

Source:

1. Financial Analysis and Modeling: Using Excel and VBA, Chandan Sengupta, 2nd edition, published by Wiley. 2009
2. Excel 2019 Power Programming with VBA, Michael Alexander, Dick Kusleika, published by Wiley. 2019

Excel version: Microsoft Excel 365 for Windows and Mac

CHAPTER1. INTRODUCING VISUAL BASIC FOR APPLICATIONS

Visual Basic for Applications (VBA) is a programming language that is built into Excel and other Microsoft Office applications.

In both Excel and VBA, you perform actions. These actions are mostly calculations with numbers, but they can also be creating charts, formatting worksheets, and so forth. In VBA, you perform actions by executing VBA code, which is a series of VBA instructions or statements.

Unlike in Excel, the actions do not take place as you write the instructions. After you write them, you have to execute them to perform the actions.

Why Should You Learn VBA?

Benefits of learning program:

1. **Easy to recognize and replicate**: In many complex Excel models, the formulas in some cells can become very long and extend to several lines. What's more, since many of these formulas are designed to be copied to many different cells, they generally have to use many cell addresses with relative or mixed references and cannot use recognizable names for those cells.
2. **Easy to detect errors**: In large Excel models, the same formula may be copied into dozens of cells. So every time such a formula is changed, it has to be recopied into many cells with the possibility of your missing some of them. Mistakes are very difficult to detect. In VBA, the same formula usually appears only once. Updating formulas and making sure that they do not get inadvertently damaged are much easier in VBA than they are in Excel.

3. **Easy to share and reuse:** VBA programs provide step-by-step instructions in the logical sequence in which the computer is supposed to execute a model; it is generally easy to follow VBA models even if they have limited documentation.

Benefits of choosing VBA: Excel VBA is designed to work seamlessly with Excel. So you can enjoy the power of a programming language and you can also take advantage of most of the attractive features of Excel.

1. VBA is already built into Excel.
2. VBA offers dozens of built-in functions, including many financial functions, to simplify the work of building models. These functions work like the Excel functions, and you can also use most Excel functions in VBA.
3. Microsoft offers a version of VBA with each of its Office applications, such as Word and Access. All of these are essentially the same programming language, and once you learn Excel VBA, it will take little additional effort to learn the other versions.

1. Getting a Head Start with the Macro Recorder

A macro is essentially VBA code that you can call to execute any number of actions. In Excel, macros can be written or recorded.

Excel programming terminology can be a bit confusing. VBA programs are also called **procedures**, **codes**, and **macros**. The names can be used interchangeably. In VBA you will write only two kinds of programs: **Sub procedures** and **Function procedures**. In this context, that is, Sub procedures are not called Sub codes, Sub macros, and so on. A recorded macro is technically no different from a VBA procedure that you create manually. Many Excel users call any VBA procedure a macro. However, when most people think of macros, they think of recorded macros.

Recording a macro is like programming a phone number into your smartphone. First you manually enter and save a number. Then when you want, you can redial the number with the touch of a button. Just like on a smartphone, you can record your actions in Excel while you perform them. While you record, Excel gets busy in the background, translating and storing your keystrokes and mouse clicks to VBA code. After a macro is recorded, you can play back those actions any time you want.

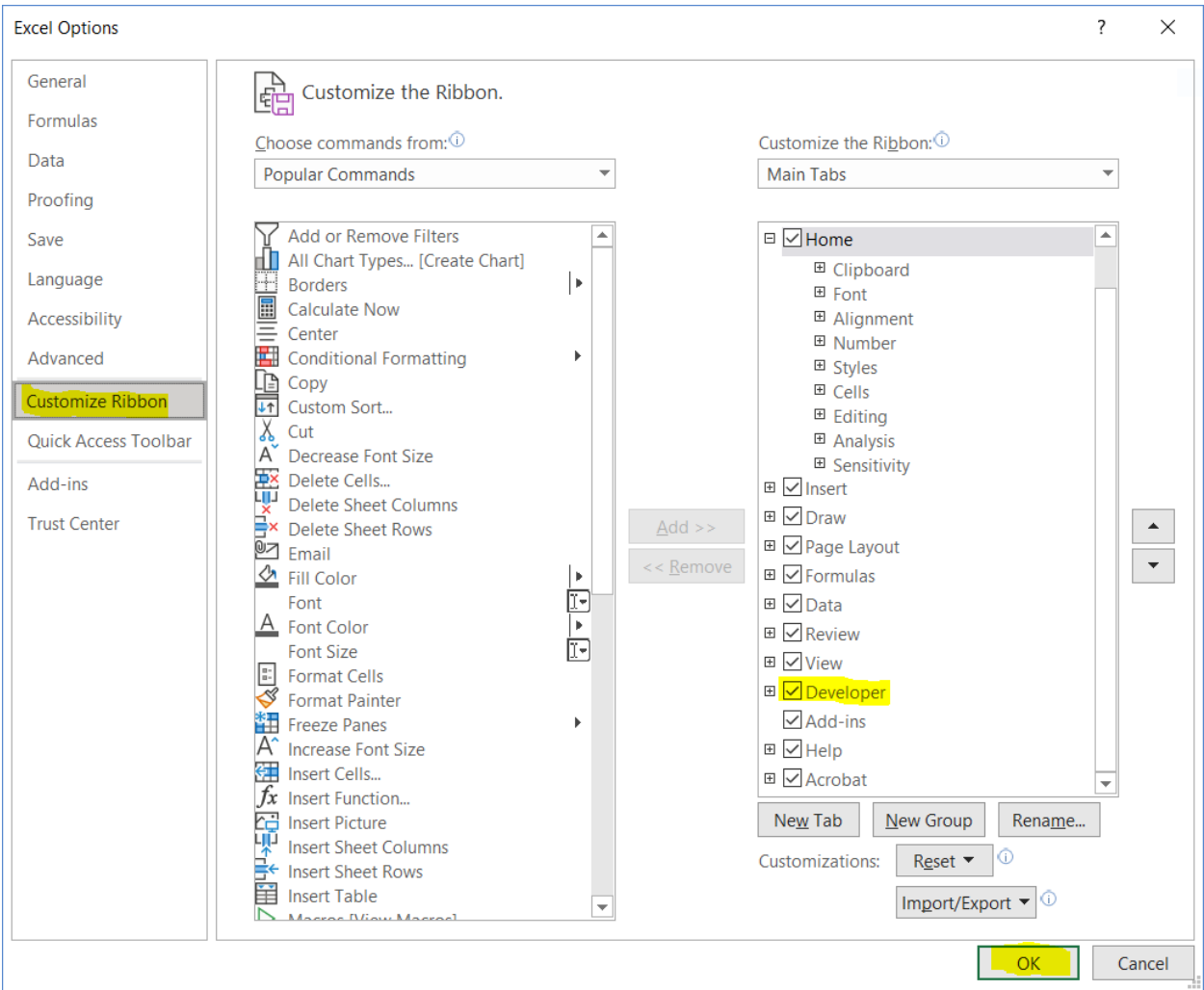
The best way to become familiar with VBA, without question, is simply to turn on the macro recorder and record some of the actions that you perform in Excel. This approach is a quick way to learn the relevant VBA syntax for a task.

In this section, you'll explore macros and learn how you can use the macro recorder to start familiarizing yourself with VBA.

1.1 Creating Your First Macro

To start recording your first macro, you first need to find the macro recorder, which is on the **Developer** tab. Unfortunately, Excel comes out of the box with the Developer tab hidden—you may not see it on your version of Excel at first. To display this tab, follow these steps:

1. Choose File ⇄ Excel Options.
2. In the Excel Options dialog box, select Customize Ribbon.
3. In the list box on the right, place a check mark next to Developer.
4. Click OK to return to Excel.

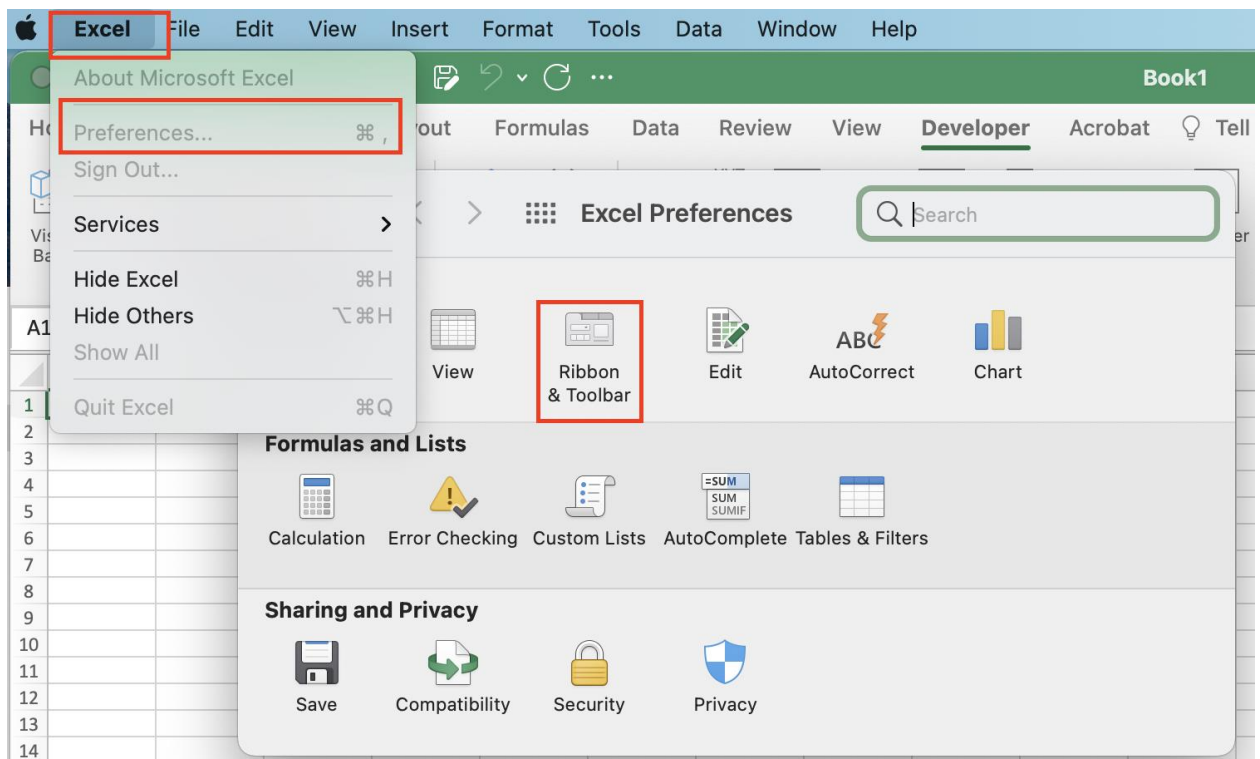


Now that you have the Developer tab showing in the Excel Ribbon, you can start up the macro recorder in the following ways:

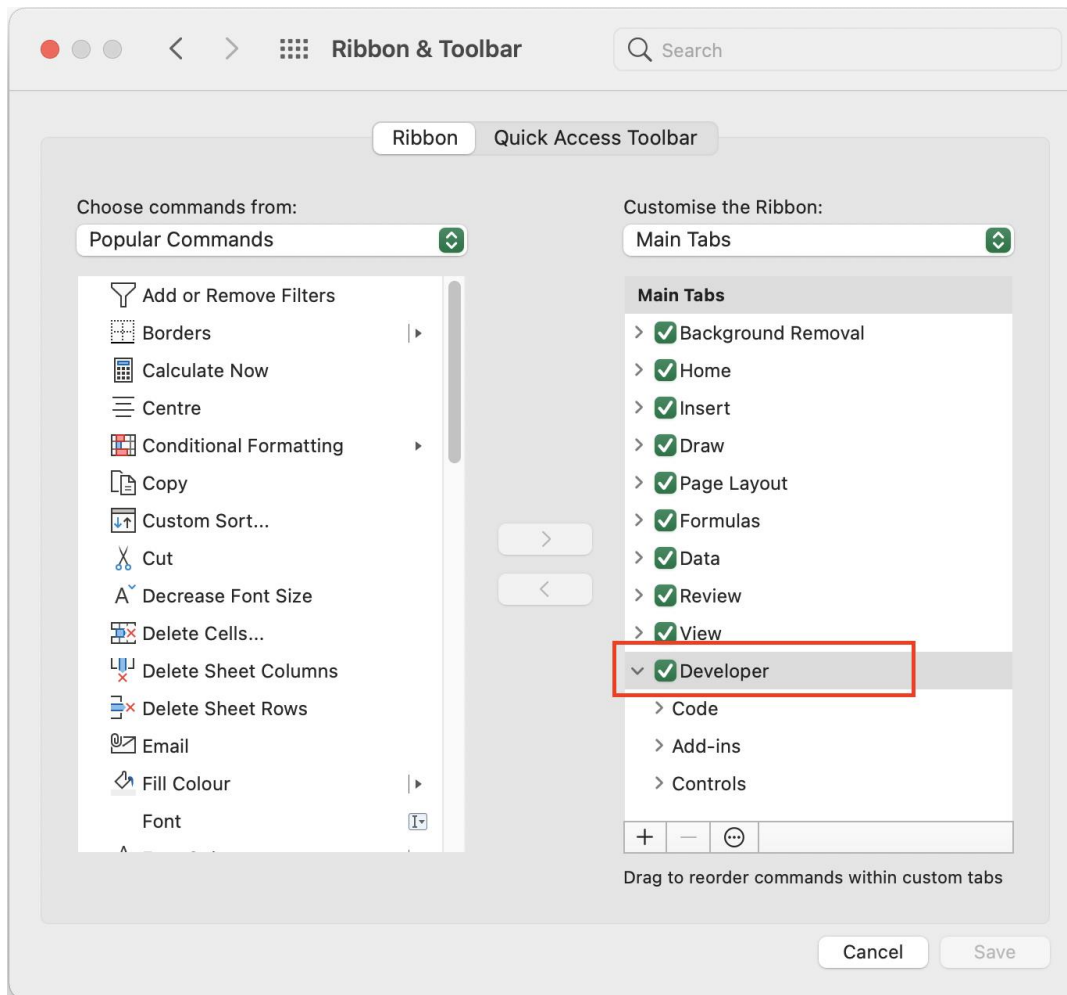
1. Go to Developer tab ⇒ Code ⇒ Record Macro.
2. Go to View tab ⇒ ⇒ Macros ⇒ Record Macro.

