options for horizontal and vertical spacing not found on the UserForm toolbar.



Working with Control Appearance

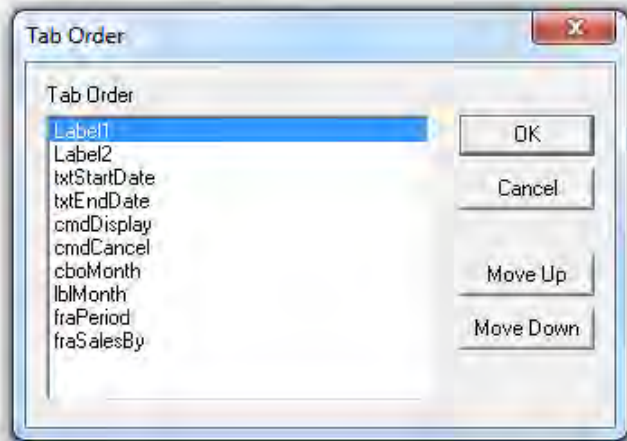
The finished form should look like the one shown below:

1. Select the *Month* option button in the Period frame.
2. Hold down and select the remaining option buttons in the frame.
3. Open the Format menu, point to *Align*, and choose *Lefts*.
4. Open the Format menu and point to *Vertical Spacing*; choose *Remove*.
5. Open the Format menu and select *Size to Fit*.
6. Repeat this process for the options in the Sales By frame.
7. Select the Period frame.
8. Hold down and select the Sales By frame.
9. Open the Format menu, point to *Align* and choose *Tops*.
10. Size the two frames simultaneously to eliminate the blank space at the bottom of the frames, if necessary.
11. Right align the Start Date and End Date labels.
12. Select the Start Date label and its corresponding text box.
13. Open the Format menu, point to *Horizontal Spacing*, and choose *Remove*.
14. Repeat this process for the End Date label and text box and for the Month combo box and its label.
15. Left align the Display and Cancel command buttons.

Setting the Tab Order

The tab order is the order in which pressing the tab key moves the focus from control to control on the UserForm. While you are building the form, the tab order is determined by the order in which you place the controls on the form. If the controls are subsequently rearranged, you may need to manually reset the tab order.

The following is an illustration of the Tab Order dialog box:



Setting the Tab Order



STEPS

1. View the desired form in the UserForm window.
2. Open the View menu.
3. Choose *Tab Order*.
4. Select the desired control from the list.
5. To move the control up in the list:
 - Click Move Up.
6. To move the control down in the list:
 - Click Move Down.

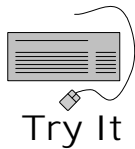


You can select multiple controls from the list in the Tab Order dialog box before moving them up or down. To do so, click the first item to be moved, hold down **Ctrl** and click additional items.


Labels, although listed in the Tab Order dialog box, are not included in the tab order.

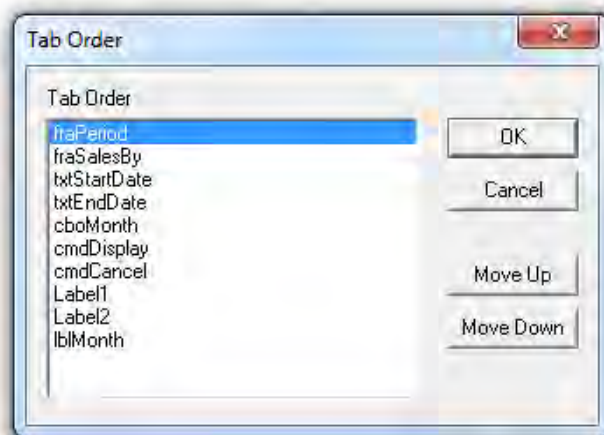



Quickly display the Tab Order dialog box by right-clicking a blank portion of the form and selecting *Tab Order* from the shortcut menu.



Setting the Tab Order

1. Select the *RunReports* form.
2. To run the form, press **[F5]**.
3. Tab through the form.
4. Click  Close on the form.
5. In the Visual Basic Editor, open the View menu and select *Tab Order*.
6. Select *fraPeriod* and click Move Up until it appears at the top of the Tab Order list.
7. Select control names and use Move Up and Move Down to change the tab order of the form to match the order shown in the illustration below:



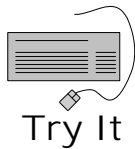
8. Click OK in the Tab Order dialog box.
 9. At the top of the Properties window open the object drop-down list select *txtStartDate*.
 10. Change the Enabled property for *txtStartDate* to *True*.
 11. Change the Enabled property of *txtEndDate* to *True*.
 12. Select the form and then press **[F5]** to run it.
 13. Tab through the form to see how the tab order has been affected.
 14. Click  Close on the form.
 15. Return the Enabled property of the text boxes to *False*.
 16. Save your work.
-

Populating a Control

A list box or a combo box control that you place on your form is not functional until you add the data that will appear in the list. This is accomplished by putting code in the sub procedure that is associated with the Initialize event of the form. The code placed in the sub procedure will execute when the event is triggered. The Initialize event is triggered when the UserForm is loaded. The AddItem method of the list or combo box control is used to specify the text that appears in the list. Since you will typically be adding a number of items to the same control, using a With statement will make coding easier.

Shown below is code to add three items to a combo box named cboColorChoices:

```
With cboColorChoices
    .AddItem "Red"
    .AddItem "Blue"
    .AddItem "Green"
End With
```



Populating a Control

1. Right-click the Run Reports form and select *View Code*.
2. Select *UserForm* from the Object drop-down list, if necessary.
3. Select *Initialize* from the Procedure drop-down list.
4. In the UserForm_Initialize sub procedure, enter the following code:

```
With cboMonth
    .AddItem "Jan"
    .AddItem "Feb"
    .AddItem "Mar"
    .AddItem "Apr"
    .AddItem "May"
    .AddItem "Jun"
    .AddItem "Jul"
    .AddItem "Aug"
    .AddItem "Sep"
    .AddItem "Oct"
    .AddItem "Nov"
    .AddItem "Dec"
End With
```

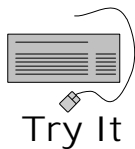
5. Double-click frmRunReports in the Project window to display the RunReports form.
 6. Press **[F5]** to run the form.
 7. Test the combo box.
 8. Close the form.
 9. Set the Text property of the cboMonth combo box to Jan.
 10. Run the form again and notice the default text in the combo box.
 11. Close the form.
 12. Save your work.
-

Adding Code to Controls

As we have seen, forms and the controls they contain are capable of responding to various events. It is the code that we create to run in response to events that gives the form its functionality. Often, the event that creates action on a form is a button click. Adding code to form and control events is accomplished in the same way as adding code to events of other objects.



Double-click the form or one of its controls to display the Code window for the form.



Adding Code to Controls

In this exercise we'll add functionality to our form by adding code to respond to some control events. We want to make certain form options available only when certain other options are selected. Specifically, when *Other period* is selected in the Period frame we want to activate the Start Date and End Date text boxes; when the Month period is selected, we want to unhide the WhichMonth combo box and its label. We'll also add code to unload the form when the Cancel button is clicked. The code for the click event of the Display button will be developed in the next section.

1. In the UserForm window, double-click a blank area of frmRunReports to display its code.
2. Under the existing code in the UserForm_Initialize event sub procedure, add the following lines of code:

```
cboMonth.Visible = False  
lblMonth.Visible = False
```

3. Add the following code to the optMonth_Change event sub procedure. It hides or shows the WhichMonth combo box and its label when the Month option is selected:

```
Private Sub optMonth_Change()  
  
If optMonth = True Then  
    cboMonth.Visible = True  
    lblMonth.Visible = True  
Else  
    cboMonth.Visible = False  
    lblMonth.Visible = False  
End If  
  
End Sub
```

continued on next page

Adding Code to Controls, continued

4. Create code similar to the code in the previous step for the Change event of optOther that will enable the Start Date and End Date text boxes when optOther is selected. (The code appears on the next page.)
5. Enter the following line of code for the cmdCancel_Click event to close the form when the Cancel button is clicked.

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

6. Under the Option Explicit statement at the top of frmRunReport's code module, declare two public string variables to hold the values selected in the frame controls:

```
Option Explicit
Public strSalesBy as String
Public strPeriod as String
```

7. Insert the structure for the optMonth_Click event procedure into the module. Enter the following line of code into the procedure to set the strPeriod variable when the Month option is selected in the Period frame.

```
Private Sub optMonth_Click()
    strPeriod = "Month"
End Sub
```

8. Create similar Click event procedures for the other option buttons in the Period and Sales By frames using the information below:

Option Name	Variable to Set	Value of Variable
optAll	strPeriod	All
optOther	strPeriod	Other
optSalesperson	strSalesBy	Salesperson
optModel	strSalesBy	Model
optClassification	strSalesBy	Classification

9. Save your work and then run the form.
 10. Try to select the Start and End Date text boxes.
 11. Select the *Other Period* option and then select the Start and End Date text boxes.
 12. Select the *Month* option; select a month from the combo box drop-down list.
 13. Select *All* in the Period frame.
 14. Click Cancel.
-

Exercise Code – Adding Code to Controls

```
Option Explicit
Public strSalesBy As String
Public strPeriod As String
-----
Private Sub cmdCancel_Click()
    Unload Me
End Sub
-----
Private Sub optAll_Click()
    strPeriod = "All"
End Sub
-----
Private Sub optClassification_Click()
    strSalesBy = "Classification"
End Sub
-----
Private Sub optModel_Click()
    strSalesBy = "Model"
End Sub
-----
Private Sub optMonth_Change()
    If optMonth.Value = True Then
        cboMonth.Visible = True
        lblMonth.Visible = True
    Else
        cboMonth.Visible = False
        lblMonth.Visible = False
    End If
End Sub
-----
Private Sub optMonth_Click()
    strPeriod = "Month"
End Sub
-----
Private Sub optOther_Change()
    If optOther = True Then
        txtStartDate.Enabled = True
        txtEndDate.Enabled = True
    Else
        txtStartDate.Enabled = False
        txtEndDate.Enabled = False
    End If
End Sub
-----
Private Sub optOther_Click()
    strPeriod = "Other"
End Sub
-----
Private Sub optSalesperson_Click()
    strSalesBy = "Salesperson"
End Sub
-----
```

Exercise Code – Adding Code to Controls

```
Private Sub UserForm_Initialize()  
    With cboMonth  
        .AddItem "Jan"  
        .AddItem "Feb"  
        .AddItem "Mar"  
        .AddItem "Apr"  
        .AddItem "May"  
        .AddItem "Jun"  
        .AddItem "Jul"  
        .AddItem "Aug"  
        .AddItem "Sep"  
        .AddItem "Oct"  
        .AddItem "Nov"  
        .AddItem "Dec"  
    End With  
    cboMonth.Visible = False  
    lblMonth.Visible = False  
End Sub
```

Launching a Form in Code

To launch a form from within a procedure use the Show method of the form object. By creating a procedure that launches a form, it allows the form to be launched on events such as the opening of a workbook. It also enables you to launch a form from the Quick Access toolbar.