In the Excel for Mac, follow these steps to display the **Developer** tab in the Ribbon:

1. In the menu bar, go to Excel ⇒ Preferences.
2. In the Excel Preferences dialog box, select Ribbon & Toolbar.



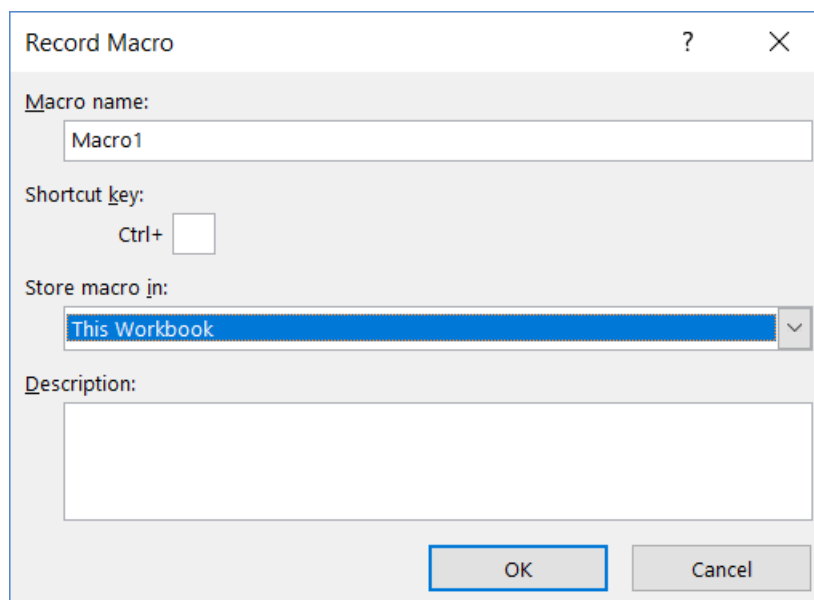
3. In the list box on the right, place a check mark next to Developer.
4. Click Save to return to Excel.



Then you can **start up the macro recorder** in three ways.

3. Go to Tools menu ⇒ Macro ⇒ Record New Macro.
4. Go to View tab ⇒ Record Macro.
5. Go to Developer ⇒ Code ⇒ Record Macro.

This activates the Record Macro dialog box, as shown below.



Here are the four parts of the Record Macro dialog box:

- **Macro Name:** This should be self-explanatory. Excel gives a default name to your macro, such as `Macro1`, but you should give your macro a name more descriptive. For example, you might name a macro that formats a generic table as `FormatTable`.
- **Shortcut Key:** Every macro needs an event, or something to happen, for it to run. This event can be a button press, a workbook opening, or, in this case, a keystroke combination. When you assign a shortcut key to your macro, entering that combination of keys triggers your macro to run. This is an optional field.
- **Store Macro In:** This Workbook is the default option. Storing your macro in **This Workbook** simply means that the macro is stored along with the active Excel file. The next time you open that workbook, the macro is available to run.
- **Description:** This is an optional field, but it can come in handy if you have numerous macros in a workbook or if you need to give a user a more detailed description about what the macro does. The description is also useful for distinguishing one macro from another when you have multiple workbooks open or you have macros stored in the **Personal Macro Workbook**.

Example 1: Open the “Lec1_MacroRecording.xlsm” file and go to worksheet Record1. Follow these steps to create a simple macro that enters your name into a worksheet cell:

1. Select cell B2.
2. Start recording the macro. With the Record Macro dialog box open, enter a new single-word name for the macro to replace the default `Macro1` name. A good name for this example is `MyName`.
3. Assign the shortcut key `Ctrl+Shift+N` to this macro by entering uppercase `N` in the edit box labeled `Shortcut Key`.
4. Click `OK` to close the Record Macro dialog box and begin recording your actions.
5. Type your name into the active cell and press `Enter`.
6. Choose `Developer ⇨ Code ⇨ Stop Recording`.

1.2 Examining Your Macro

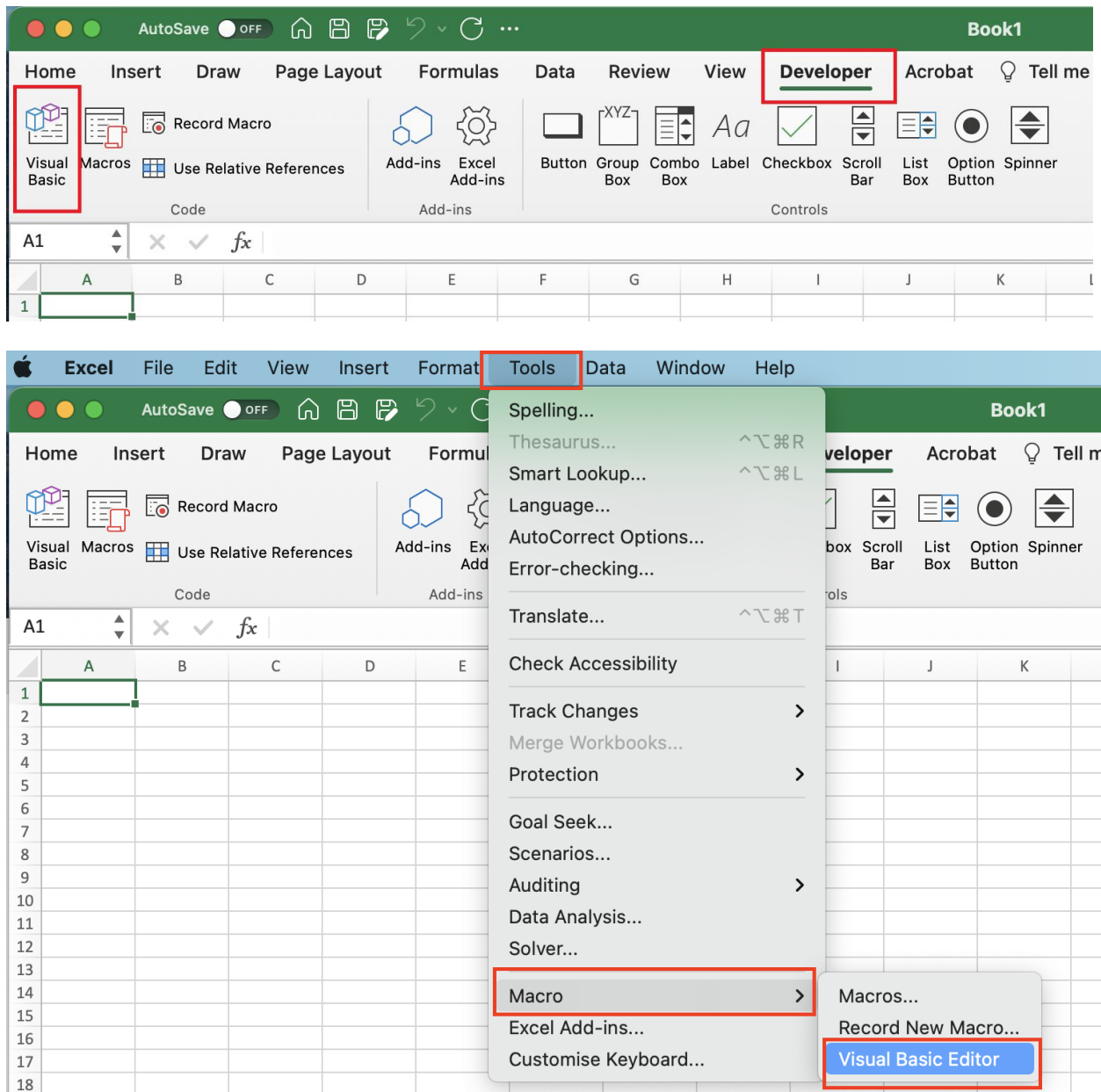
Excel stored your newly recorded macro in a new module that it created automatically and named `Module1`. To view the code in this module, you must activate the **Visual Basic Editor (VBE)**, which is where you will do most of your work with VBA. The VBE is part of Excel, but it does not open automatically when you start Excel.

You can activate the VBE window in either of two ways:

1. Press **Alt+F11**.
2. Choose `Developer ⇨ Code ⇨ Visual Basic`.

In Excel for Mac, you can start VBE window in three ways:

1. Press **option+fn+F11**
2. Go to `Developer ⇨ Code ⇨ Visual Basic`.
3. Go to menu bar and choose `Tools ⇨ Macro ⇨ Visual Basic Editor`

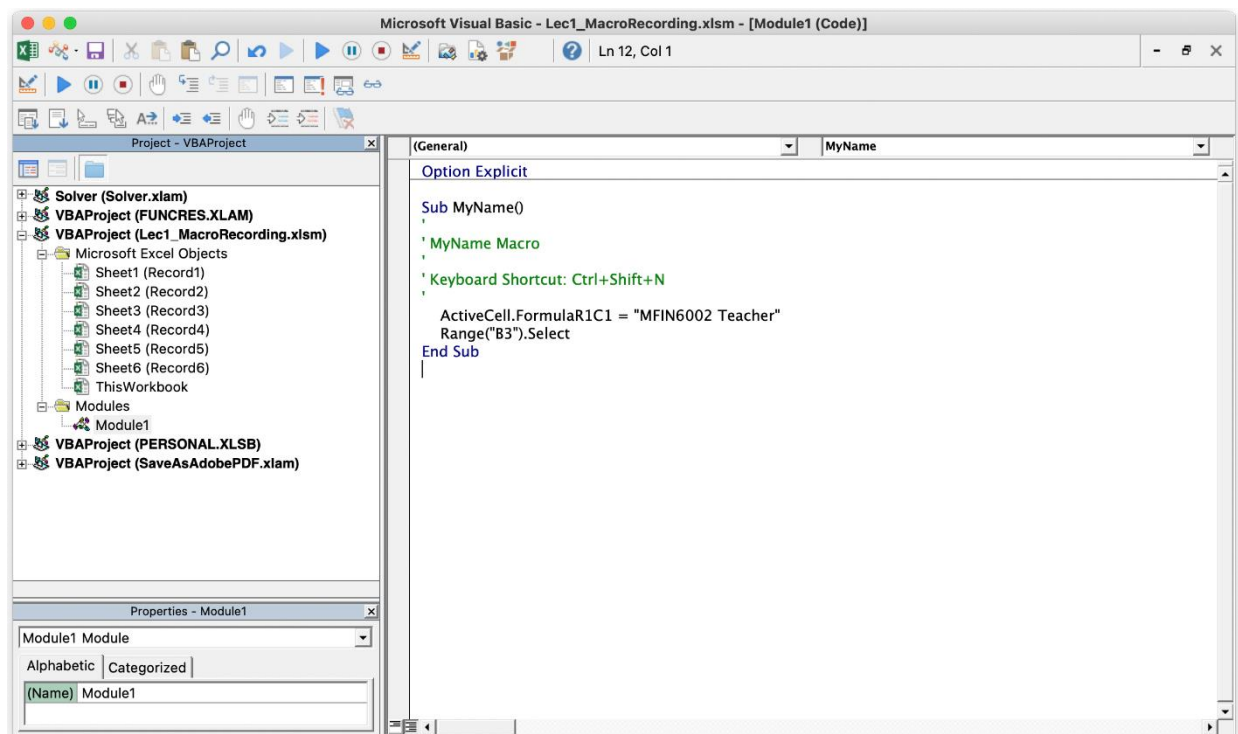


In the VBE, the **Project window** displays a list of all open workbooks and add-ins. This list is displayed as a tree diagram on the left of the screen, which you can expand or collapse. The code that you recorded previously is stored in `Module1` in the current workbook. When you double-click `Module1`, the code in the module appears in the Code window.