Following is the syntax that launches a form:

```
FormName.Show  
  
frmRunReports.Show
```



Launching Forms from Procedures

1. Insert a new module into the Fiscal Years Sales project and name it modReports.
2. Create the following procedure in modReports:

```
Sub LaunchReports()  
    frmRunReports.Show  
End Sub
```

3. Run the LaunchReports procedure.
 4. Click Cancel.
 5. Save the workbook.
 6. Close the Visual Basic Editor.
-



Working with the PivotTable Object

Understanding PivotTables

Creating a PivotTable using worksheet data

Working with PivotTable objects

Working with the PivotFields collection

Assigning a macro to the Quick Access Toolbar

Understanding PivotTables

A PivotTable is a table in a worksheet that can be used to summarize data from a worksheet or an external source such as a database. When creating a PivotTable, you can select the data to analyze and the method to be used to summarize the data. After creating the PivotTable, you can rearrange the column and row headings to display different views of the data.

Below is an illustration of a PivotTable:

Year	(All)				
Sum of Selling Price	Column Labels				
Row Labels	Car	Suv	Truck	Van/Minivan	Grand Total
Buick	\$109,555.00				\$109,555.00
Cadillac	\$126,125.00	\$96,974.00			\$223,099.00
Chevrolet	\$18,485.00	\$79,421.00	\$21,542.00	\$18,808.00	\$138,256.00
GMC		\$45,062.00		\$35,289.00	\$80,351.00
Oldsmobile	\$139,127.00	\$34,625.00		\$32,768.00	\$206,520.00
Pontiac	\$108,275.00				\$108,275.00
Grand Total	\$501,567.00	\$256,082.00	\$21,542.00		

The PivotTable Field List appears in a task pane when the PivotTable is selected. The selections in the Field List shown here correspond to the PivotTable in the picture above.

PivotTable Fields

Choose fields to add to report:

☐ Vin Number
☒ Year
☒ Make
☐ Model
☒ Classification
☐ Color
☐ Dealer Cost
☐ MSRP

Drag fields between areas below:

FILTERS

Year

COLUMNS

Classification

ROWS

Make

VALUES

Sum of Selling P...

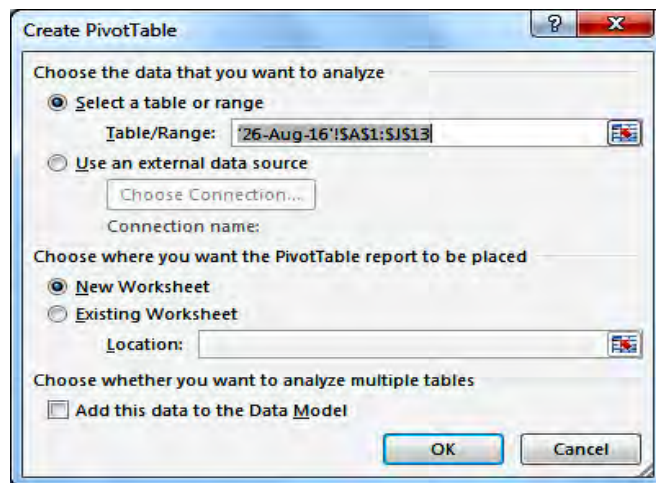
☐ Defer Layout Update

UPDATE

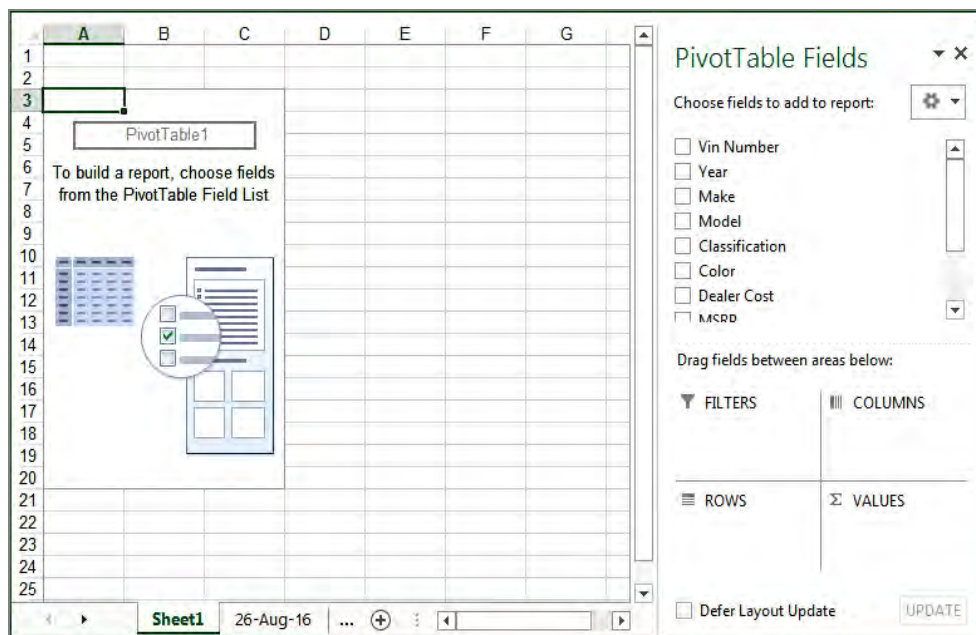
Creating a PivotTable Using Worksheet Data

A PivotTable is created by setting the data and the location for the PivotTable in the Create PivotTable dialog box. Once the PivotTable structure is created, the table is filled with data by selecting the row, column, value, and filter fields from the PivotTable Field List.

The Create PivotTable dialog box is shown below:




Creating the PivotTable results in an empty PivotTable structure. The PivotTable Field List shows the fields from the data upon which the PivotTable is based. Both are shown here:




Creating a PivotTable Using Worksheet Data



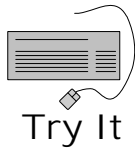
STEPS

1. Click the Insert tab.
2. In the Tables group, click  PivotTable.
3. In the Create Pivot Table dialog box, verify or enter the data range in the Table/Range text box.
4. Make a selection from the options under Choose where you want the PivotTable report to be placed.
5. Click OK.
6. In the PivotTable Field List, drag the field names to the desired filter, row, column, and value areas.






To delete a PivotTable from a worksheet, click the Options tab of the PivotTable Tools. In the Actions group, click  Select and then choose *Entire Table*. With the table selected, press **Delete**.

The PivotTables Tools are only visible when the cell pointer is within a PivotTable.



Creating a PivotTable Using Worksheet Data

In this chapter, we'll complete the functionality of `frmRunReports`. We'll be using PivotTables to display the data in the Fiscal Year Sales workbook according to selections made in the form. In this exercise, we'll create a PivotTable while running a macro so that we can understand which objects are necessary when manipulating PivotTables programmatically.

1. In the **Fiscal Year Sales** workbook, display the 26-Aug-16 worksheet and select cell A1.
2. Display the Developer tab and find the Code group; click .
3. Enter `CreatePivotTable` as the Macro name. Make sure that *This Workbook* is the selection in the list under Store macro in and then click OK.
4. Click the Insert tab. In the Tables group, click  PivotTable.
5. In the Create PivotTable dialog box, under *Select a table or range*, verify that the data range from the 26-Aug-16 worksheet appears in the *Table/Range* text box. The column headings should be included with the data. The total should not be included in the range.
6. Verify that *New Worksheet* is selected under *Choose where you want the PivotTable report to be placed*; then click OK.
7. On the Status bar, click  to stop the macro recorder.
8. Start the macro recorder again to record a new macro named `SetPivotFields`.
9. In the PivotTable Fields list, drag the Year field to the Filters area.
10. Set the other fields for the PivotTable using the information below:

PivotTable Field	PivotTable Area
Classification	Columns
Make	Rows
Selling Price	Values

11. Stop the macro recorder.
12. Display the Visual Basic Editor.
13. Display the code window for the module that contains the two macros and review the recorded code.
14. Return to the Excel window and delete the worksheet containing the PivotTable.

Working with PivotTable Objects

Two objects are necessary when creating a PivotTable programmatically. One is the PivotTable object and the other is the PivotCache object. As you might guess, the PivotTable object represents a PivotTable in a workbook. The PivotCache object is something a little less obvious. When you create a PivotTable, a special memory location or “cache” is necessary to hold the data that the PivotTable is based on. The object representing a cache that stores PivotTable data is called the PivotCache object. A PivotCache object is part of the PivotCaches collection which holds all of the PivotCaches for the PivotTables in a workbook.

The PivotCache needs to exist before you can create a PivotTable. Use the Create method of the PivotCaches collection to create a PivotCache. The arguments for the method are shown here:

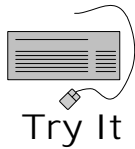
Argument	Definition
SourceType	Required. The type of PivotTable data indicated by one of the following constants, xlConsolidation, xlDatabase, xlExternal, xlPivotTable, or xlScenario.
SourceData	Optional. A Range object specifying the data for the PivotTable.
Version	Optional. Sets the Excel version of the PivotTable. Can be used to ensure that a PivotTable created in a later version of Excel is compatible with an earlier version of Excel.