The macro should look something like this:

```
Sub MyName()  
    ' MyName Macro  
    ' Keyboard Shortcut: Ctrl+Shift+N  
    ActiveCell.FormulaR1C1 = "MFIN6002 Teacher"  
    Range("B3").Select  
End Sub
```

The VBE window in Excel for Mac should look something like this:



The macro recorded is a **Sub procedure** that is named `MyName`. The statements tell Excel what to do when the macro is executed. Notice that Excel inserted some comments at the top of the procedure. These comments are some of the information that appeared in the Record Macro dialog box. These comment lines (which begin with an apostrophe) aren't necessary and deleting them has no effect on how the macro runs. If you ignore the comments, you'll see that this procedure has two VBA statements.

```
ActiveCell.FormulaR1C1 = "MFIN6002 Teacher"
```

This statement causes the name you typed while recording to be inserted into the active cell.

```
Range("B3").Select
```

This statement activates cell B3. The macro recording results are saved in Excel file "Lec1_MacroRecording_Result.xlsm".

1.3 Testing Your Macro

Notice that you selected your target cell before you started recording your macro. This step is important. If you select a cell while the macro recorder is turned on, the actual cell that you selected will be recorded into the macro. In such a case, the macro would always format that cell.

You also delete `Range("B3").Select`. Then this macro is a general-purpose macro.

When Excel is active, activate a worksheet. (It can be in the workbook that contains the VBA module or in any other workbook.) Select a cell and press Ctrl+Shift+N. The macro immediately enters your name into the cell.

1.4 Editing Your Macro

After you record a macro, you can make changes to it. For example, assume that you want your name to be bold. You could re-record the macro, but this modification is simple, so editing the code is more efficient. Press Alt+F11 to activate the VB Editor window. Then activate `Module1` and insert `ActiveCell.Font.Bold = True`, as demonstrated in the following sample code:

```
ActiveCell.Font.Bold = True
```

The edited macro appears as follows:

```
Sub MyName1()  
    ActiveCell.Font.Bold = True  
    ActiveCell.FormulaR1C1 = "MFIN6002 Teacher"  
End Sub
```

Test this new macro, and you'll see that it performs as it should.

1.5 Why Do We Use Macro Recorder?

The Macro Recorder is useful for recording and then automatically repeating a series of tasks—especially long ones—that you perform often. It is particularly useful for recording formatting because writing codes for formatting can be tedious. Typically, you record the formatting, and then cut and paste the relevant part of the recorded code into other codes you are writing. The Macro Recorder can also produce excellent code and save a lot of work when what you are doing involves using a lot of menu commands (for example, creating a chart).

Another important use is to learn how to write code. If it is something you can do in Excel, you can always turn on the Macro Recorder and record the action.

1.6 Limitations and Potential Problems

Since the Macro Recorder can record only your actions, it cannot generate codes such as looping and `If` statements. Therefore, what it can do is limited.

Macro Recorder often generates large and inefficient codes with lots of superfluous statements. But once you are familiar with VBA, you will be able to edit the generated code to what you really need and then use the macro by itself or cut and paste part or all of it into your other codes.

Caution: When you use the recorded macro in another worksheet, you may not be able to repeat everything. It happens when the macro contains the name of specific objects, such as “Sheet 2”, “Chart 20”. When you switch to another worksheet and plot a new chart, these names do not exist. So, you need to modify the name of these objects to make them independent of the worksheet you use.

1.7 Absolute and Relative Reference

You recorded the test macro using absolute reference, meaning that whenever you run it, it will perform the recorded actions exactly in the same locations irrespective of which cell is active immediately before you run the macro.

If you want the macro’s actions to take place in **relative locations**, you can use relative reference during recording. To change to relative locations, you can click the **Use Relative References** under Macros before or during recording the macro.

1.7.1 Recording Macros with Absolute References

Example 2: Open “Lec1_MacroRecording.xlsm” and go to worksheet Record2.

1. Before recording, make sure that cell A1 is selected.
2. Select Record Macro from the Developer tab.
3. Name the macro `AddTotal`.
4. Choose This Workbook in the Store Macro In drop-down.
5. Click OK to start recording.
6. At this point, Excel is recording your actions. While Excel is recording, perform the following steps:
7. Select cell A16, and type Total in the cell.

8. Select the first empty cell in Column D (D16), type = COUNTA(D2:D15), and then press Enter. This gives a count of branch numbers at the bottom of column D. The COUNTA function is used to catch any branch numbers stored as text.
9. Click Stop Recording on the Developer tab to stop recording the macro.

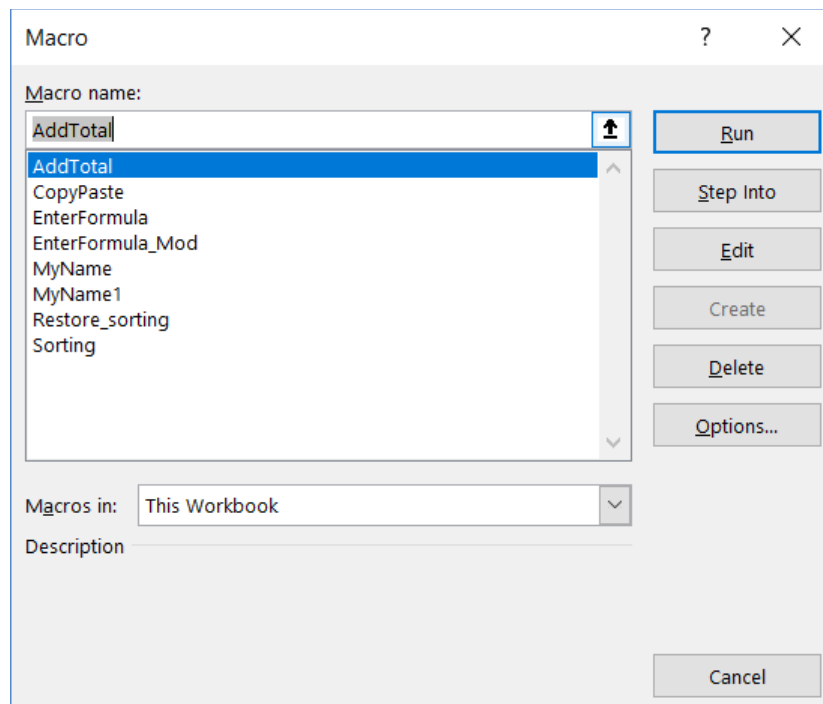
To see your macro in action, delete the total row that you just added and run the recorded macro.

1.7.2 Running Macros

If you want to run the recorded macro **from Excel spreadsheet interface**, you open the **Macro dialog box**, which by default, list the macros available in all open Excel workbooks. You can limit the list to only those macros contained in the active workbook by changing the Macros In setting to This Workbook.

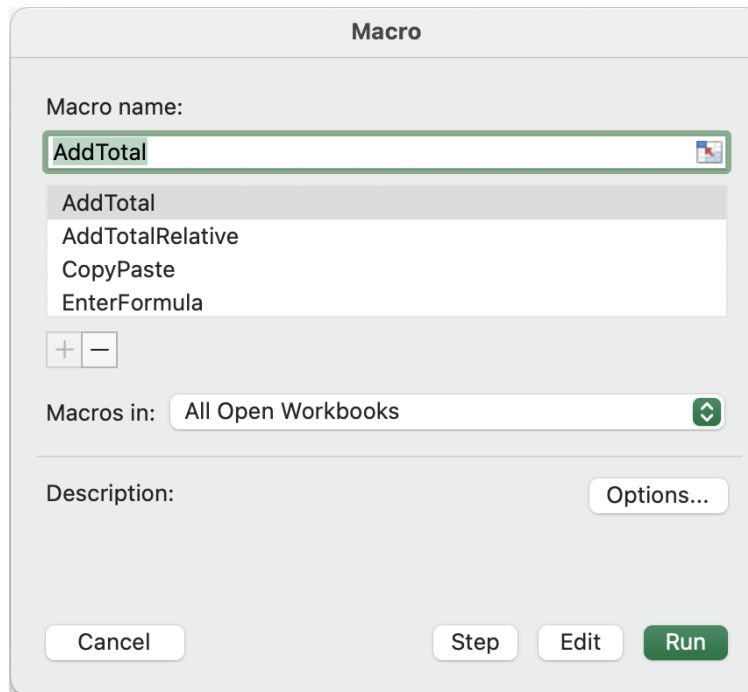
You can open the Macros dialog box by the following three ways:

1. Press Alt+F8
2. Go to View ⇒ Macros ⇒ View Macros
3. Go to Developer ⇒ Macros



In Excel for Mac, you can open the Macro dialog box by the following three ways:

1. Go to Tools menu ⇒ Macro ⇒ Macros
2. Go to View ⇒ View Macros
3. Go to Developer ⇒ Macros



In this Macro dialog, find and select the `AddTotal` macro that you just recorded, and click the **Run** button.

In Macro dialog box, if you select the `AddTotal` macro and click the **Edit** button, VBE window is open to show you the code that was written when you recorded your macro. The code that you recorded previously is stored in `Module2` in the current workbook.

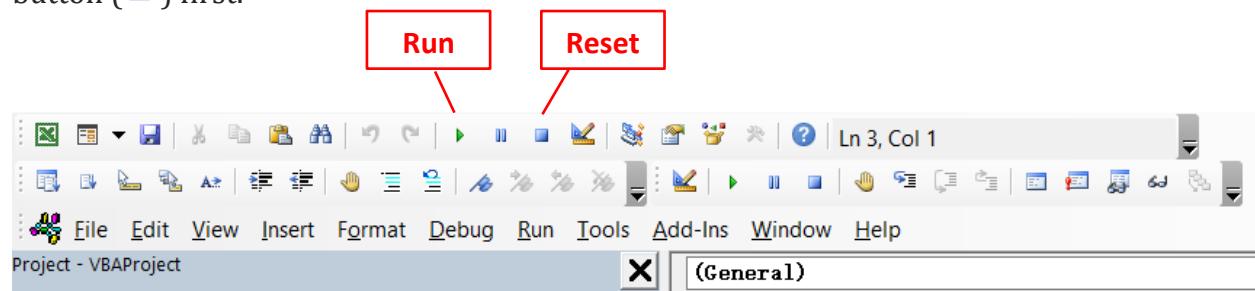
```
Sub AddTotal ()
    Range("A16").Select
    ActiveCell.FormulaR1C1 = "Total"
    Range("D16").Select
    ActiveCell.FormulaR1C1 = "=COUNTA(R[-14]C:R[-1]C)"
    Range("D17").Select
End Sub
```

If you want to run the macro **in the VBE window**, put your cursor anywhere inside the `AddTotal` code and use one of the following ways:

1. Press **F5**

2. Click the Run Sub/UserForm button (▶) on the toolbar
3. Go to Run menu and click Run Sub/UserForm.

Note: If for any reason the execution is halt (for instant, an error occurs when running the code), you should find that one of the statements is highlighted. If you want to modify the code, rerun the code or take actions in Excel spreadsheet interface, you have to click Reset button (■) first.



If all goes well, the macro plays back your actions perfectly and gives your table a total. However, you can't make the `AddTotal` macro work on the second table. Why? Because you recorded it as an absolute macro.

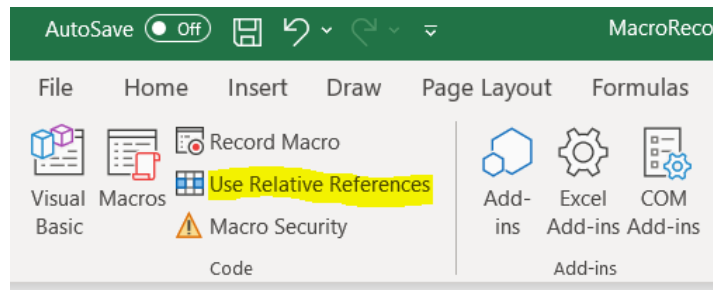
Pay particular attention to line 2, 4 and 6 of the code. When you asked Excel to select cell range A16 and then D16, those cells are exactly what it selected. Because the macro was recorded in absolute reference mode, Excel interpreted your range selection as absolute. In the next section, you will examine what the same macro looks like when recorded in relative reference mode.

1.7.3 Recording Macros with Relative References

In the context of Excel macros, relative means relative to the currently active cell. Thus, you should use caution with your active cell choice—both when you record the relative reference macro and when you run it.

First, make sure that the "Lec1_MacroRecording.xlsm" file is open and go to worksheet `Record2`. Use the following steps to record a relative reference macro:

1. Select the Use Relative References toggle button from the Developer tab, as shown in the figure below.



2. Before recording, make sure that cell A1 is selected.
3. Select Record Macro from the Developer tab.
4. Name the macro AddTotalRelative.
5. Choose This Workbook in the Store Macro In drop-down.
6. Click OK to start recording.
7. Select cell A16 and type Total in the cell.
8. Select the first empty cell in Column D (D16), type = COUNTA(D2:D15), and then press Enter.
9. Click Stop Recording on the Developer tab to stop recording the macro.

At this point, you have recorded two macros. Take a moment to examine the code for your newly created macro. Select Macros from the Developer tab to open the Macro dialog box. Here, choose the AddTotalRelative macro and click Edit. Your code looks like the following (ignoring comments):

```
Sub AddTotalRelative()
    ActiveCell.Offset(15, 0).Range("A1").Select
    ActiveCell.FormulaR1C1 = "Total"
    ActiveCell.Offset(0, 3).Range("A1").Select
    ActiveCell.FormulaR1C1 = "=COUNTA(R[-14]C:R[-1]C)"
    ActiveCell.Offset(1, 0).Range("A1").Select
End Sub
```

Notice that there are no references to any specific cell ranges at all (other than the starting point "A1"). Let's take a moment to look at what the relevant parts of this VBA code really mean.

In line 2, Excel uses the **Offset property** of the active cell. This property tells the cursor to move a certain number of cells up or down and a certain number of cells left or right.

The `Offset` property code tells Excel to move 15 rows down and 0 columns across from the active cell (in this case, A1). There's no need for Excel to select a cell explicitly, as it did when recording an absolute reference macro.

To see this macro in action, delete the total row and do the following:

1. Select cell A1.
2. Select Macros from the Developer tab.
3. Find and select the `AddTotalRelative` macro.
4. Click the Run button.
5. Now select cell F1.
6. Select Macros from the Developer tab.
7. Find and select the `AddTotalRelative` macro.
8. Click the Run button.

Notice that this macro, unlike your previous macro, works on both sets of data. Because the macro applies the totals relative to the currently active cell, the totals are applied correctly.

For this macro to work, you simply need to ensure that

- You've selected the correct starting cell before running the macro.
- The block of data has the same number of rows and columns as the data on which you recorded the macro.

Ideally, this simple example has given you a firm grasp of macro recording of both absolute and relative references.

1.8 Other Macro Recording Concepts

Next are some of the other important concepts you'll need to keep in mind when writing or recording macros.

1.8.1 Excel File Extension

By default, Excel workbooks are given the standard file extension `.xlsx`. Be aware that files with the .xlsx extension cannot contain macros. If your workbook contains macros and then you save that workbook as an `.xlsx` file, all VBA code is removed automatically. Luckily, Excel will warn you that your macro content will be removed when saving a workbook with macros as an `.xlsx` file.

If you want to retain the macros, you must save your file as an **Excel Macro-Enabled workbook**. This gives your file an **`.xlsm` extension**.

Alternatively, you can save your workbook as an Excel 97-2003 Workbook (with the `.xls` extension). The `.xls` file type can contain macros, but it doesn't support some of the modern features of Excel such as conditional formatting icons and pivot table slicers. You

would typically use this file type only if there is a specific reason, such as that you need to have your workbook interact with an add-in that works only with `.xls` files.

1.8.2 Storing Macros in Your Personal Macro Workbook

Most user-created macros are designed for use in a specific workbook, but you may want to use some macros in all your work. The Personal Macro Workbook is a hidden workbook named `Personal.xlsb`, which is created automatically when you save the first macro in it. It is stored in the XLSTART folder and loaded automatically whenever you start Excel; the macros in it are accessible from all workbooks. If you record any general-purpose macros that you may use often with different workbooks, you should consider saving it in this workbook.

To record the macro in your Personal Macro Workbook, select the Personal Macro Workbook option in the Record Macro dialog box before you start recording. This option is in the Store Macro In drop-down.

When you want to exit, Excel asks whether you want to save changes to the Personal Macro Workbook.

1.8.3 Assigning a Macro to a Button and Other Form Controls

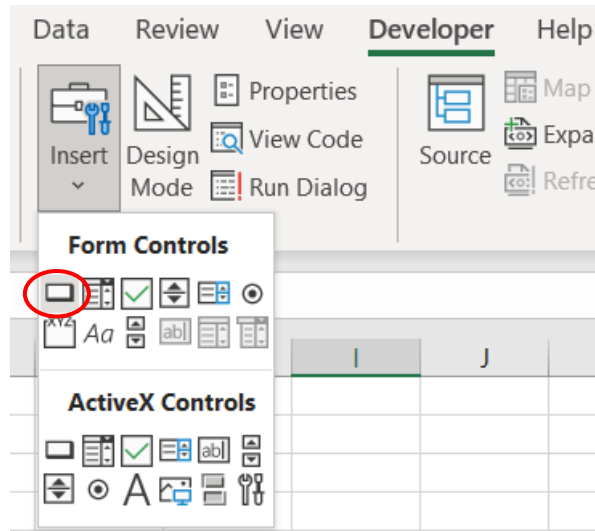
When you create macros, you may want to have a clear and easy way to run each macro. A basic button can provide a simple but effective user interface. Excel offers a set of **form controls** designed specifically for creating user interfaces directly on spreadsheets. There are several different types of form controls, from buttons (the most commonly used control) to scrollbars.

The idea behind using a form control is simple. You place a form control on a spreadsheet and then assign a macro to it—that is, a macro you’ve already recorded or written. When a macro is assigned to the control, that macro is executed or played when the control is clicked.

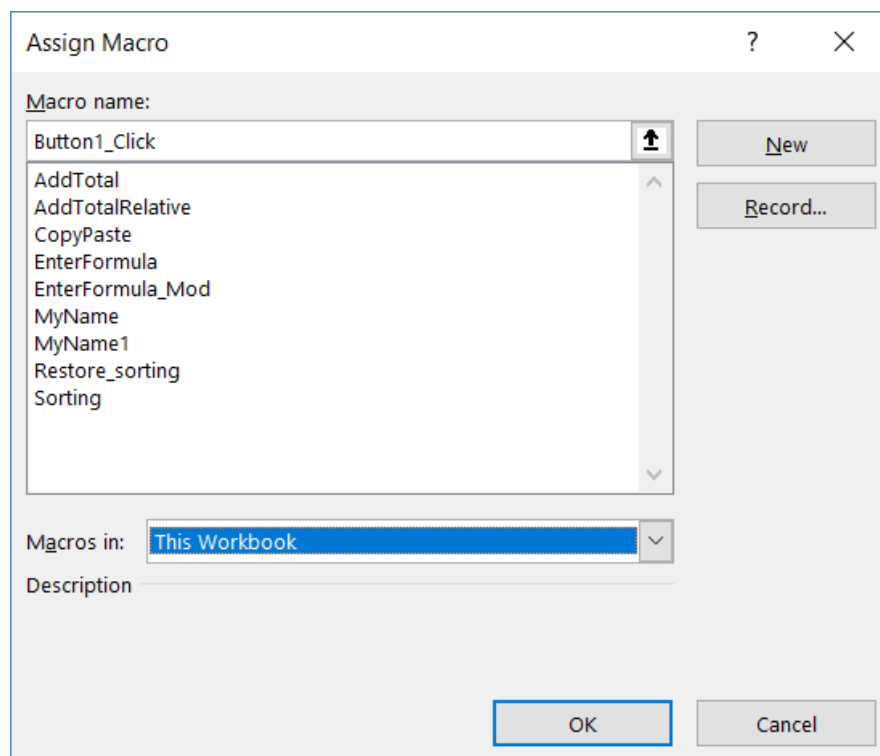
Take a moment to create a button for the `AddTotalRelative` macro you created earlier.

Here’s how:

1. Click the Insert button on the Developer tab.



2. Select the Button control from the drop-down list that appears.
3. Click the location where you want to place your button.
4. When you drop the button control onto your spreadsheet, the Assign Macro dialog box, as shown below, activates and asks you to assign a macro to this button.



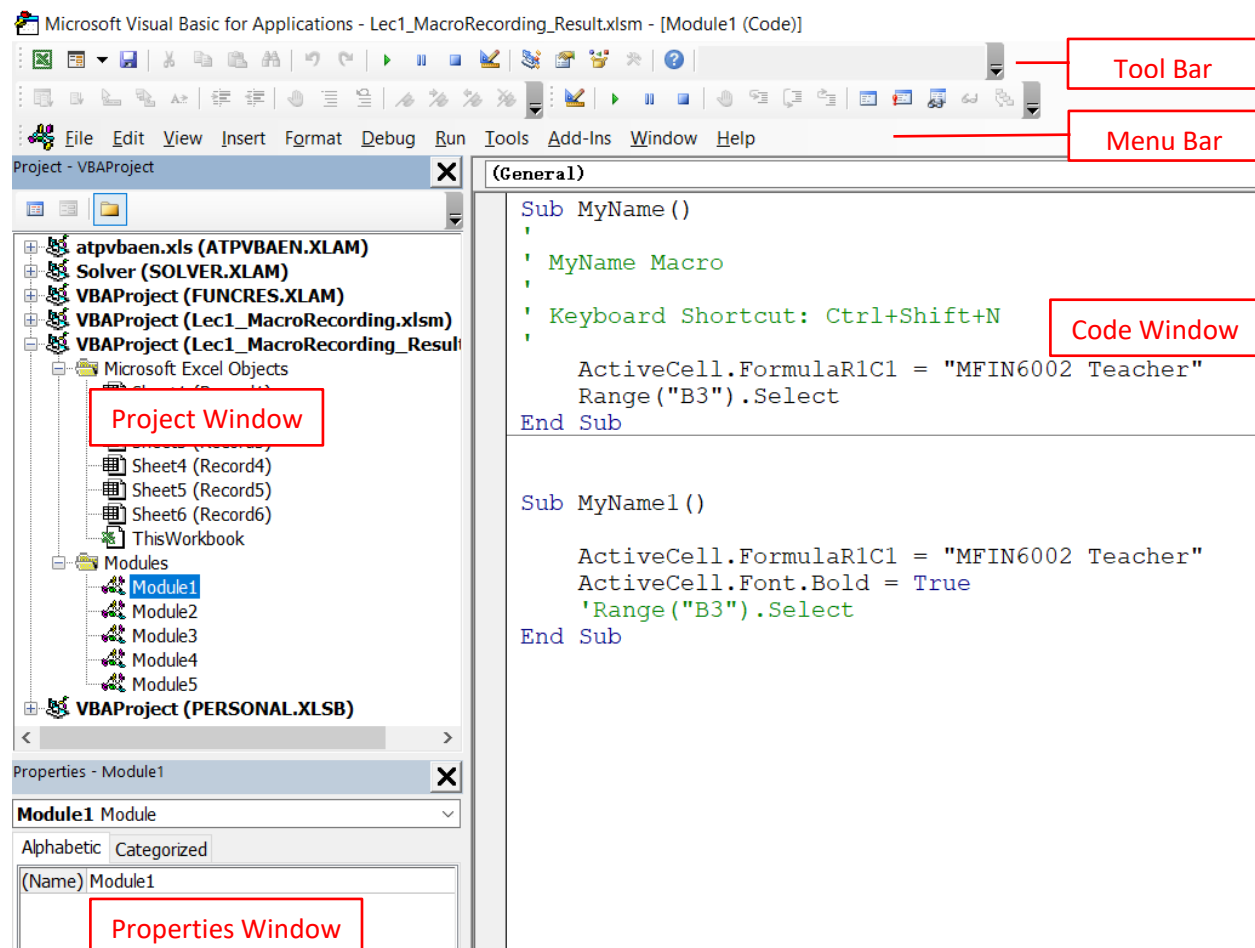
Note: The form controls and ActiveX controls look similar, they're quite different. Form controls are designed specifically for use on a spreadsheet, and ActiveX controls are typically used on Excel user forms. As a general rule, you should always use form controls when working on a spreadsheet. Why? Form controls need less overhead, so they perform

better, and configuring form controls is far easier than configuring their ActiveX counterparts.

2. Working with the Visual Basic Editor

The Visual Basic Editor (VBE) is a separate application that runs when you open Excel. To see this hidden VBE environment, you'll need to activate it. The quickest way to activate the VBE is to press Alt+F11 when Excel is active. To return to Excel, press Alt+F11 again. You can also activate the VBE by using the Visual Basic command on Excel's Developer tab.

The figure below shows the VBE program with some of the key parts identified. Chances are that your VBE program window won't look exactly the same. The VBE contains several windows and is highly customizable. You can hide windows, rearrange windows, dock windows, and so on.



2.1 Menu Bar

VBE's menu bar is similar to the menu bars in Excel and all other Microsoft Office applications, and you use it the same way. The shortcut key available for any menu command is shown next to it.

2.2 Toolbar

The toolbar below the menu bar also works the same way the Excel toolbar does. To see all the toolbars available, click View ⇒ Toolbar. You should always have the Standard, Edit and Debug toolbars open.

In Excel for Mac, Excel 365 version has Standard, Edit and Debug toolbars by default.

2.3 Project Explorer Window

The Project Explorer window is a narrow long window along the left side of your screen with the title "Project" in its title bar. If you do not see it, click View ⇒ Project Explorer, or press **Ctrl+R** to open it.

In Excel for Mac, if you do not see the Project Explorer window, go to View ⇒ Project Explorer or press **control+command+R**.

In the Project Explorer window, each open workbook is considered a project and is listed (along with a few other things) in a tree diagram. Every project expands to show at least one node called **Microsoft Excel Objects**. This node expands to show an item for each sheet in the workbook (each sheet is considered an object), and another object called ThisWorkbook (which represents the Workbook object).

Under the VBA project of the activated Excel file, find the node titled Modules and if it has a "+" next to it. Click on it to show all the modules and then double-click the module.

Modules are like worksheets and are saved in the same workbook. As with worksheets, you can add new modules as needed and remove any module you do not need anymore.