Once you create the PivotCache, use the CreatePivotTable method of the PivotCache object to create the PivotTable. The arguments of the CreatePivotTable method are explained in the table below:

Argument	Definition
TableDestination	Required. The upper-left cell of the PivotTable’s destination range.
TableName	Optional. A name for the PivotTable.
ReadData	Optional. True or False to indicate whether to create a cache containing all of the records from an external database.
DefaultVersion	Optional. The default version of the PivotTable report.



You can create both the PivotCache and PivotTable objects using just one line of code.



Working with PivotTable Objects

1. Display the Visual Basic Editor and enter the following procedure into modReports. This code creates the PivotTable programmatically, capturing the data in the current worksheet and then placing a PivotTable in a worksheet called Reports. For now the PivotTable will contain no data because we haven't assigned the row, column, or data fields.

```
Sub MakePivot()  
    Dim Destination As Range  
    Dim DataRange As Range  
  
    Set Destination = Worksheets("Reports").Range("A1")  
    Set DataRange = Range("A1", Range("J1").End(xlDown))  
  
    Activeworkbook.PivotCaches.Create _  
        (SourceType:=xlDatabase, SourceData:=DataRange) _  
        .CreatePivotTable TableDestination:=Destination, TableName:="PivotInfo"  
End Sub
```

2. Compile and save your work.
 3. Switch to the Excel window.
 4. In the **Fiscal Year Sales** workbook, create a new worksheet named Reports.
 5. Select any sheet that contains data.
 6. Run the MakePivot sub procedure.
 7. Switch to the Reports worksheet to view the PivotTable structure.
 8. Save the workbook.
-

Working with the PivotFields Collection

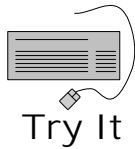
The PivotFields collection is a member of the PivotTable object and consists of the columns of data in the data source with each PivotField getting its name from the column header. In the Fiscal Year Sales example, the PivotFields are: Vin Number, Year, Make, Model, Classification, Color, Dealer Cost, MSRP, Selling Price, and Salesperson. In the Excel user interface, we drag PivotFields from the PivotTable Fields list to the area of the PivotTable where we want them to appear. In VBA, we accomplish the same result by setting the Orientation property of the PivotField to a constant representing the destination.

The following table lists the possible PivotTable destinations and the corresponding constants for PivotFields:

Destination	Constant
Row Labels	xlRowField
Column Labels	xlColumnField
Report Filter	xlPageField
Values	xlDataField
To Hide a field	xlHidden

The syntax for setting the Orientation property of a PivotField to the desired destination constant is shown here:

```
.PivotTables(Index).PivotFields(Index).Orientation = Destination  
  
.PivotTables("PivotInfo").PivotFields("Make").Orientation = xlRowField
```



Working with the PivotFields Collection

In the following exercise we are going to create a sub procedure that sets the Orientation property of PivotFields named after the following public variables: strPageName, strRowName, strColumnName, and strDataName. Eventually, these variables will be filled with selections made on the Run Reports form.

PivotSelect is a method of the PivotTable object that is used to select PivotFields. In the following code, PivotSelect is used to select the DataField so that we can apply numeric formatting to it.

1. Enter the following sub procedure into modReports in the Fiscal Year Sales project. Add the Public variables to the General Declarations area of the module.

```
Public strPageName As String
Public strRowName As String
Public strColumnName As String
Public strDataName As String

Sub SetPivot()

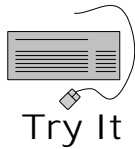
    Dim pvtTable As PivotTable
    Set pvtTable = Worksheets("Reports").PivotTables("PivotInfo")
    With pvtTable
        .PivotFields(strPageName).Orientation = xlPageField
        .PivotFields(strRowName).Orientation = xlRowField
        .PivotFields(strColumnName).Orientation = xlColumnField
        .PivotFields(strDataName).Orientation = xlDataField
    End With

    ActiveWorkbook.Sheets("Reports").Activate
    pvtTable.PivotSelect "", xlDataOnly
    Selection.NumberFormat = "$#,##0"
    Range("E1").Select
End Sub
```

2. Until the UserForm is run, the variables will contain no values. For the sake of testing, temporarily assign values to the variables using the Immediate window. Enter the following code in the Immediate window. (Note that these are column names from the worksheet data and must match exactly.) Make sure to press after each line.

```
strPageName = "Year"
strRowName = "Make"
strColumnName = "Salesperson"
strDataName = "Selling Price"
```

3. Run SetPivot.
4. View the Reports worksheet and then delete it.
5. Save the workbook.



Review - Completing the Run Reports Form

A form's functionality comes from its ability to call procedures to perform desired tasks. Once a form is created and has code that evaluates what selections the user has made, code needs to be created that distributes those responses to the correct procedures.

We have begun a UserForm named Run Reports that allows users to make selections for custom reports. We've also explored the PivotTable object and how to work with it in Visual Basic. In this exercise we will add code to our form to create PivotTable reports based on the user selections.

The event that will launch our code is the clicking of cmdDisplay. In English, here is what we want to happen when the Display button is clicked:

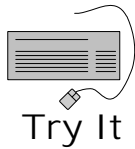
- The form is hidden.
 - The option selected in fraPeriod is evaluated. Depending on the selected option, either the month or a start and end date for the report are determined.
 - The option selected in fraSalesBy is evaluated. The variables that hold the PivotField names are set based on this selection.
 - The data upon which the report will be based is pulled from the workbook and put into a worksheet.
 - The data is moved to a new workbook and the PivotTable is created.
1. Display the Code window for frmRunReports.
 2. Use the Object and Procedure drop-down lists to insert the structure for the cmdDisplay_Click event procedure.
 3. Enter the following code to close the form once the user has clicked the Display button:

```
Private Sub cmdDisplay_Click()  
    Me.Hide  
End Sub
```

4. In the General Declarations area of the module, declare the following public variables.

```
Public strMonth as String  
Public datStartDate as Date  
Public datEndDate as Date
```


continued on next page



Review - Completing the Run Reports Form, continued

5. Move to the end of the frmRunReports code module and enter the procedure shown below. The StrPeriod variable, as you may recall, is set when an option in the Period frame is selected. GetEarliestDate and GetMostRecentDate are the names of functions that we'll use to find the earliest and latest dates in the workbook.

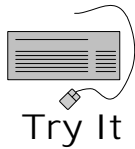
```
Sub GetWhichDates()  
  
    Select Case strPeriod  
        Case "Month"  
            strMonth = cboMonth  
        Case "All"  
            datStartDate = GetEarliestDate  
            datEndDate = GetMostRecentDate  
        Case "Other"  
            datStartDate = txtStartDate  
            datEndDate = txtEndDate  
    End Select  
End Sub
```

6. Move to a blank line at the end of the module.
7. Open the Insert menu and select *File*. Navigate to the folder where the student data is stored. In the InsertText folder, select **GetMostRecentDate.txt** and click Open.
8. Review the GetMostRecentDate function.
9. Display the Immediate window, if necessary. Type the following code and press  to call the function and see its result:

```
? frmRunReports.GetMostRecentDate
```

10. Create a copy of the GetMostRecentDate function at the end of module.
11. Change the name of the function to GetEarliestDate.
12. Change the code in the GetEarliestDate function so that it will return the earliest date in the workbook. (The completed code is at the end of the exercise.)

continued on next page



Review - Completing the Run Reports Form, continued

13. Still within the code module for frmRunReports, create a procedure that uses a Select Case structure to assign values to the PivotField variables according to the option selected in the Sales By frame. Earlier we created the variable strSalesBy to store the selection made in the frame. The procedure is started below:

```
Sub GetSalesByGroup()
    Select Case strSalesBy
        Case "Salesperson"
            strPageName = "Salesperson"
            strRowName = "Year"
            strColumnName = "Make"
            strDataName = "Selling Price"

    End Select
End Sub
```

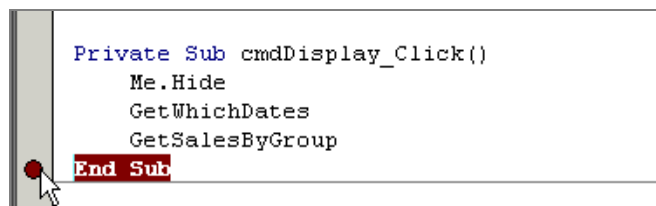
14. Finish the procedure by adding the cases for Model and Classification. For those cases the variables should be assigned as follows: (The completed code is at the end of the exercise.)

Variable	Model	Classification
strPageName	Model	Classification
strRowName	Year	Make
strColumnName	Color	Year
strDataName	Color	Selling Price

15. Return to the Click event for cmdDisplay and call GetWhichDates and GetSalesByGroup:

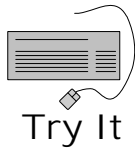
```
Sub cmdDisplay_Click()
    Me.Hide
    GetWhichDates
    GetSalesByGroup
End Sub
```

16. As shown in the illustration below, click the gray bar next to the End Sub statement in the cmdDisplay_Click procedure to pause the program before it ends:




17. Compile and save your work.
18. Run the Run Reports form.


continued on next page



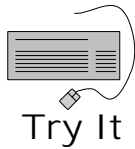
Review - Completing the Run Reports Form, continued

19. Select *All* in the Period frame and *Model* from the Sales By frame.
20. Click Display.
21. When the program pauses, type the following in the Immediate window and press  to see the value of strPeriod.