2.4 Adding a New Module

To add a new module, right-click the name of the project to open its shortcut menu and choose Insert ⇒ Module. You can also select the project's name and then choose Insert ⇒ Module from the menu bar.

When you record a macro, Excel automatically inserts a VBA module to hold the recorded code. The workbook that holds the module for the recorded macro depends on where you chose to store the recorded macro, just before you started recording.

In general, a VBA module can hold three types of code.

1. **Declarations:** One or more information statements that you provide to VBA. For example, you can declare the data type for variables you plan to use or set some other module-wide options.
2. **Sub procedures:** A set of programming instructions that performs some action. All recorded macros will be Sub procedures.
3. **Function procedures:** A set of programming instructions that returns a single value (similar in concept to a worksheet function, such as Sum).

A single VBA module can store any number of Sub procedures, Function procedures, and declarations. How you organize a VBA module is completely up to you. Some people prefer to keep all of their VBA code in a single VBA module; others like to split up the code into several different modules. It's a personal choice.

2.5 Renaming a Module

To rename a module you have to use the **Properties window** in VBE. To open it, in VBE press **F4** (In Excel for Mac, shortcut is **F6**) or choose View ⇨ Properties Window.

Select the module you want to rename and change the name next to Alphabetic tab. Modules are automatically arranged in alphabetical order.

2.6 Removing a Module

To remove a module, open the module to make sure this is the module you want removed and then right-click its name in the Project Explorer window. If you choose not to export it, the system will permanently remove the module.

2.7 Exporting and Importing Modules

To export a module to a file, right-click it in the Project Explorer window and in the shortcut menu click Export. The Export File dialog box will open where you can give the file a name (retaining the .bas extension) and save it in the appropriate directory.

To import a saved file, you can import only files previous created with Export. Right-click the project into which you want to import it.

2.8 Immediate Window

The Immediate window may or may not be visible. If it isn't visible, press **Ctrl+G** (In Excel for Mac, shortcut is **control+command+G**) or use the View ⇨ Immediate Window

command. To close the Immediate window, click the Close button in its title bar (or right-click anywhere in the Immediate window and select Hide from the shortcut menu).

The Immediate window is most useful for executing individual VBA statements and for debugging your code. If you're just starting out with VBA, this window won't be all that useful, so feel free to hide it and free up some screen space for other things.

In the next chapter, we will introduce `Debug.Print` statement which is used to write results in the Immediate window.

2.9 Code Window

This is the large window that takes up most of the right-hand side of the VBE window. It holds the modules in which you work with and store your codes.

Before you can do anything meaningful, you must have some VBA code in the VBA module. You can get VBA code into a VBA module in three ways.

- Use the Excel macro recorder to record your actions and convert them to VBA code.
- Enter the code directly.
- Copy the code from one module and paste it into another.

You have discovered the excellent method for creating code by using the Excel Macro recorder. However, not all tasks can be translated to VBA by recording a macro. You often have to enter your code directly into the module. Entering code directly basically means either typing the code yourself or copying and pasting code you have found from somewhere else.

Entering and editing text in a VBA module works as you might expect. You can select, copy, cut, paste, and do other things to the text.

2.10 Code Example:

Open Excel file "`Lec1_IntroductionToVBA.xlsm`". Read the message once and then click Enable Content (or Enable Macros). This will internally enable the macros in the workbook, but you will not see them yet. Open VBE by pressing `Alt+F11`.

Read the Sub procedure `FirstCode` in Module `A_GetStarted`.

Let's start by making a few general comments about how programs are organized and executed:

- A program comprises a series of **statements** or **instructions**. Each statement represents one step of the program (A statement may be one or more lines long.)

- Remember that the program is written in the VBA language and these sentences follow the syntax of VBA.
- When you run the program, VBA executes one statement at a time, starting at the top and moving down sequentially—unless it is instructed by the program itself to jump ahead or backward.
- In VBA an **equal (=) sign** instructs the computer to evaluate the part of the statement on its right and assign the result to whatever is on its left. This is why in VBA it is called the **assignment operator**; it is not an equal sign in the sense we normally use it.

2.11 Entering Code:

Start with typing the following line into a new module:

```
Sub procedurename ( )
```

Once you press Enter, VBA will add the line `End Sub`, and now you can start adding statements between those beginning and ending lines.

For entering code, the VBE works very much like a word processor except that it does a lot of editing for you. Once you press return at the end of a statement, VBE tries to convert whatever you have typed into legitimate VBA statements by properly capitalizing VBA key words, putting spaces where needed, and so forth (unless the line starts with an apostrophe to indicate that it is a comment). If there is any grammatical error (called “**compile error**”) in your statement, VBE typically detects them immediately, colors them in red, and displays a warning in a message box.

Note: VBA code is not case sensitive.

A simplified structure of a sub procedure looks like this:

```
Sub Procedure_Name ( [argumentlist] )  
    [Statements]  
    [Exit Sub]  
    [Statements]  
End Sub
```

A Sub procedure can **perform many actions** but **cannot return a value**. This is an important difference with Function procedures.

Sub procedures that do not have any argument can be run directly and show up in the Macro dialog box. We will introduce the sub procedures with arguments in later lectures.

This sub procedure uses the following concepts:

1. Defining a Sub procedure (the first line)
2. Declaring variables (the `Dim` statements)
3. Assigning values to variables (`loanAmount`, `intRate` and `intExpense`)
4. Joining strings of text (using the `&` operator)
5. Using `Debug.print` statement
6. Using object-oriented statements (`ThisWorkbook`, `Worksheets`, `Range`)
7. Ending a Sub procedure (the last line)

If this is your first exposure to VBA, you're not expected to understand all the statements in this example today. We will introduce all these concepts in the following lectures.

2.12 Organizing Procedures in Modules

Line continuation

VBA will let you make a statement as long as you want. But to make your statement readable, it is a good habit to break down long statements into two or more lines.

You can continue a statement in the next line by putting **a space** followed by **an underscore (`_`)** at the end of it.

For example, in `FirstCode` procedure,

```
ThisWorkbook.Worksheets("Sheet1").Range("A2").Value _  
    = "Interest rate is " & intRate * 100 & "%"
```

Multiple statements in one line

If you want to, you can put more than one statement in a line using a colon (`:`) as a separator.

```
loanAmount = 100000: intRate = 0.05
```

Adding comments

Comments are an extremely important part of the overall documentation of a project. There are at least three reasons to write comments in your code:

- We use comments to separate long codes into different sections.
- Comments are notes and reminders you include at various places in your VBA code to make it easier to understand and modify it in the future.

- If you want to test an alternative statement in your procedure, instead of deleting the original statement, you can turn it into a comment, and type the alternative one.

To insert a comment, precede the text with an apostrophe ('). Excel VBA colors the line green to indicate that it's a comment. To comment or uncomment multiple lines of codes, click **Comment Block** or **Uncomment Block** button on the Edit toolbar.

CHAPTER 2. INTRODUCING THE EXCEL OBJECT MODEL

VBA is an object-oriented programming language. The basic concept of **object-oriented programming** is that a software application (Excel in this case) consists of various individual objects.

An Excel application contains workbooks, worksheets, cells, charts, pivot tables, shapes, and the list goes on. Each object has its own set of attributes, which are called **properties**, and its own set of actions, called **methods**.

VBA objects have their identifiable attributes and actions. A workbook, for example, is an object with attributes (properties), such as its name, the number of worksheets it contains, and the date it was created. A workbook object also has actions (methods) such as Open, Close, and Save.

Excel thinks about objects such as workbooks, worksheets, and ranges internally as all part of a hierarchical model called the **Excel Object Model**. The Excel Object Model is a clearly defined set of objects that are structured according to the relationships among them.

1. Understanding Objects

In the real world, you can describe everything you see as an object. When you look at your house, it is an object. Your house has rooms; those rooms are also separate objects. Those rooms may have closets. Those closets are likewise objects. As you think about your house, the rooms, and the closets, you may see a hierarchical relationship among them. Excel works in the same way.

In Excel, the **Application object** is the all-encompassing object—similar to your house. Inside the Application object, Excel has a workbook. Inside a workbook is a worksheet. Inside that is a range. These are all objects that live in a hierarchical structure.

You write down an object by telling VBA the “address” of it. **The “address” is organized in hierarchies**, separated by periods (.). The most often used is workbook-worksheet-range hierarchy:

```
Application.ThisWorkbook.Sheets("Sheet1").Range("A1")  
Workbooks("myBook").Worksheets("mySheet").Range("A1:D3")
```

Comments:

1. If you include actual names of the workbook and the worksheet, you have to enclose them in quotation marks because they are texts. You can also use text variables containing the names, in which case you do not use the quotations.
2. To refer to just one cell like A1, you will use `Range ("A1")` in the above expression.
3. An alternate way to address just one cell is to use `Cells (rowNum, columnNum)` in place of the `Range` in the above expression. For example, `Cells (1, 1)` is the same as `Range ("A1")`. But you cannot use `Cells` to refer to more than one cell at a time.
4. `Range ("A1 : D3")` can also be addressed using `Range (Cells (1, 1) , Cells (3, 4))`.
5. If you do not specify the workbook, VBA assumes it to be the one where the code is running. So most of the time you can omit it.
6. If you do not specify a worksheet, VBA assumes it to be the active worksheet. The worksheet that is selected before you run a procedure is the active worksheet until you change it from the procedure. For example,

```
Worksheets ("Sheet3") .Select
```

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

7. Indeed, if you have your cursor already in cell A1, you can simply use the `ActiveCell` object, negating the need to spell out the range.

```
ActiveCell.Select
```

8. If you want to refer to a worksheet instead of a range, you will drop the `Range` part of the expression, and you may not have to specify the workbook as well.
9. All these objects (workbook, worksheet, etc.) belong to Excel, referred to as `Application`, so in principle, all the hierarchies should start with `"Application."`. But you almost always can omit it.

For more details about the objects, properties, and methods, please refer to the Excel file **"Appendix B VBA Quick Reference.xlsx"** on Moodle.

2. Understanding Collections

Many of Excel's objects belong to collections. Your house sits within a neighborhood, for example, which is a collection of houses called a neighborhood. Each neighborhood sits in a collection of neighborhoods called a city. Excel considers collections to be objects themselves.

In each Workbook object, you have a collection of Worksheets. The Worksheets collection is an object that you can call upon through VBA. Each worksheet in your workbook lives in the Worksheets collection.

If you want to refer to a worksheet in the Worksheets collection, you can refer to it by its position in the collection as an index number starting with 1 or by its name as quoted text. If you run these two lines of code in a workbook that has only one worksheet and that worksheet is called `MySheet`, they both do the same thing:

```
Sheet1.Select  
Worksheets(1).Select  
Worksheets("MySheet").Select
```

If you have two worksheets in the active workbook that have the names `MySheet` and `YourSheet`, in that order, you can refer to the second worksheet by typing either of these statements:

```
Sheet2.Select  
Worksheets(2).Select  
Worksheets("YourSheet").Select
```

If you want to refer to a worksheet in a workbook called `MySheet` in a particular workbook that is not active, you must qualify the worksheet reference and the workbook reference. Here's an example:

```
Workbooks("Lec2_ExcelObjectModel.xlsm").Worksheets("MySheet").  
Select
```

3. Understanding Properties

Properties are essentially the characteristics of an object. Your house has a color, a square footage, an age, and so on. Some properties can be changed, like the color of your house. Other properties can't be changed, like the year your house was constructed.

Likewise, an object in Excel, like the Worksheet object, has a sheet name property that can be changed, and a `Rows.Count` row property that cannot.

You refer to the property of an object by referring to the object and then the property. For instance, you can change the name of your worksheet by changing its `Name` property.

In this example, you are renaming `Sheet1` to `MySheet`:

```
Sheets("Sheet1").Name = "MySheet"
```

Some properties are read-only, which means you can't assign a value to them directly—for instance, the `Text` property of a cell. The `Text` property gives you the formatted appearance of value in a cell, but you cannot overwrite or change it.

You refer to a property of an object by attaching the name of the property to the object's address with a period (.).

```
Worksheets("Sales").Cells(3,5).Value = 100
Range("B4:B15").Name = "MonthlyCosts"
Range("B16").Formula = "=SUM(MonthlyCosts)"
Range("B16").NumberFormat = "$#,##0"
```

Specifying Properties for the Active Object

When you're working with Excel, only one workbook at a time can be active. In that workbook, only one sheet can be active. And if the sheet is a worksheet, one cell is the active cell (even if a multicell range is selected). VBA knows about active workbooks, worksheets, and cells, and it lets you refer to these active objects in a simplified manner.

This method of referring to objects is often useful because you won't always know the exact workbook, worksheet, or range on which you want to operate. VBA makes object referencing easy by providing properties of the `Application` object. For example, the `Application` object has an `ActiveCell` property that returns a reference to the active cell. The following instruction assigns the value 1 to the active cell:

```
ActiveCell.Value = 1
```

In the preceding example, we omitted the reference to the `Application` object and to the active worksheet because both are assumed. This instruction will fail if the active sheet isn't a worksheet. For example, if VBA executes this statement when a chart sheet is active, the procedure halts and you get an error message.

If a range is selected in a worksheet, the active cell is a cell within the selected range. In other words, the active cell is always a single cell (never a multicell range).

The `Application` object also has a `Selection` property that returns a reference to whatever is selected, which may be a single cell (the active cell), a range of cells, or an object such as `ChartObject`, `TextBox`, or `Shape`.

Table below lists the other `Application` properties that are useful when working with cells and ranges.

Property	Object Returned
ActiveCell	The active cell.
ActiveSheet	The active sheet.
ActiveWorkbook	The active workbook.
Selection	The object selected. It could be a Range object, Shape, ChartObject, and so on.
ThisWorkbook	The workbook that contains the VBA procedure being executed. This object may or may not be the same as the ActiveWorkbook object.

The advantage of using these properties to return an object is that you don't need to know which cell, worksheet, or workbook is active, and you don't need to provide a specific reference to it. This allows you to write VBA code that isn't specific to a particular workbook, sheet, or range. For example, the following instruction clears the contents of the active cell, even though the address of the active cell isn't known:

```
ActiveCell.ClearContents
```

The example that follows displays a message that tells you the name of the active sheet:

```
MsgBox ActiveSheet.Name
```

If you want to know the name and directory path of the active workbook, use a statement like this:

```
MsgBox ActiveWorkbook.FullName
```

If a range on a worksheet is selected, you can fill the entire range with a value by executing a single statement. In the following example, the `Selection` property of the `Application` object returns a `Range` object that corresponds to the selected cells. The instruction simply modifies the `Value` property of this `Range` object, and the result is a range filled with a single value.

```
Selection.Value = 12
```

If something other than a range is selected (such as a `ChartObject` or a `Shape`), the preceding statement generates an error because `ChartObject` and `Shape` objects don't have a `Value` property.

To find out how many cells are selected in the active window, access the `Count` property. Here's an example:

```
MsgBox ActiveWindow.RangeSelection.Count
```

4. Understanding Methods

Methods are the actions that can be performed with an object. It helps to think of methods as verbs. A simple example of an Excel method is the `Select` method of the `Range` object.

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

Another is the `Copy` method of the `Range` object.

```
Range("A1").Copy
```

Some methods have arguments that can dictate how they are applied. For instance, the `Paste` method can be used more effectively by explicitly defining the `Destination` argument.

```
ActiveSheet.Paste Destination:=Range("B1")
```

Range Object

Much of the work that you will do in VBA involves cells and ranges in worksheets. That being the case, let's take some time to use the `Range` object as a case study on how to explore and get familiar with a specific object.

`Range` object exposes three properties that can be used to manipulate your worksheets via VBA.

- The `Range` property of a `Worksheet` or `Range` class object
- The `Cells` property of a `Worksheet` object
- The `Offset` property of a `Range` object

The Range Property

The `Range` property returns a `Range` object. This property has two syntaxes:

```
object.Range(cell1)
```

```
object.Range(cell1, cell2)
```

The `Range` property applies to two types of objects: a `Worksheet` object or a `Range` object. Here, `cell1` and `cell2` refer to placeholders for terms that Excel recognizes as

identifying the range (in the first instance) and delineating the range (in the second instance). The following are a few examples of using the `Range` property.

The instruction that follows simply enters a value into the specified cell. In this case, it puts the value 12.3 into cell A1 on `Sheet1` of the active workbook.

```
Worksheets("Sheet1").Range("A1").Value = 12.3
```

The `Range` property also recognizes defined names in workbooks. Therefore, if a cell is named `Input`, you can use the following statement to enter a value into that named cell:

```
Worksheets("Sheet1").Range("Input").Value = 100
```

The example that follows enters the same value in a range of 20 cells on the active sheet. If the active sheet isn't a worksheet, the statement causes an error message.

```
ActiveSheet.Range("A1:B10").Value = 2
```

The next example produces the same result as the preceding example:

```
Range("A1", "B10") = 2
```

The sheet reference is omitted, however, so the active sheet is assumed. Also, the `Value` property is omitted, so the default property (which is `Value` for a `Range` object) is assumed. This example also uses the second syntax of the `Range` property. With this syntax, the first argument is the cell at the top left of the range, and the second argument is the cell at the lower right of the range.

The following example uses the Excel range intersection operator (a space) to return the intersection of two ranges. In this case, the intersection is a single cell, C6. Therefore, this statement enters 3 in cell C6:

```
Range("C1:C10 A6:E6") = 3
```

Finally, if the range you're referencing is a noncontiguous range (a range where not all the cells are adjacent to each other), you can use commas to serve as a union operator. For example, the following statement enters the value 4 in five cells that make up a noncontiguous range. Note that the commas are within the quote marks.

```
Range("A1,A3,A5,A7,A9") = 4
```

So far, all the examples have used the `Range` property on a `Worksheet` object. As mentioned, you can also use the `Range` property on a `Range` object. For example, the following line of code treats the `Range` object as if it were the upper-left cell in the worksheet, and then it enters a value of 5 in the cell that would be B2. In other words, the reference returned is relative to the upper-left corner of the `Range` object. Therefore, the

statement that follows enters a value of 5 into the cell directly to the right and one row below the active cell:

```
ActiveCell.Range("B2") = 5
```

Fortunately, you can access a cell relative to a range in a much clearer way—the `Offset` property. We discuss this property after the next section.

The Cells Property

Another way to reference a range is to use the `Cells` property. You can use the `Cells` property, like the `Range` property, on `Worksheet` objects and `Range` objects. The `Cells` property has three syntaxes.

```
object.Cells(rowIndex, columnIndex)
```

```
object.Cells(rowIndex)
```

```
object.Cells
```

Some examples demonstrate how to use the `Cells` property. The first example enters the value 9 in cell A1 on `Sheet1`. In this case, we're using the first syntax, which accepts the index number of the row (from 1 to 1048576) and the index number of the column (from 1 to 16384):

```
Worksheets("Sheet1").Cells(1, 1) = 9
```

Here's an example that enters the value 7 in cell D3 (that is, row 3, column 4) in the active worksheet:

```
ActiveSheet.Cells(3, 4) = 7
```

You can also use the `Cells` property on a `Range` object. When you do so, the `Range` object returned by the `Cells` property is relative to the upper-left cell of the referenced `Range`. For example, if the `Range` object is A1:D10 (40 cells), the `Cells` property can have an argument from 1 to 40 and can return one of the cells in the `Range` object. In the following example, a value of 2000 is entered in cell A2 because A2 is the 5th cell (counting from the top, to the right, and then down) in the referenced range:

```
Range("A1:D10").Cells(5) = 2000
```

The Offset Property

The `Offset` property, like the `Range` and `Cells` properties, also returns a `Range` object. But unlike the other two methods discussed, the `Offset` property applies only to a `Range` object and no other class. Its syntax is as follows:

```
object.Offset(rowOffset, columnOffset)
```

The `Offset` property takes two arguments that correspond to the relative position from the upper-left cell of the specified `Range` object. The arguments can be positive (down or to the right), negative (up or to the left), or 0. The example that follows enters a value of 12 into the cell directly below the active cell:

```
ActiveCell.Offset(1,0).Value = 12
```

The next example enters a value of 15 in the cell directly above the active cell:

```
ActiveCell.Offset(-1,0).Value = 15
```

If the active cell is in row 1, the `Offset` property in the preceding example generates an error because it can't return a `Range` object that doesn't exist. The `Offset` property is useful, especially when you use variables in looping procedures. We discuss these topics in the next chapter.

When you record a macro using the relative reference mode, Excel uses the `Offset` property to reference cells relative to the starting position (that is, the active cell when macro recording begins). For example, we used the macro recorder to generate the following code. We started with the cell pointer in cell B1, entered values into B1:B3, and then returned to B1.

```
Sub AddTotalRelative()  
    ActiveCell.Offset(15, 0).Range("A1").Select  
    ActiveCell.FormulaR1C1 = "Total"  
    ActiveCell.Offset(0, 3).Range("A1").Select  
    ActiveCell.FormulaR1C1 = "=COUNTA(R[-14]C:R[-1]C)"  
    ActiveCell.Offset(1, 0).Range("A1").Select  
End Sub
```

The generated code references cell A1—a cell that wasn't even involved in the macro. This notation is a quirk in the macro recording procedure that makes the code more complex than necessary. You can delete all references to `Range("A1")`, and the macro still works perfectly.

```
Sub AddTotalRelative_Mod1()  
    ActiveCell.Offset(15, 0).Select  
    ActiveCell.FormulaR1C1 = "Total"  
    ActiveCell.Offset(0, 3).Select  
    ActiveCell.FormulaR1C1 = "=COUNTA(R[-14]C:R[-1]C)"  
    ActiveCell.Offset(1, 0).Select  
End Sub
```

In fact, you can enter this much more efficient version of the macro. The macro recorder uses the `FormulaR1C1` property. Normally, you want to use the `Value` property to enter a value in a cell. However, using `FormulaR1C1` or even `Formula` produces the same result.

```
Sub AddTotalRelative_Mod2()  
    ActiveCell.Offset(15, 0) = "Total"  
    ActiveCell.Offset(15, 3) = "=COUNTA(R[-14]C:R[-1]C)"  
    ActiveCell.Offset(16, 3).Select  
End Sub
```

CHAPTER 3. VBA PROGRAMMING FUNDAMENTALS

1. Operators

Assignment Operator

In VBA, the equals sign (=) is used as the assignment operator. It means that when a statement has an equals sign in it, VBA first evaluates whatever is on the right-hand side of it and then assigns the result to whatever is on the left-hand side.

Like $x = x + 1$, which you will see all the time in VBA, it means adding 1 to the current value of x and making that the new value of x .

Mathematical Operators

Same as in Excel. For example, * is multiplication operator, ^ is power operator.

Comparison Operators

Same as in Excel. For example, >= is greater or equal to, <= is less or equal to, <> is not equal to.

Logical Operators

Same as in Excel: And, Or, Not. For example,

```
If loanSize > 1000000 And credRat = 1 Then intRate = 0.08  
If loanSize > 1000000 Or credRat = 1 Then intRate = 0.08
```

2. Variables, Data Types, and Constants

VBA's main purpose is to manipulate data. Some data resides in objects, such as worksheet ranges. Other data is stored in variables that you create.

You can think of a variable as a named storage location in your computer's memory. Variables can accommodate a wide variety of data types—from simple Boolean values (True or False) to large, double-precision values. You assign a value to a variable by using the equal sign operator.

You make your life easier if you get into the habit of making your variable names as descriptive as possible. VBA does, however, have a few rules regarding variable names.

- You can use alphabetic characters, numbers, and some punctuation characters, but the first character must be alphabetic.
- VBA doesn't distinguish between cases. To make variable names more readable, programmers often use mixed case (for example, `InterestRate` rather than `interestrate`).
- You can't use spaces or periods. To make variable names more readable, programmers often use the underscore character (`Interest_Rate`).
- You can't embed special type declaration characters (`#`, `$`, `%`, `&`, or `!`) in a variable name.
- You cannot use names identical to the names and words used by VBA, for example `MsgBox`.
- Variable names can be as long as 254 characters—but using such long variable names isn't recommended.
- If you are working on a big project with many variables, then you should first create and write down a variable-naming convention for the project before you start coding. This will save you a lot of time later.

The following list contains some examples of assignment expressions that use various types of variables. The variable names are to the left of the equal sign. Each statement assigns the value to the right of the equal sign to the variable on the left.

```
x = 1
InterestRate = 0.075
LoanPayoffAmount = 243089.87
DataEntered = False
x = x + 1
MyNum = YourNum * 1.25
UserName = "Bob Johnson"
DateStarted = #12/14/2012#
```

VBA has many reserved words, which are words that you can't use for variable or procedure names. If you attempt to use one of these words, you get an error message. For example, although the reserved word `Next` might make a very descriptive variable name, the following instruction generates a syntax error:

```
Dim Next
Next = 132
```



Unfortunately, syntax error messages aren't always descriptive. If the Auto Syntax Check option is turned on, you get the error `Compile error: Expected: variable`. If Auto Syntax Check is turned off, attempting to execute this statement results in `Compile error: Syntax error`.

2.1 Defining Data Types

VBA makes life easy for programmers because it can automatically handle all the details involved in dealing with data. Some programming languages, however, are strictly typed, which means that the programmer must explicitly define the data type for every variable used.

Data type refers to how data is stored in memory—as integers, real numbers, strings, and so on. Although VBA can take care of data typing automatically, it does so at a cost: slower execution and less efficient use of memory. As a result, letting VBA handle data typing may present problems when you're running large or complex applications.

Another advantage of explicitly declaring your variables as a particular data type is that VBA can perform some additional error checking at the compile stage. These errors might otherwise be difficult to locate.

Check [VBA online document](#) for more details.

`Variant` is the most flexible data type. Variables declared as `Variant` can store any type of data, and if you do not specify the data type for a variable, VBA automatically uses `Variant` data type for it.