? frmRunReports.strPeriod

22. Use the Immediate window to test the value of strSalesBy, datStartDate, and datEndDate.
23. Click  Reset.
24. In the Code window for the form, click the dot in the gray bar to remove it.
25. Display the Code window for modReports and move the insertion point to the end of the module.
26. Open the Insert menu and select *File*.
27. Navigate to the student data directory and select the text file **ConsolidateData** from the InsertText folder.
28. Click Open.
29. The inserted code contains procedures named ConsolidateData, GrabCells, and FinishReport. Review the information below about the procedures.
 - ConsolidateData is a sub procedure that creates a temporary worksheet, named Reports, to hold the PivotTable data in **Fiscal Year Sales**. It collects data from the worksheets in the **Fiscal Year Sales** workbook based on the time period that was selected in the Period frame. For each sheet of data to be included, it calls a function called GrabCells.
 - GrabCells is a function that is passed a value containing the row number from which to begin to get cells. The first time it runs it begins at row 1 to get the column headers, which the PivotTable needs to run. Every subsequent time it starts at row 2 so the headings are not copied again. The GrabCells function copies the data in the worksheets to the Reports worksheet.
 - FinishReport is a sub procedure that moves the collected data into a workbook called Reports, deletes the temporary worksheet in **Fiscal Year Sales**, and creates a PivotTable in **Reports**. The third line of the FinishReport sub procedure opens a workbook called **Reports** in the current directory.

continued on next page



Review - Completing the Run Reports Form, continued

30. Switch to the code module for `frmRunReports`, move to the `cmdDisplay_Click` event procedure and add the following code before the end sub line:

```
Me.Hide
GetWhichDates
GetSalesByGroup
ConsolidateData StrMonth
FinishReport
Unload Me
```

31. Return to `modReports` and locate the `SetPivot` sub procedure. The code formats the data in the table as currency which is not appropriate when the Sales By option is Model because the data is a count of inventory, not a dollar figure. In the procedure, add the unshaded lines of code shown below to create the formatting based on the value of the `strSalesBy` variable:

```
Sub SetPivot
...
ActiveWorkbook.Sheets("Reports").Activate
If frmRunReports.strSalesBy <> "Model" Then
    pvtTable.PivotSelect "", xlDataOnly
    Selection.NumberFormat = "$#,##0"
End If
Range("E1").Select
End Sub
```

32. While still in `modReports`, locate the `MakePivot` sub procedure. Modify the line of code that sets the `DataRange` to match the code below. (Change "J1" to "I1".) This is because it is not necessary to get the Vin Number field for the PivotTable, so the consolidated data only fills columns A:I.

```
Set DataRange = range("A1", range("I1").End(xlDown))
```

33. Save the workbook.
34. Run the form.
35. Select *Month* for the period and then select *Jul* from the combo box.
36. Select *Salesperson* from the Sales By frame and click Display.
37. When prompted, click Delete to delete the temporary worksheet created in **Fiscal Year Sales**.
38. View and then close the Reports workbook. (It is not necessary to save it.)
39. Run the form a few more times checking that the form is functioning correctly. Note that you can only select a month for which data exists, i.e. Jun, Jul, or Aug. Remember to close the **Reports** workbook after each running of the form.

Exercise Code – Completing the Run Reports Form

```
Function getEarliestDate()  
    Dim wsSheet As Worksheet  
    Dim datEarliest As Date  
    Dim datTestDate As Date  
  
    datEarliest = 99999  
    For Each wsSheet In Worksheets  
        datTestDate = CDate(wsSheet.Name)  
        If datTestDate < datEarliest Then  
            datEarliest = datTestDate  
        End If  
    Next wsSheet  
    getEarliestDate = datEarliest  
End Function  
-----  
Sub GetSalesByGroup()  
    Select Case strSalesBy  
        Case "Salesperson"  
            strPageName = "Salesperson"  
            strRowName = "Year"  
            strColumnName = "Make"  
            strDataName = "Selling Price"  
        Case "Model"  
            strPageName = "Model"  
            strRowName = "Year"  
            strColumnName = "Color"  
            strDataName = "Color"  
        Case "Classification"  
            strPageName = "Classification"  
            strRowName = "Make"  
            strColumnName = "Year"  
            strDataName = "Selling Price"  
    End Select  
End Sub
```

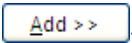



Assigning a Macro to the Quick Access Toolbar

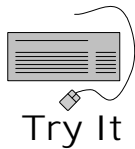
Macros can be executed from the Macro dialog box and from within other procedures. Additionally, you can make a macro more accessible by adding an icon to the Quick Access Toolbar that runs the macro.



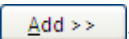
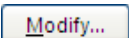
STEPS

Assigning a Procedure to the Quick Access Toolbar

1. Click the File tab and then click Options.
2. Select the Quick Access Toolbar category.
3. Open the *Choose commands from* list and select *Macros*.
4. Open the *Customize Quick Access Toolbar* drop-down list and make the desired selection.
5. Select the macro to be added to the Quick Access Toolbar.
6. Click .
7. To modify the appearance and text for the button:
 - Click Modify;
 - In the Modify Button dialog box, select an icon for the button;
 - In the Display Name text box, type a name for the button;
 - Click OK.
8. Use  or  to position the tool on the Quick Access toolbar.
9. Click OK.



Assigning a Macro to the Quick Access Toolbar

1. In **Fiscal Year Sales**, Click the File tab and then click Options.
 2. Select the Quick Access Toolbar category.
 3. Open the *Choose commands from* list and select *Macros*.
 4. Open the *Customize Quick Access Toolbar* drop-down list and select *For Fiscal Year Sales.xlsm*.
 5. Select *LaunchReports* from the Macros list.
 6. Click .
 7. Click .
 8. Select a symbol to appear on the button.
 9. In the Display name text box, modify the name to include a space between the words “Launch” and “Reports”.
 10. Click OK in the Modify Button dialog box.
 11. Click OK in the Excel Options box.
 12. On the Quick Access toolbar, test the Launch Reports button.
 13. Click Cancel.
 14. Save and close the workbook. Verify that the tool no longer appears on the Quick Access toolbar.
 15. Open the **Fiscal Year Sales** workbook.
 16. Verify that the Launch Reports button appears on the Quick Access toolbar.
-

8

Debugging Code

Understanding errors

Using debugging tools

Setting breakpoints

Stepping through code

Using Break mode during Run mode

Determining the value of expressions

Understanding Errors

Throughout program development, problems with the project will occur. Incorrect control logic or function usage, as well as overflow and division by zero errors are just a few of the things that will cause an application to break or not produce the intended results.

Errors are often called *bugs* and the process of removing bugs is called *debugging*. To assist you in resolving errors, VBA provides tools that help you see what is happening when your code runs.

There are three general types of errors that can occur:

Syntax Errors

Syntax errors occur when code is entered incorrectly and are typically discovered by the line editor or the compiler.

- *Discovered by Line Editor:* When you move off of a line of code in the Code window, the syntax of the line is checked. If a syntax error is located in the line of code, the entire line will turn red by default to indicate that the line needs to be changed. Examples of common syntax errors that the line editor will detect are: not completing an expression or not entering all required arguments for a procedure.
- *Discovered by Compiler:* Whereas the line editor only checks one line of code at a time, the compiler checks all lines within each procedure and all declarations within the project. If **Option Explicit** appears, the compiler checks that all variables are declared and that all objects have references to correct methods, properties, and events. The compiler also checks block statements to ensure that all required statements are present, for example, that each *If* has a corresponding *End If*, and so forth. When the Compiler locates an error, it generally displays a message box describing the error.

Run-Time Errors

When a program is running and encounters a line of code that cannot be executed, a run-time error is generated. These types of errors often occur when a certain condition exists. A procedure may work ten consecutive times, but on the eleventh time an error may occur. Recognizing what condition was different on the eleventh pass through the procedure is the key to understanding what caused the error. When a run-time error occurs, execution is halted and a message box appears that defines the error.

Logic Errors

Logic errors create unexpected outcomes when a procedure is executed. Unlike syntax or run-time errors, your application will not be halted and you will not be shown an offending line of code. These errors are generally more difficult to locate and correct.

Understanding Errors, continued:

Strategies for Minimizing Errors

Unfortunately, there is no science to debugging errors. Finding errors is often the result of detective work, patience, and some lucky guessing. What follows are a few suggestions that you can use to help minimize errors or make it easier to locate errors in your code when they do occur:

- Add comments to your code. This is particularly important if someone else will be looking at your code. Well-written comment lines explain what a line of code or procedure is supposed to do and help you to find the code of interest when debugging.
- Create meaningful variable names so that it is easy to recognize what a variable represents. Use variable prefixes to indicate the data or object type.
- Any time you use division that contains a variable in the denominator, first test the denominator to ensure that it is not equal to zero.
- Force variable declarations with the use of Option Explicit. A simple misspelling of a variable name will lead to a logic error, not a run-time error.
- Give procedures names that clearly describe what they do.
- Keep each procedure as short as possible, giving it one or two specific tasks to carry out.
- Test procedures with large data sets that represent all possible permutations of reasonable and unreasonable data. In short, try to make the procedure fail before someone else does.

If You're Stuck

There will be times when you have a bug that isn't easily fixed. Sometimes it may be something simple that you just overlook. It's also possible that the error is not in your code, but caused by a bug in Excel itself. Whatever the reason, the following suggestions may be helpful to try when you are stuck:













- Take a break for a while and come back to the problem later.
- Ask a co-worker for help. They may be more likely to find that missing quotation mark or ampersand. You may resolve the problem yourself in the process of explaining it to someone else.
- Call or email a support line, or search or post a message to a user group.

Using Debugging Tools

VBA's debugging tools are useful for finding and understanding the cause of run-time and logic errors in your code. They can be generally categorized in three ways:

- Tools that suspend the execution of your application.
- Tools that help you manually control the execution of statements while the program is suspended.
- Tools that help you determine the value of expressions.

The following is an explanation of the debugging tools on the Debug toolbar:









	Run	runs code or resumes the running of code after a break.
	Break	pauses the execution of code and allows you to continue execution from the break point.
	Reset	stops code execution.
	Toggle Breakpoint	a user-determined line of code at which execution will break.
	Step Into	executes the next line of code.
	Step Over	allows selected code to be stepped over during execution.
	Step Out	executes remaining code in the calling procedure after a break.
	Locals Window	displays the value of variables and properties during code execution.
	Immediate Window	displays a window where individual lines of code can be executed and variables can be evaluated.
	Watch Window	displays a window for monitoring specified expressions with options for breaking when the value of an expression changes or is True.
	Quick Watch	returns the value of an expression.
	Call Stack	displays all procedures that have been called up to a breakpoint in code execution.