For worksheet calculation, Excel uses the `Double` data type, so that's a good choice for processing numbers in VBA when you don't want to lose any precision. For integer calculations, you can use the `Integer` type (which is limited to values less than or equal to 32,767). Otherwise, use the `Long` data type. In fact, using the `Long` data type even for values less than 32,767 is recommended because this data type may be a bit faster than using the `Integer` type. When dealing with Excel worksheet row numbers, you want to

use the `Long` data type because the number of rows in a worksheet exceeds the maximum value for the `Integer` data type.

2.2 Declaring Variables

You generally declare the data type of a variable using the `Dim` statement at the beginning of a procedure before any executable statement.

```
Dim issueDate As Date, maturityDate As Date
Dim numDays, nameDays As String, myDataType
```

- You have to declare the data type of each variable separately, but you can declare several variables in one `Dim` line by separating them by commas.
- If you don't declare the data type for `numDays` and `myDataType`, VBA assumes them to be `Variant`.
- When declaring variables, it is good practice to group related variables together to make it easier to find them later.

Data stored as a `Variant` acts like a chameleon: it changes type, depending on what you do with it.

Go to Excel file "Lec3_VBAFundamentals.xlsm" Module `A_Variables`.

`VariantDemo` procedure demonstrates how a variable can assume different data types:

```
Sub VariantDemo()
    MyVar = True
    MsgBox TypeName(MyVar)
    MyVar = MyVar * 100
    MsgBox TypeName(MyVar)
    MyVar = MyVar / 4
    MyVar = "Answer: " & MyVar
    MsgBox MyVar
End Sub
```

Open the Locals Window and step into the `VariantDemo` procedure. `MyVar` starts as a `Boolean`. The multiplication operation converts it to an `Integer`. The division operation converts it to a `Double`. Finally, it's concatenated with text to make it a `String`. The `MsgBox` statement displays the final string: `Answer: -25`.

You can use the VBA `TypeName` function to determine the data type of a variable.

To demonstrate further the potential problems in dealing with `Variant` data types, try executing this procedure:

```
Sub VariantDemo2()  
    MyVar = "123"  
    MyVar = MyVar + MyVar  
    MyVar = "Answer: " & MyVar  
    MsgBox MyVar  
End Sub
```

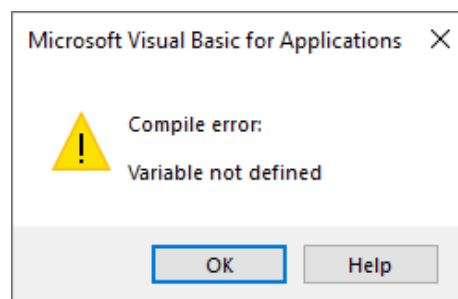
The message box displays `Answer: 123123`. This is probably not what you wanted. When dealing with variants that contain text strings, the `+` operator will join (concatenate) the strings together rather than perform addition.

Forcing Variable Declaration: An even safer approach is to force yourself to declare all variables because that will catch all misspelled variables. To force variable declaration, include the statement `Option Explicit` as the first statement in every VBA module. This will make your program stop and display an error message during execution if VBA encounters any variable you have not declared.

You can have VBA automatically enter this statement at the top of every new module by selecting `Tools` ⇒ `Options` ⇒ select `Require Variable Declaration` in the `Editor` tab.

In Excel for Mac, activate VBE window then go to `Excel` menu ⇒ `Preferences` ⇒ select `Require Variable Declaration`.

I recommend that you always keep this option selected. After you select this option, VBA's way of telling you that your procedure contains an undeclared variable is



2.3 Scope of Variables

Scope refers to in what parts of a project and at what times during execution a variable and its value will be available. Depending on how you declare a variable, it can be available only to one procedure, all procedures in a module, or all procedures in a project (that is, all the

modules in the workbook). In large projects, especially those extending over many modules, specifying the right scope for your variables is more important.

Procedure-Level Variables: If you declare a variable using a `Dim` statement inside a procedure, then it becomes a procedure-level variable. The variable can be used and its value is available only within that procedure. When the procedure ends, the variable no longer exists, and Excel frees up the memory that the variable used.

Multiple procedures can use the same variable name for procedure-level variables, but each instance of the variable is unique to its own procedure.

In general, local variables are the most efficient because VBA frees up the memory that they use when the procedure ends.

Module-Level Variables: If you want to make a variable available to all procedures in a module (but not to all the other modules in the project) then you should declare it using a `Dim` or a `Private` statement before the first procedure of the module. The value of a module-level variable remains available as long as the code (project) is running.

Project-Level Variable: To make a variable available to all the procedures in a project declare it using a `Public` statement at the beginning of one of the VBA modules of the project before the first procedure of the module. A project-level variable remains available as long as the code (project) is running.

Scope	To Declare a Variable with This Scope
Single procedure	Include a <code>Dim</code> or <code>Static</code> statement within the procedure.
Single module	Include a <code>Dim</code> or <code>Private</code> statement before the first procedure in a module.
All modules	Include a <code>Public</code> statement before the first procedure in a module.

2.4 Working with Constants

If you need to use some constant values in your procedures, you can give them meaningful names using the `Const` statement and then use the names throughout your code. This will make your code easier to read, and in case you need to change one of the values, you will have to make it in only one place. Defining a constant is a safe way to avoid accidentally changing the value in the code.

As with variables, the scope of a constant depends on where you declare it.

If you declare a constant using a `Const` statement within a procedure, then it will be available only to that procedure. If you place the `Const` statement before the first procedure of a module, then it will be available to all procedures in that module only. To make a `Const` available to all procedures in a project, use the statement `Public Const` instead of just `Constant` and put it in any standard VBA module of the project before the module's first procedure.

Here are some examples:

```
Const intRate = 0.05
Const myName as String = "Jim Smith"
Public Const busSgmnts as Integer = 6
```

2.5 Working with Text

Data type of text is `String`. In sub procedure `DataType` in module `A_Variable`, the `nameDays` variable is declared to be a string.

```
Dim nameDays As String
```

We can declare `nameDays` variable as a string with a maximum length of 5 characters by

```
Dim nameDays As String * 5
```

A variable you intend to use for storing text must be of `Variant` or `String` data type. For example:

```
firstName = "John": lastName = "Smith"
```

Texts always have to be enclosed in quotation marks (`"`). To join the texts, we use ampersand (`&`). The ampersand does not add any space when it joins texts; you have to provide any space you want. If you must include a quotation mark as one of the characters in the string, you use two contiguous quotation marks (`"`).

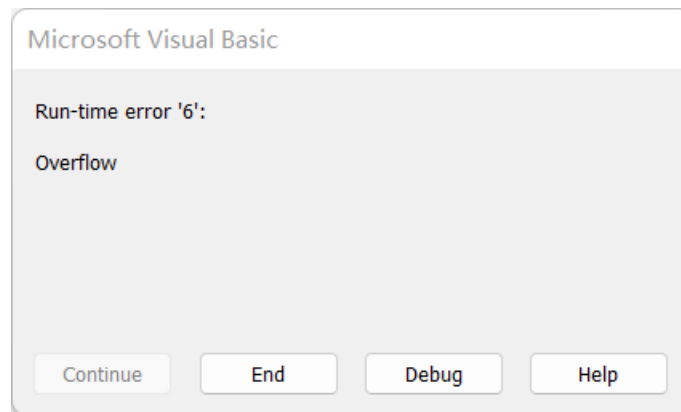
```
fullName = "John " & "Smith"
fullName = firstName & " " & lastName
```

Both Excel and VBA also offer many built-in functions to manipulate text variables. For example, you can create a new variable `shortName` with the first 3 characters of `firstName` using the function `Left()` to extract them as follows:

```
shortName = Left(firstName, 3)
```

2.6 Working with Numbers

A variable declared as `Integer` can hold whole numbers from -32,768 to 32,767. If you attempt to assign a value outside that range to an `Integer` variable, you'll receive an error:



To handle larger whole numbers, you can declare the variable as `Long` data type. Note that if you declare a variable as `Integer` or `Long`, you should be careful about assigning real numbers to it or using it in mathematical operations because you may get unexpected results.

When you declare the data type of a variable as integer, but you try to set it to a fraction, the number is rounded up or down to the nearest integer value. If the number is equally close to two integer values, the value is rounded to the nearest even integer. For example, 10.5 is rounded down to 10, and 11.5 is rounded up to 12.

For real numbers, you can use the `Single` data type or, if you want to carry a larger number of significant digits, the `Double` data type.

2.7 Working with Dates

A variable defined as a date uses 8 bytes of storage and can hold dates ranging from January 1, 0100, to December 31, 9999. That's a span of nearly 10,000 years—more than enough for even the most aggressive financial forecast!

To store dates or time in a variable, you must enclose a Date literal within number signs (`#`). You must specify the date value in the format **M/d/yyyy**, for example `#5/31/1993#`, or **yyyy-MM-dd**, for example `#1993-5-31#`. This requirement is independent of your locale and your computer's date and time format settings.

You can specify the time value in either 12-hour or 24-hour format, for example #1:15:30 PM#, #13:15:30# or specify both date and time #2/2/1998 5:14:00 PM#.

If you want to do date mathematics (for example, find the number of days between two dates), make sure you use `Date` or `Variant` variables and not `String` variables. (See Sub procedure `DataType`.)

Also check [VBA online document](#) for more details.

2.8 Working with Object

An object variable is one that represents an entire object, such as a range or a worksheet. Object variables are important for two reasons.

- They can simplify your code significantly.
- They can make your code execute more quickly.

Object variables, like normal variables, are declared with the `Dim` or `Private` or `Public` statement. For example, the following statement declares `DataRng` as a `Range` object variable:

```
Dim DataRng As Range
```

Use the `Set` keyword to assign an object to the variable. Here's an example:

```
Set DataRng = Range("B2:B26")
```

(See Sub procedure `Data_v1`, `Data_v2`, `Data_v3` in Module `F_ReadData`.)

2.9 Initial Value of Variables

When a procedure begins running, all variables are initialized. A numeric variable is initialized to 0, a variable-length string is initialized to a zero-length string (""), and a fixed-length string is filled with the character represented by the ASCII character code 0, or `Chr(0)`. Variant variables are initialized to `Empty`. When you declare an object variable, space is reserved in memory, but its value is set to `Nothing` until you assign an object reference to it by using the `Set` statement.

2.10 Array Variables

Variables and constants hold only one piece of data at a time. If you have a large quantity of related data such as 300 days of price history for a stock, you will need 300 single-element variables to store and manipulate them, which is cumbersome and inefficient.

VBA offers a second kind of variable, called array variable that can hold any quantity of data under the same name. You refer to a specific element of the array by the array name and one or more index numbers enclosed in parentheses. For example, you can store the 300 days of stock price history in an array called `stkPrice` with 300 elements and refer to the data for the first day as `stkPrice(1)` and so forth. If the price on the first day was 30.2, then you will store it in the array using a statement like `stkPrice(1) = 30.2` and refer to it by that name in other statements.

Array Dimension

A one-dimensional array (like the one we talked about for storing 300 days of stock price history) can be viewed as a row or column of data.

If you had 300 days of price data for 5 different stocks, you could store them all in a one-dimensional array with 1,500 elements or five different arrays with 300 elements each.

A two-dimensional array is similar to an Excel spreadsheet. You need two index numbers—a row number and a column number—to refer to its individual elements. For example, you would refer to the price of the third stock on the 100th day as `stkPrice(100, 3)`.

If instead of just the closing price you had 4 different prices (open, high, low, close) for each stock for each day, then you could use a three-dimensional array to store the data, with the first index referring to the day of the data, the second to the data type (1 for open, 2 for high, etc.) and the third to the stock number. Thus `stkData(51, 2, 3)` could refer to the 51st day's high price for the third stock.

Declaring Arrays

As with variables, you can declare (specify) the data type for an array or leave the data type unspecified, in which case VBA will assume the data type to be `Variant`. You must declare the size of an array, though, so that VBA knows how much memory it must allocate for the array.

You declare an array the same way as other variables using the `Dim`, `Static`, `Private`, or `Public` statements, except that in the case of an array you generally specify the size of the array at the same time. VBA arrays can have up to 60 dimensions, although you'll rarely need more than three dimensions (a 3D array).

```
Dim stkPrice(1 To 300, 1 To 5) As Single, tkrSym(1 To 5)
```

This example declares a two-dimensional array called `stkPrice` with 300 rows and 5 columns and data type `Single`. You can think of `stkPrice` as occupying a 300×5

matrix. To refer to a specific element in a 2D array, you need to specify two index numbers. For example, here's how you can assign a value to an element in the preceding array:

```
stkPrice(5, 1) = 100
```

The second array, which is for storing the ticker symbols of the stocks, is a one-dimensional array with 5 elements. Since the data type for it is not specified, VBA will assume it to be `Variant`. You can assign any type of data to each element. Elements can also have different types.

```
tkrSym(1) = "AAPL"  
tkrSym(5) = 1003
```

You can also specify only the upper bound of an array index as in the second example, but you have to be careful. If you do not specify the lower bound of an index, VBA assumes by default that the lower bound of the index is 0 and not 1. For example,

```
Dim stkPrice(300, 5) As Single, tkrSym(5)
```

The `stkPrice` array is a two-dimensional array of size 301×6 .