Using Debugging Tools, continued:

Most debugging is done when the application is suspended which is also known as being in *Break mode*. In this mode, everything loaded into memory remains in memory and can be evaluated. A program typically enters Break mode in one of the following ways:

- A code statement generates a run-time error.
- A breakpoint is intentionally set on a line of code.
- A **Stop** statement is entered within the program code.

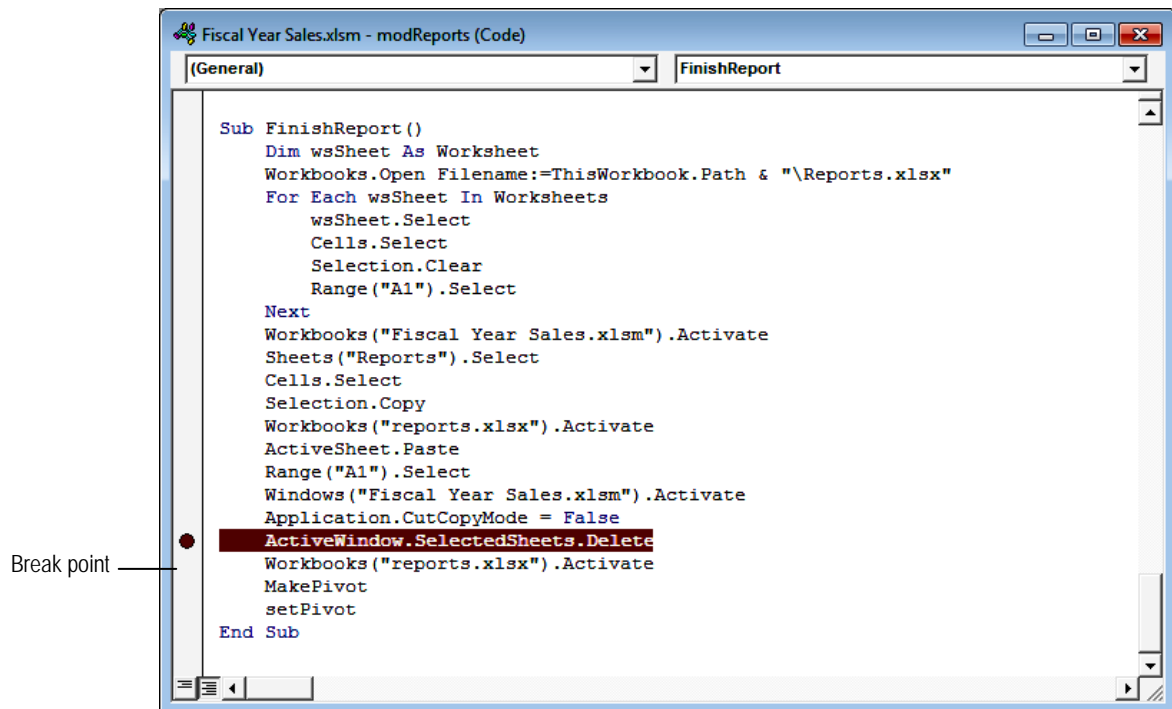
The table below summarizes when the debugging tools are available:

Tool	Design Mode	Break Mode	Run Time
 Step Into	✓	✓	
 Step Over		✓	
 Step Out		✓	
 Locals Window		✓	
 Immediate Window	✓	✓	✓
 Watch Window		✓	✓
 Quick Watch		✓	
 Call Stack		✓	
Auto Data Tips		✓	

Setting Breakpoints

By setting a breakpoint, you identify the location in your program where you want it to enter into Break mode. The program runs to the line of code and stops before executing it. The Code window displays and the line of code where the breakpoint is set is highlighted. When the code is halted, the value of a variable or expression in a certain line of code can be checked, either in the Immediate window or by holding the mouse pointer over the expression.


The illustration below shows the Code window with a line of code marked as a breakpoint:





STEPS

Setting Breakpoints

1. Open the Code window and select the desired procedure.
2. Position the insertion point on the desired line of code.
3. To set a breakpoint, perform one of the following:
 - Open the Debug menu; select *Toggle Breakpoint*.
 - Click  Toggle Breakpoint on the Debug toolbar.



The **Stop** command can also be entered within a procedure to halt program execution.

Each of the commands to set a breakpoint works as a toggle switch.



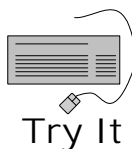
Click in the gray bar to the left of the line of code to quickly set a breakpoint.

Position the insertion point on the desired line of code and press **[F9]** to set a breakpoint. Pressing **[F9]** again on the same line of code will remove the breakpoint.

To remove all breakpoints in a project, press **[Ctrl] – [⇧ Shift] – [F9]**.



Breakpoints are cleared upon closing the project. Stop statements are permanent until you remove them from code.



Try It

Setting Breakpoints





1. Open the **Fiscal Year Sales** workbook, if necessary.
2. Display the Visual Basic Editor.
3. Open the Code window for modReports.
4. In the FinishReport procedure, click in the gray bar on the left to set a breakpoint on the following line of code:

```
ActiveWindow.SelectedSheets.Delete
```

Stepping through Code

The step tools give you the opportunity to step one line at a time through your code to see exactly which statements in your procedures are being executed and where things may be going wrong.





The step tools are accessible through the Debug menu and the keyboard; most are also available on the Debug toolbar as well. They are described below:

	<i>Step Into</i>	F8	steps through the current procedure one statement at a time, pausing after each step. If a statement calls another procedure, execution steps into the called procedure and continues to execute statements one at a time.
	<i>Step Over</i>	⇧ Shift – F8	steps through the current procedure one statement at a time. However, if a statement calls another procedure, the procedure is executed without pausing.
	<i>Step Out</i>	Ctrl – ⇧ Shift – F8	runs the rest of the code in a procedure without pausing if you are otherwise stepping through code. If control is returned to a calling procedure, break mode is reentered.
	<i>Run to Cursor</i>	Ctrl – F8	runs from the current statement to the location of your cursor in the Code window if you are otherwise stepping through code.
	<i>Set Next Statement</i>	Ctrl – F9	runs a statement of your choice rather than the next statement.
	<i>Call Stack</i>	Ctrl – L	lists all currently loaded procedures.




STEPS

Stepping Through Code

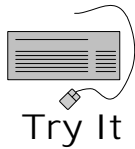
1. Display the desired procedure.
2. Set a breakpoint or type `Stop` at the desired line of code.
3. Execute the procedure.
4. To execute code a line at a time from the current line, do one of the following:
 - Click  Step Into once for each line of code to be executed.
 - Press `[F8]` once for each line of code to be executed.
 - Open the Debug menu; choose *Step Into* once for each line of code to be executed.
5. To execute all lines of code in a called procedure while stepping through code in the current procedure, do one of the following:
 - Click  Step Over.
 - Press `[Shift] – [F8]`.
 - Open the Debug menu; select *Step Over*.
6. To run the rest of the current procedure and return to the preceding procedure:
 - Click  Step Out.
 - Press `[Ctrl] – [Shift] – [F8]`.
 - Open the Debug menu; choose *Step Out*.
7. To run a statement other than the next statement, do one of the following:
 - Place the cursor in the line of code to be executed next and press `[Ctrl] – [F9]`.
 - Place the mouse pointer over the yellow arrow that is pointing to the next line of code and drag the arrow to the line to be executed next.
8. To terminate the execution of the procedure:
 - Click  Reset.



When setting the next line of code, the mouse pointer will appear as  when it is properly positioned to drag the arrow to a new location.





Using the *Set Next Statement* option within a procedure requires that program execution has been suspended within that same procedure.



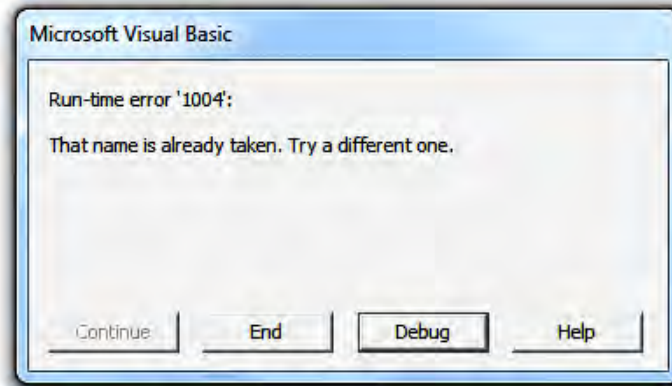
Stepping through Code

In this exercise, we'll be tiling the Visual Basic Editor window and the Excel window. Before beginning, close any other open program windows. **Fiscal Year Sales** should be the only open workbook.

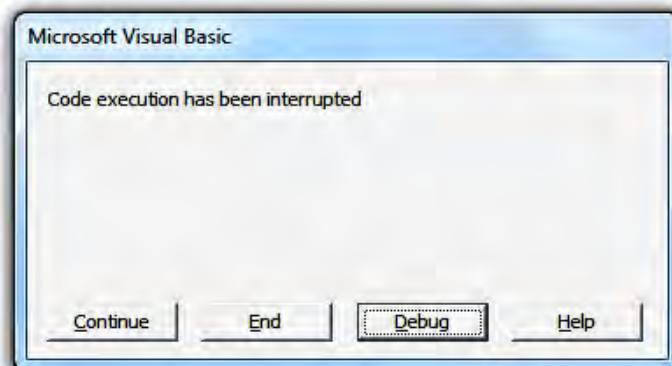
1. In the Visual Basic Editor, display the Debug toolbar, if necessary.
 2. Right-click the taskbar and select *Show windows side by side*.
 3. To see the Code window better, close the Properties window and the Project Manager, if necessary.
 4. In the Excel window, click the Launch Reports button on the Quick Access toolbar.
 5. Select *Other Period* under Period.
 6. Enter 8/01/16 in the Start Date text box.
 7. Enter 8/15/16 in the End Date text box.
 8. Select *Classification* under Sales By and click Display.
 9. When the program pauses, note the data on the Reports worksheet.
 10. On the Debug toolbar, click  Step Into.
 11. Click Delete in the message box and note the change in the Excel window.
 12. Press **[F8]** to execute the next line of code and note that the Reports workbook is now active.
 13. Press **[F8]** again on the line that calls the MakePivot sub procedure.
 14. Click  Step Out to run the MakePivot sub procedure without pausing. Note the changes in Excel. (In the Reports workbook, display the Reports worksheet, if necessary.)
 15. In the Visual Basic Editor, press **[F8]** on the line that calls the SetPivot procedure.
 16. Press **[Ctrl] – [⇧ Shift] – [F8]** to run SetPivot without pausing.
 17. Press **[F8]** four more times to finish; after each press of the key, note which code is being executed and the procedure that contains it.
 18. Right-click the taskbar and select *Undo Show side by side*.
 19. Remove the breakpoint from the FinishReport procedure.
 20. Close the **Reports** workbook without saving changes.
-

Using Break Mode during Run mode

Sometimes during code execution you will enter Break mode, either intentionally or because of a run-time error that suspends execution. When a run-time error occurs a message appears that describes the error. Clicking the Debug button in the message box will display the Code window with the offending line of code highlighted.




Once in a while it may be necessary for you to intervene in order to suspend program execution, often as a result of an endless loop. Pressing **Ctrl** – **Break** or the Break button in the Visual Basic Editor will suspend program execution and produce the following message:



Using Break Mode during Run Mode







STEPS

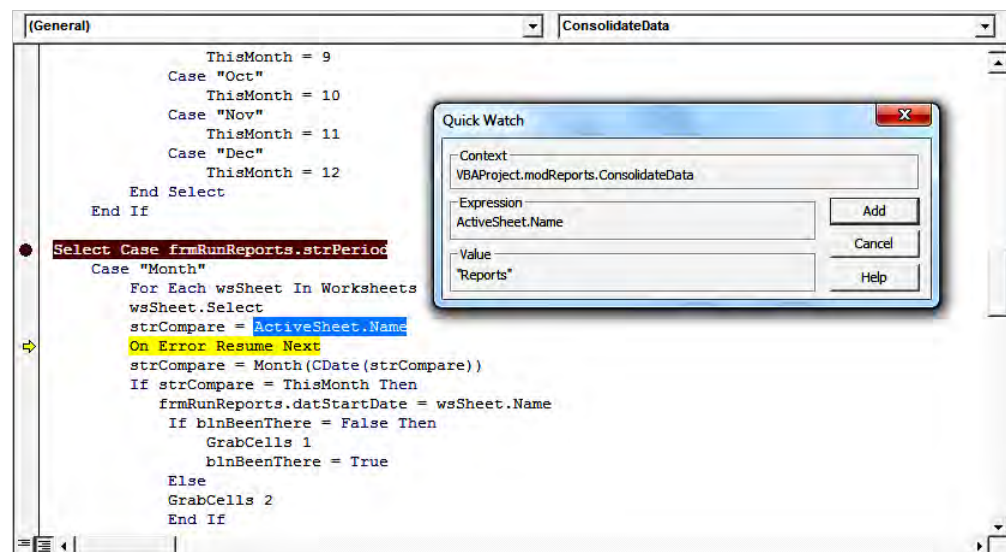
1. To work in Break mode when a run-time error occurs:
 - Click Debug in the error message window.
2. To intentionally stop program execution, do one of the following:
 - Press **Ctrl** – **Break**.
 - In the Visual Basic Editor, click  Break.

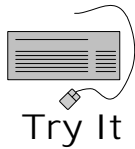
Determining the Value of Expressions

While debugging it is often useful to find out the value of variables and expressions while your code is executing. VBA has several tools for this purpose, which are listed and described below:

	Locals Window		displays the value of variables and properties during code execution.
	Immediate Window	Ctrl – G	displays a window where individual lines of code can be executed and querying of specific variables can occur. While in Break mode, the scope of the Immediate window matches that of the currently running procedure.
	Watch Window		displays the value of each expression that is added to the window.
	Quick Watch	Shift – F9	displays the value of a selected expression.
	Auto Data Tip		displays the value of the expression where the mouse is pointing.

The Quick Watch window is useful for expressions that cannot be determined using the Auto Data Tips feature. It is illustrated here:













Determining the Value of Expressions





1. Switch to the Visual Basic Editor. If necessary, maximize the window and display the Project Explorer.
2. Display the ConsolidateData procedure in modReports.
3. Set a breakpoint by the following line of code:

Select Case frmRunReports.strPeriod

4. Click  Locals Window on the Debug toolbar.
5. Switch to Excel and select the first sheet in the Fiscal Year Sales workbook.
6. Run frmRunReports using the Launch Reports button on the Quick Access toolbar.
7. Select *Month* under Period and *Aug* from the Month combo box.
8. Make a selection under Sales By and click Display.
9. When the program pauses, view the variable values in the Locals window.
10. Click  next to modReports to see the value of the public variables in the module. Click  to collapse the view of the public variables.
11. Press  until the **On Error Resume Next** statement is highlighted.
12. Place the mouse pointer over the variable name strCompare that appears on the next line of code to view the Auto Data Tip. (StrCompare is set to "Reports".)
13. In that same line select the expression: CDate(StrCompare)
14. Click  Quick Watch.
15. Note the value of the expression in the Quick Watch window. (Excel cannot convert the text into a date.)
16. Click Cancel to close the Quick Watch window.
17. Press  twice. The If statement is highlighted. Press  again to see that the statements in the If block do not execute since the test criterion is False.
18. Press  until the **On Error Resume Next** statement is highlighted again.
19. Place the mouse pointer over the variable strCompare to see the value.
20. Use the Quick Watch window to test the value of the following expressions found in the next line of code:
 - CDate(StrCompare)
 - Month(CDate(strCompare))

continued on next page

Determining the Value of Expressions, continued

21. Continue stepping through the code. After execution jumps to the GrabCells procedure, click  Step Out to run the procedure and return to the ConsolidateData procedure.
 22. Step through the code until the **On Error Resume Next** statement is highlighted again.
 23. In the Locals window, select the line that shows the ThisMonth variable information.
 24. Click again in the Value column and change the value shown to "7" and press .
 25. Press  three more times and see that the statements in the If block are not executed.
 26. Click  Reset.
 27. Remove the breakpoint from the ConsolidateData procedure.
 28. Close the Locals window.
-

9

Handling Errors

Understanding error handling

Understanding VBA's Error Trapping Options

Trapping Errors with the On Error statement

Understanding the Err object

Writing an error-handling routine

Working with inline error handling

Understanding Error Handling

Although the word *error* has a bad connotation, anticipating and handling errors is just another aspect of writing good code. The error handling mechanism in VBA allows you to enter an instruction into a procedure that directs the program in case of an error. When Visual Basic recognizes an error, it looks for the error-handling instruction within the procedure where the error occurred. Typically the instruction directs the program to an error-handling routine within the procedure that contains additional instructions to take care of the problem.

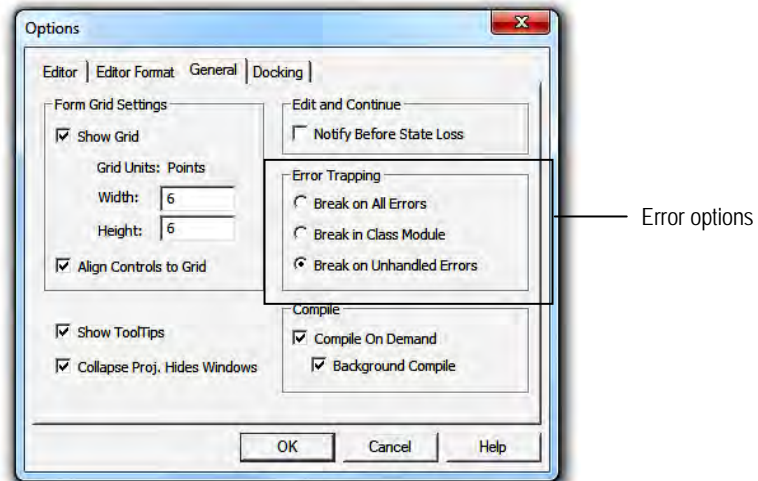
Error handling often deals with managing certain types of situations that happen within an application which may be extraneous to the task at hand. Maybe the task is to save a file, but the disk is out of space. Or maybe you created a procedure that expects an argument with a value between 1 and 5, and it receives a value of 7. You will want to write code to handle these types of issues as well as trap unexpected errors. Sometimes the most efficient course will be to simply ignore the error. Essentially, good error handling should keep your program from terminating when an error occurs.

It should also be noted that sometimes when an error is discovered, an effective way to address the problem is to prevent it from occurring by adding some decision structures to your code. For instance, a divide-by-zero error might best be handled by assuring that the divisor is not equal to zero before performing a division operation.

Understanding VBA's Error Trapping Options

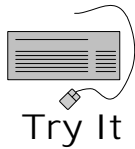
Before we dive into the details of error handling, you need to understand how the error-trapping mechanism can be turned on or off or otherwise modified while you are developing a project. You also need to know how the mechanism should be set before you distribute your application to others.

You set error handling options for your project on the General tab of the Options dialog box, illustrated here:



The Error Trapping options are explained below:

<i>Break on All Errors</i>	causes the program to enter Break mode and display an error message, regardless of whether you have written code to handle the error. This option essentially turns the error handling mechanism off and should be used for debugging only.
<i>Break in Class Module</i>	causes the program to enter Break mode and display an error message when an unhandled error occurs within a procedure of a class module such as a UserForm. If the Debug button is clicked within the error message window, the Code window will display with the line of code that generated the error highlighted. With this option, if Procedure A calls a class module's Procedure B and the unhandled error occurs in Procedure B, the offending line in <i>Procedure B</i> will be highlighted. Use this option for debugging only.
<i>Break on Unhandled Errors</i>	causes the program to enter Break mode and display an error message when any unhandled error occurs. If the Debug button is clicked within the error message window, the Code window will display with the line of code that generated the error highlighted. With this option, if Procedure A calls a class module's Procedure B and the unhandled error occurs in Procedure B, the line in <i>Procedure A</i> that called Procedure B will be the one highlighted, which may not be useful. This option can be useful during debugging, and this is the setting that should be selected before distributing your application.





Understanding VBA's Error Trapping Options

1. In the Visual Basic Editor, open the Code window for frmRunReports.
2. Add the following procedure to the end of the module:

```
Public Sub TestErrorSettings()  
    Dim i  
    i = i / 0    'Generates a divide by zero error  
End Sub
```

3. Open the Code window for modReports.
4. Modify the code in the LaunchReports procedure. Before the code that shows the form, add a statement that calls the test procedure:

```
frmRunReports.TestErrorSettings
```

5. Open the Tools menu and select *Options*. On the General tab, select *Break on Unhandled Errors*, if necessary, and then click OK.
6. Return to Excel and use the button on the Quick Access toolbar to launch the Run Reports form.
7. When the Error message displays, click Debug. Note that the procedure that displays does not contain the line of code that caused the error.
8. Click  Reset.
9. In the Options dialog box, change the Error Trapping option to *Break in Class Module*.
10. Return to Excel and launch the form again.
11. When the error message is displayed, click Debug. Note which procedure displays and that the highlighted line of code is the one that caused the error.
12. Click  Reset.
13. Click OK in the message box.
14. Add an error trap to the TestErrorSettings procedure. Before the line that causes the error, add the code statement: `On Error Resume Next`
15. In Excel, run the form and note that it doesn't cause an error. The *Break in Class Module* setting only breaks on unhandled errors and the `On Error Resume Next` statement handled the error.

continued on next page

Understanding VBA's Error Trapping options, continued

16. Click Cancel on the Run Reports form.
 17. Change the Error Trapping setting back to *Break on Unhandled Errors*.
 18. Display the Excel window and launch the form. No error is generated because the error was handled by the **On Error Resume Next** statement.
 19. Click Cancel.
 20. Return to the Visual Basic Editor. Change the Error Trapping setting to *Break on All Errors* and test it again. An error is generated regardless of the existing error handling.
 21. Click End and then click OK.
 22. In the Code window for frmRunReports, delete or comment out the TestErrorSettings procedure.
 23. Display the Code window for modReports and delete or comment out the line in the LaunchReports procedure that calls the TestErrorSettings procedure.
 24. Open the Options dialog box and set the Error Trapping option to *Break on Unhandled Errors*.
 25. Close any open Code windows as well as frmRunReports, if necessary.
-

Trapping Errors with the On Error Statement

Within a procedure, you enable an error trap with an **On Error** statement. If an error is generated after this statement is encountered, the error handler becomes active and passes control to wherever the **On Error** statement specifies.

The error-handling statement syntax is below:

On Error <branch instruction>

On Error GoTo ErrorHandler

On Error Resume Next

Once an On Error statement has trapped an error, the error needs to be handled in some fashion. There are three basic styles of handling errors in VBA:

Write an error handler

This is a routine that is branched to using the **On Error Goto <Line Label>** statement. It would include statements designed to address one or more types of errors for the procedure.

Ignore the error

If the error is inconsequential, use the **On Error Resume Next** statement to both trap and handle the error. The program continues on the next line of code.

Use in-line error handling

Use the **On Error Resume Next** statement to trap the error. Then enter code in the procedure to check for errors immediately following any statements that are expected to generate errors.

On Error Goto 0

The statement **On Error GoTo 0** disables error handling for that procedure at least until another On Error statement is encountered. This is helpful to use temporarily while debugging as an alternative to changing the Error Trapping setting to *Break On All Errors*, as it only affects the procedure in which the statement appears. The VBA error window with its Debug button will display when an error occurs in the given procedure. (You may have to set the Error Trapping style to *Break In Class Module* to obtain this behavior.) Once you have resolved the issue, you would remove the On Error Goto 0 statement.



Although it is rarely necessary, a procedure may have multiple On Error statements.

Error trapping is defined on a procedure-by-procedure basis; VBA does not provide the ability to specify a global error trap.

Understanding the Err Object

Visual Basic provides an Err object that you can use to examine information about an error that has just occurred. The Err object is built into Visual Basic and has a global scope.

The Err object has properties and methods that are useful for finding out information about the current error, clearing error information, and generating errors. The properties contain information about the error that just occurred in the current procedure.

Below is a list of the most commonly used properties associated with the Err object:

Property	Description	Data Type
Number	represents the identification number of the most recent error. The numbers represent different types of errors. This is the default property for the Err object.	Long
Description	wording that describes the error that just occurred and which corresponds to the error number.	String
Source	contains a name used to identify the component, module, and/or procedure that generated the error.	String

Below is an explanation of the two methods of the Err object:

Method	Description
Clear	resets all the Err object's properties to zero or zero-length strings (""). This method is invoked automatically when any of the following statements are encountered: On Error, Resume (any type), Exit Sub, Exit Function, and Exit Property.
Raise	generates a run-time error. This error can be a custom error that only has meaning within your application or it can specify the number of an error defined by VBA, Excel, or another component such as Word. The Raise method has arguments that allow specifying the values for each of the Err object's properties.



Using the Raise method to force an error can be helpful for testing your error-handling routines. For example, the following statement will generate a "Division by zero" error message: `Err.Raise 11`

Writing an Error-Handling Routine

The **On Error GoTo** statement is used to branch to a block of code within the same procedure that handles the errors that occur within the procedure. This block of code is referred to as an error-handling routine and is identified by a line label. The routine is almost always stored at the bottom of the procedure and is preceded with an **Exit** statement that prevents the routine from being executed unless an error has occurred. Common line labels used to identify an error-handling routine are “ErrorHandler” and “EH”. You can use one of these names or one of your choosing to identify all of your error-handling routines.

One of the benefits of using this style of handling errors is that all of your error-handling logic is located at the bottom of a procedure rather than being mixed in with the primary logic of the procedure. This usually makes it easier to write the procedure and certainly makes the procedure easier to read and understand at a later date.

The example below illustrates a simple error handling routine for a sub procedure:

```
Sub CalculateSomething(x as integer, y as integer)
    Dim z as double
    On Error GoTo ErrorHandler           ' error trap
    z = X^3-45 / y                        ' this might generate a divide-by-zero error
    z = z + (y*67)
    CalculateSomething = SomeFunction(z)  ' this function could return an error
    Exit Sub                             ' prevents error-handling code from running
                                         ' when no error occurs

ErrorHandler:                          ' line label for error routine
    If Err.Number = 11 Then
        MsgBox "Please supply a non-zero value for y."
    Else
        MsgBox "Error Number: " & Err.Number & vbCrLf & _
            "Error Description: " & Err.Description
    End If
End Sub
```



Line labels only have to be unique within the enclosing procedure.

Writing an Error-Handling Routine, continued:

Once execution has been passed to an error routine, there are several ways of specifying the code to be executed next:

Resume	Execution continues on the same line within the procedure that directly or indirectly caused the error.
Resume Next	Execution continues on the line within the procedure that follows the line that directly or indirectly caused the error.
Resume <Line Label>	Execution continues on a line identified by the line label. Typically this points to another routine within the procedure that is used to perform “clean-up” duties for the procedure, such as releasing variables and deleting temporary files.
End Sub/End Function	It is quite common to have the error handler address the issue and then exit the procedure normally by reaching the End Sub or End Function statement.
Exit Sub/Exit Function	The routine may exit the procedure using the appropriate Exit Sub or Exit Function commands.

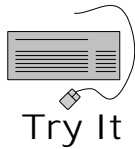
Understanding the Call Chain

Except for when the error trapping option is set to *Break on All Errors*, VBA will automatically begin searching for an error trap whenever an error occurs. It will first determine if an error trap is enabled within the procedure that generated the error. If none is found, the error will be passed up the *call chain*. This means the error will be passed to the procedure that called the current procedure and then to the procedure that called it and so on until either an error trap is found or there are no more procedures to check. If no error trap is found, the program crashes and an error message displays.

It's important to note what happens when a calling procedure's error-handling code is invoked for an error in a called procedure. If the calling procedure uses a **Resume**, **Resume Next**, or **Resume <line label>** statement within its error-handling routine, the resumption of execution applies to that procedure, not to the procedure that originally generated the error.




Although it is typically used for debugging, you may choose to use the **On Error Goto 0** statement permanently within a procedure if you would prefer that a particular error be handled by the calling procedure, rather than by the procedure containing the error.



Writing an Error-Handling Routine

In this exercise, we'll create an error by renaming the Reports workbook. We'll handle the error by displaying a message to the user.

1. Open Windows Explorer and navigate to the Data folder.
2. Rename the **Reports** workbook to `Reports2`.
3. In Excel, delete the Reports worksheet, if necessary, and then launch the Run Reports form.
4. Select *All* under Period and *Model* under Sales By; click Display.
5. Note the error number in the message box and click Debug. The FinishReport procedure displays with the line of code that caused the error highlighted.
6. Click  Reset.
7. Modify the FinishReport procedure to match the code shown below:

```
Sub FinishReport()  
    Dim wsSheet As Worksheet  
    On Error GoTo errorHandler  
    Workbooks.Open Filename:=ThisWorkbook.Path & "\Reports.xlsx"  
    ...  
    MakePivot  
    SetPivot  
    Exit Sub  
  
errorHandler:  
    Select Case Err.Number  
        Case 1004  
            MsgBox "The Reports workbook is not available."  
        Case Else  
            MsgBox "Error Number: " & Err.Number & vbCrLf & _  
                "Error Description: " & Err.Description  
    End Select  
End Sub
```

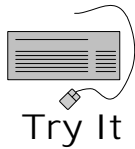
8. Compile and save your work.
 9. Switch to Excel and delete the Reports worksheet in **Fiscal Year Sales**.
 10. Launch the Run Reports UserForm.
 11. Select *All* under Period and *Model* under Sales By; click Display.
 12. Click OK in the message box.
 13. In the Windows Explorer, rename the **Reports2** workbook to `Reports`.
-

Working with Inline Error Handling


When you use inline error handling, you place the code to handle errors directly into the body of the procedure, rather than placing it at the end in one routine. To enable inline error handling, you must enter the **On Error Resume Next** statement into your procedure. Error handling code is then placed immediately after the particular line of code that is expected to cause an error. You may find this method easier to use in longer procedures where you anticipate two or more errors to occur. Both styles are equally valuable and may even be combined within a procedure.

The following code illustrates an inline error handler trapping for possible errors resulting from a file-opening operation:

```
Sub ProcFileOpen()  
    On Error Resume Next  
    Open "c:\data\sample1.txt" For Input As #1  
    Select Case Err  
        Case 0  
            ' This is expected, so check it first.  
        Case 53  
            MsgBox "File not found: c:\data\sample1.txt"  
        Case 55  
            MsgBox "File in use: c:\data\sample1.txt"  
        Case Else  
            MsgBox "Error Number: " & Err.Number & vbCrLf & _  
                "Error Description: " & Err.Description  
    End Select  
    Err.Clear      ' reset Err object  
End Sub
```



Working with Inline Error Handling

1. In Excel, launch the Run Reports form. (Do not delete the Reports worksheet first.)
2. Select *Month* under Period and select *Jul* from the combo box.
3. Make a selection under Sales By and click Display.
4. Note the error number and click Debug.
5. Note the highlighted line of code that caused the error and click  Reset.
6. After the variable declarations in the ConsolidateData procedure, add the following On Error statement:

```
On Error Resume Next
```

7. Immediately after the line of code that caused the error enter the code shown below to handle the error:

```
If Err.Number = 1004 Then
    ActiveSheet.Delete
    Sheets("Reports").Cells.Select
    Selection.Clear
    Err.Clear
End If
```

8. Save your work.
9. Switch to Excel and delete the empty generically-named worksheet that was created earlier.
10. Run the Run Reports form.
11. Select *Month* under Period and select *Jul* from the combo box.
12. Make a selection under Sales By and click Display.
13. Click Delete at both prompts.
14. View the Reports workbook to see that the PivotTable was generated correctly.
15. When you are finished, close the **Reports** and **Fiscal Year Sales** workbooks.
16. Close Excel.

Index

A

- Adding code to controls, 6-27
- AND, 5-3
- Arguments, 2-4
 - named versus positional, 3-14
 - optional, 2-8
 - passing multiple, 2-8
 - passing values to, 2-8
- As clause, 2-11
- Assignment statement, 4-5
- Auto Data Tips, 2-17, 8-12
- Auto List, 2-17
- Auto Quick Info, 2-17
- Auto Quick Info feature, 2-8

B

- Boolean expressions, 5-3, 5-4
- Branching, 5-2
- Break mode, 8-5
 - getting into from Run Mode, 8-11
- Breakpoints
 - setting, 8-6

C

- Call Chain, 9-9
- Call Stack, 8-8
- Calling sub and function procedures, 2-8
- Class, 3-8
- Class Module, 2-2
- Code
 - Commenting, 2-18
 - continuing on the next line, 2-18
 - editing guidelines, 2-18
 - indenting, 2-18
 - stepping through, 8-8
- Code Editor, 2-16
- Code Editor Options, 2-17
- Code window, 1-13, 2-16
 - navigating in, 2-19
 - printing from, 2-19
- Collections, 3-3
 - referencing objects in, 3-4
 - understanding, 3-4
- Combo box
 - populating, 6-25
- Combo box control, 6-16
- Command button control, 6-14
- Commenting, 2-18

- Comparison operators, 5-3
- Compiling
 - understanding, 4-6
- Concatenation, 4-8
- Constants, 3-8, 4-2, 4-19
 - declaring, 4-19
 - intrinsic, 4-20
- Control-of-Flow structures
 - described, 5-2
 - Do...Loop, 5-15
 - For Each...Next, 5-20
 - For...Next, 5-18
 - guidelines for use of, 5-21
 - If...End If, 5-5
 - Select...Case, 5-12
- Controls
 - adding code to, 6-27
 - appearance, 6-20
 - Combo box, 6-16
 - Command button, 6-14
 - Frame, 6-18
 - Label, 6-10
 - listed, 6-4
 - naming conventions for, 6-8
 - Option button, 6-19
 - populating, 6-25
 - setting properties in the Properties Window, 6-9
 - Text box, 6-12
 - understanding, 6-7
- Creating standard modules, 2-3

D

- Data types
 - listed, 4-9
 - understanding, 4-8
- Debug toolbar, 8-4
- Debug.Print statement, 2-9
- Debugging
 - described, 8-2
 - error types, 8-2
- Decision structures
 - guidelines for use of, 5-21
- Determining the value of expressions, 8-12
- Developer Tab, 1-3
- Do Until, 5-15
- Do While, 5-15
- Do...Loop, 5-15
- Documenting code, 2-18

E

- ElseIf, 5-6
- enabling macros, 1-22
- endless loop, 8-11
- Enum, 3-8
- Err object, 9-7
- Error Handling
 - inline, 9-11
 - understanding, 9-2
- Error Trapping options
 - setting, 9-3
- Error types
 - understanding, 8-2
- Error-handling routine, 9-9
 - creating, 9-8
- Errors
 - logic errors, 8-2
 - minimizing, 8-3
 - run-time errors, 8-2
 - syntax errors, 8-2
- Event Procedures
 - creating, 3-17
- Events, 3-8
 - defined, 3-2
- Excel Object Model, 3-3
- Excel Objects
 - referencing, 3-3
- Expressions, 4-2
 - determining the value of while debugging, 8-12

F

- For Each...Next, 5-20
- For...Next, 5-18
- Form
 - Adding to a project, 6-3
 - Launching in Code, 6-31
 - setting tab order, 6-23
- Forms, 6-2
- Frame control, 6-18
- Function procedures
 - calling, 2-8
 - creating, 2-11
 - described, 2-4
- Functions
 - evaluation, 4-14
 - intrinsic, 4-14
 - string, 4-15
 - type conversion, 4-14

G

- General Declarations, 2-2

H

- Help
 - using in Visual Basic, 1-19

I

- If...End If, 5-5
- If...Then...Else, 5-5
- Immediate Window, 8-12
 - calling procedures from, 2-9
 - displaying, 2-9
 - evaluating expressions in, 2-9
- Inline Error Handling, 9-11
- Input Box
 - described, 4-26
- InputBox function, 4-26
- Intrinsic Functions, 4-14

K

- Keyword
 - Me, 6-6
 - private, 2-5
 - public, 2-5
 - set, 4-28
 - Stop, 8-7
 - With...End With, 3-13

L

- Label control, 6-10
- line label, 9-8
- Literal, 4-2
- Locals Window, 8-12
- logic errors, 8-2
- Logical operators, 5-3
- Looping, 5-2
- Looping structures
 - guidelines for use of, 5-21

M

- Macro
 - Editing, 1-10
 - recording, 1-4
 - running, 1-8
- Macro Security, 1-22
- Macro-Enabled Workbook, 1-6
- Message Box
 - described, 4-21
- Methods, 3-8
 - calling, 3-14
 - defined, 3-2
 - described, 3-14
- Module, 1-10
 - defined, 2-2
- Module-level Variables, 4-11
- Modules, 3-8
 - creating standard, 2-3
- MsgBox function, 4-21

N

NOT, 5-3
Numeric Data Types, 4-8
Numeric expression, 4-2

O

Object Browser, 3-7, 3-8
Object List, 2-16
Object variables, 4-28
 declaring, 4-28
Objects
 global, 3-3
 understanding, 3-2
On Error statements, 9-6
Operator, 4-2
Option button control, 6-19
Option Explicit statement, 2-17, 4-6
OR, 5-3

P

Personal.xls workbook, 1-4
PivotCache object, 7-6
PivotFields collection, 7-8
PivotTable object, 7-6
PivotTable
 Add PivotFields to a, 7-8
 creating, 7-3
 understanding, 7-2
Populating a control, 6-25
Procedure List, 2-16
Procedure-level Variables, 4-11
Procedures, 1-10, 2-2
 calling, 2-8
 described, 2-4
 Naming, 2-15
Project Explorer, 1-13, 1-16
Properties, 3-8
 assigning values to, 3-11
 defined, 3-2
 described, 3-11
Properties Window, 1-13, 1-17
Public Variables, 4-11

Q

Quick Access toolbar
 assigning macro to, 7-16
Quick Watch window, 8-12

R

Resume statements, 9-9
Run to Cursor tool, 8-8
Running macros, 1-8
Run-time error, 8-11
run-time errors, 8-2

S

Security settings for macros, 1-22
Select...Case, 5-12
Set Next Statement, 8-8
Standard Module, 2-2
Statements, 4-4
Step Into tool, 8-8
Step Out tool, 8-8
Step Over tool, 8-8
Stepping through Code, 8-8
String Data Types, 4-8
String expression, 4-2
Sub procedures
 calling, 2-8
 creating, 2-5
 described, 2-4
syntax errors, 8-2

T

Text box control, 6-12
Toolbox, 6-4
Totaling a column, 4-29
Trapping Errors, 9-6
Trust Center
 about, 1-22
 opening, 1-23
Trusted Documents, 1-22

U

User-defined functions, 2-11
UserForms, 6-2
 Adding to a project, 6-3
 Events, 6-6
 Methods, 6-6
 Properties, 6-5

V

Variable Scope
 about, 4-11
Variables, 4-2
 declaring explicitly, 4-6
 implicit versus explicit declaration, 4-6
Variables, 4-2
 assigning values to, 4-5
 declaring, 4-5, 4-6
 defined, 4-5, 4-6
 Global, 4-11
 local, 4-11
 module-level, 4-11
 naming conventions, 4-5
 object, 4-28
Variant data type, 4-8
vbBack, 4-20
vbCr, 4-20
vbCrLf, 4-20

- vbLf, 4-20
- vbNullString, 4-20
- vbTab, 4-20
- Visual Basic Editor, 1-10, 1-14, 1-15, 1-16, 1-17
 - about, 1-13
 - closing, 1-21
 - displaying, 1-11
- Visual Basic for Applications
 - Introduction, 1-2

W

- Watch Window, 8-12
- With Statement
 - using, 3-13

ONLCWEXP16