If you prefer to have the index values start at 1 rather than 0, you can change the default setting by including the following statement before any procedure in the module:

```
Option Base 1
```

In the third example,

```
Dim annlSales(1990 To 2005)
```

The array will hold sales numbers for years 1990 to 2005, and if you dimension it as shown, you will be able to refer to the sales for a particular year more easily by using the year number as the index, such as in `annlSales(2001)`.

Dynamic Arrays

If the size of an array is not known at the time of writing the code (for example, the size will depend on some input data) or you want to specify and change it during the execution of the code, then you can set up the array as a dynamic array.

Here, you declare the array as usual but with a blank set of parentheses as follows:

```
Dim stkPrice() As Single
```

Before you use the array, you must specify its dimensions using the `ReDim` statement. You can `ReDim` an array in your procedure any number of times. When you change an array's dimensions, the existing values are destroyed. If you want to preserve the values in the array when you `ReDim` it, you have to use `ReDim Preserve` statement instead.

Please note that `ReDim Preserve` statement can preserve the data in the existing array only when you change the size of only the last dimension.

Also check [VBA online document](#) for more details.

Some Functions for Arrays

`LBound` and `UBound` functions are used to find the lowest and highest subscripts (that is, boundaries) for each dimension of an array. The syntax is:

```
LBound (arrayname [, dimension])
UBound (arrayname [, dimension])
```

For the argument `dimension` you enter 1,2, and so on, to indicate if you want the lowest subscript for the first dimension (row), second dimension (column), and so on. If you skip the argument `dimension`, VBA assumes it to be 1.

A function that returns a VBA array

VBA includes a useful function called **Array**. The `Array` function returns a variant that contains an array (that is, multiple values). Syntax is

```
Array(arglist)
```

The required `arglist` argument is a comma-delimited list of values that are assigned to the elements of the array contained within the `Variant`. If no arguments are specified, an array of zero length is created.

```
Sub Array_Fn()
    Dim Arr
    Arr = Array("Jan", "Feb", "Mar", "Apr", "May", "Jun",
               "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")
End Sub
```

`Arr` is a 1D array variable with 12 elements, values of which are manually entered in `Array` function.

(See Sub procedure `Array_1D`, `Array_2D`, `Array_Dyn`, `Array_Fn` in Module `B_Array`, and `ArrayVal_1D`, `ArrayVal_2D`, `ArrayVal_3D` in Module `E_Looping`.)

3. Controlling Code Execution – Branching

In VBA procedures, you can include statements that will make execution jump forward or backward over one or more lines of statements. Often, however, you need to control the flow of your routines by skipping over some statements, executing some statements multiple times, and testing conditions to determine what the routine does next.

The preceding section describes **branching** statements and **looping** statements.

When the action is dependent upon whether certain conditions you specify are satisfied, this is called *branching*.

You can also make VBA repeat a group of statements until or while certain conditions are satisfied. This is called *looping*.

3.1 If-Then Constructs

If-Then statements are the most commonly used branching or decision-making statements in VBA.

➤ If ... Then ... Else ... Statement

The syntax of the statement is:

```
If condition Then true_statement [Else false_statement]
```

If the condition is satisfied, then the *true_statement* is executed; otherwise the *false_statement* is executed. The condition can be a simple condition or you can create a complex condition using logical operators. The *Else* part of the statement is optional. If you omit it then the *true_statement* will be executed if the condition is satisfied; otherwise control will move on to the next statement.

If statement exclusively for cases where you have only one *true_statement* and only one *false_statement* to execute.

(See Sub procedure `if_test1` in Module `C_Branching`.)

➤ If ... Then ... End If Statement

The syntax of the statement is:

```
If condition Then
    true_statement1
    true_statement2
```



```

    true_statement3
    ...
End If

```

You can put any number of statements between the first and the last lines. If the condition is not satisfied, all the intermediate lines will be skipped and control will pass to the line following the `End If` line.

➤ **If ... Then ... Else ... End If Statement**

The syntax of the statement is:

```

If condition Then
    true_statements
Else
    false_statements
End If

```

Here is only one condition. (See Sub procedure `if_test2`)

If there are more conditions, we can create nested `If` statements in both *true_statements* and *false_statements*. (See Sub procedure `if_test3`)

If ... Then ... ElseIf ... Else ... End If Statement

You can also create a nested `If` structure using `ElseIf` as follows:

```

If condition1 Then
    true_statements
ElseIf condition2 Then
    1false_2true_statements
ElseIf condition3 Then
    1false_2false_3true_statements
Else
    1false_2false_3false_statements
End If

```

(See Sub procedure `if_test4`)

3.2 GoTo Statement

You use a `GoTo` statement to transfer program control to an earlier or later line in a procedure (instead of the next line). The line you want control to jump to must be a line

that has only a label, which is a text string that starts with a letter and ends with a colon (:). From there, control will again start moving line by line.

The label can also be a number (with no colon), but using labels that are descriptive text strings will make your code more readable. Of course, you want a `GoTo` statement to be executed only when some condition is met. `GoTo` statements are therefore almost always used in conjunction with some kind of `If` statement.

(See Sub procedure `GoToExample` and `AlternateTo_GoToExample`)

3.3 Select Case Constructs

The `Select Case` construct is useful for choosing among three or more options. This construct also works with two options, and it is a good alternative to `If-Then-Else`. The syntax for `Select Case` is as follows:

```
Select Case testexpression
  [Case expressionlist-n
    [instructions-n]]
  [Case Else
    [default_instructions]]
End Select
```

If *testexpression* matches any *Case expressionlist* expression, the statements following that *Case* clause are executed up to the next *Case* clause, or, for the last clause, up to `End Select`. Control then passes to the statement following `End Select`. If *testexpression* matches an *expressionlist* expression in more than one *Case* clause, only the statements following the first match are executed.

The `Case Else` clause is used to indicate the *elsestatements* to be executed if no match is found between the *testexpression* and an *expressionlist* in any of the other *Case* selections. Although not required, it is a good idea to have a `Case Else` statement in your `Select Case` block to handle unforeseen *testexpression* values. If no *Case expressionlist* matches *testexpression* and there is no `Case Else` statement, execution continues at the statement following `End Select`.

(See Sub procedure `SelectCase1`, `SelectCase2`, `SelectCase3`)

4. Controlling Code Execution – Looping

In financial models, you often want to evaluate the same formulas many times with different values for one or more of the variables involved. In VBA you do it by creating loops. Loops are a lot more flexible and powerful than Excel's copy-and-paste approach.

4.1 For ... Next Statement

This is the simplest and most widely used loop. The syntax of the statement is:

```
For counter = startValue to endValue [Step stepValue]
    [Statements]
[Exit For]
[Statements]
Next [counter]
```

The *counter* is a numeric variable you have to specify. The first time the loop is executed, the value of the counter is set equal to *startValue* and then every time the loop is repeated the value of the counter is increased by the *stepValue*. If you omit the *Step stepValue*, then by default *stepValue* is 1.

Notes:

- Although you will often use positive integers for *startValue*, *endValue*, and *stepValue*, you can also use negative integers and numbers with decimal points for any of them.
- Every time VBA loops back to the first line, it calculates the new value for the counter and compares it to the *endValue*. If the *stepValue* you have specified is positive or 0, then the loop is repeated as long as *counter* is \leq *endValue*. If the *stepValue* is negative, then the loop is repeated as long as *counter* is \geq *endValue*.
- Although you are allowed to do so, do not write any statement inside the loop to change the value of the counter. It will make reading and debugging the code more difficult.
- If you want VBA to exit the loop before all the repetitions have been completed if certain conditions are met. You can do so by including one or more `Exit For` statements in the loop in a way that control will reach one of them only when one of the conditions is met.
- You do not have to include the *counter* after the `Next` in the last line of the loop. However, if you have several `For . . . Next` loops in your code, then including the

counter in the last line makes it easier to recognize where a particular loop begins and ends.

We can include loops within other loops, which is called nested loops.

(See Sub procedure `ArrayVal_1D`, `ArrayVal_2D`, `ArrayVal_3D`, `ForNext` in Module `E_Looping`)

4.2 Do ... Loop Statement

A `Do ... Loop` does not have a loop counter. Instead, you can—but do not have to—include a condition in the `Do` or `Loop` part of the statement to specify that the loop will repeat *while* or *until* that condition is met. You can also use one or more `Exit Do` statements within the loop to exit under additional conditions. Four syntaxes:

```
Do [While condition]  
    [Statements]  
    [Exit Do]  
    [Statements]  
Loop  
  
Do  
    [Statements]  
    [Exit Do]  
    [Statements]  
Loop [While condition]  
  
Do [Until condition]  
    [Statements]  
    [Exit Do]  
    [Statements]  
Loop  
  
Do  
    [Statements]  
    [Exit Do]  
    [Statements]  
Loop [Until condition]
```

(See Sub procedure `DoLoop1`, `DoLoop2` and `DoLoop3` in Module `E_Looping`)

4.3 For Each ... Next Statement

If we repeat a group of statements for each element in an array or collection, we use `For Each ... Next` statement. Syntax:

```
For Each element In group
    [Statements]
    [Exit For]
    [Statements]
Next [element]
```

The group can be the name of an array or an object collection.

(See Sub procedure `ForEachNext1`, `ForEachNext2` in Module `E_Looping`)

4.4 With ... End With Statement

The `With...End With` construct looks like a looping structure, but its purpose is to let you shorten your statements when you are repeatedly dealing with the same object.

Syntax:

```
With object
    Statements
End With
```

(See Sub procedure `With_Test1`, `With_Test2` in Module `E_Looping`)

5. Built-in Functions

VBA offers dozens of built-in functions that you use in VBA statements essentially the same way you use Excel functions in cell formulas. Many of these are similar or identical to Excel functions.

First, you can use Excel functions in VBA, but when you do, you have to precede the function name by `Application` or `WorksheetFunction` followed by a period (.). For example, to use Excel's `Average` function in VBA you will use `Application.WorksheetFunction.Average`, or `Application.Average`, or `WorksheetFunction.Average` (I prefer this one!).

Second, you cannot use an Excel worksheet function in VBA if VBA has an equivalent function. To see a list of all the Excel functions you can use in VBA, search in VBA Help under “Function” and then choose “List of Worksheet Functions Available to Visual Basic.” You can also type `WorksheetFunction` followed by a period (.) anywhere in a module. In the list that pops up, functions are preceded by green icons.

(See Excel file “Appendix C Excel and VBA Built-In Functions.xlsx”).

6. Stop, Exit, End function

By itself, an `End` statement put anywhere in a code will immediately terminate execution of the code. For example, you can use it to terminate execution if a particular type of error occurs during execution. The `End` statement closes all open files, resets all variables, and so forth, so that execution cannot be restarted from the point where it ended.

You use an `Exit` statement to exit from the middle of a block of `Do . . . Loop`, `For . . . Next`, `Function`, `Sub` or `Property` code blocks most often because a particular condition has been satisfied. VBA offers the following `Exit` statements: `Exit Do`, `Exit For`, `Exit Function`, `Exit Property`, and `Exit Sub` to exit the different types of code blocks.

You can use a `Stop` statement instead to suspend execution (for example, to check the current values of some variables) and then you can restart execution by clicking the `Run Sub` button on the toolbar. The `Stop` statement is similar to setting a breakpoint in the code except that you build it into the code.

CHAPTER 4. SUB PROCEDURE AND FUNCTION PROCEDURE

A procedure is a series of VBA statements that resides in a VBA module, which you access in the Visual Basic Editor (VBE). A module can hold any number of procedures. A procedure holds a group of VBA statements that accomplishes a desired task. Most VBA code is contained in procedures.

You have a number of ways to `call`, or execute, procedures. A procedure is executed from beginning to end, but it can also be ended prematurely.

Some procedures are written to receive arguments. An argument is information that is used by the procedure and that is passed to the procedure when it is executed. Procedure arguments work much like the arguments that you use in Excel worksheet functions. Instructions within the procedure perform operations using these arguments, and the results of the procedure are usually based on those arguments.

A Sub procedure can perform many actions but **cannot return a value**. This is an important difference with Function procedures.

1. Sub Procedure

A simplified structure looks like this:

```
[Private|Public] [Static] Sub name([arglist])  
    [Statements]  
[Exit Sub]  
    [Statements]  
End Sub
```

Here's a description of the elements that make up a Sub procedure:

- **Private:** Optional. Indicates that the procedure is accessible only to other procedures in the same module.
- **Public:** Optional. Indicates that the procedure is accessible to all other procedures in all other modules in the workbook. If used in a module that contains an Option Private Module statement, the procedure is not available outside the project (other workbooks or Microsoft Office applications that may attempt to call the procedures in the module).
- **Static:** Optional. Indicates that the procedure's variables are preserved when the procedure ends.

- **Sub:** Required. The keyword that indicates the beginning of a procedure.
- **name:** Required. Any valid procedure name.
- **arglist:** Optional. Represents a list of variables, enclosed in parentheses, that receive arguments passed to the procedure. Use a comma to separate arguments. If the procedure uses no arguments, a set of empty parentheses is required.
- **Statements:** Optional. Represents valid VBA statements.
- **Exit Sub:** Optional. Forces an immediate exit from the procedure prior to its formal completion.
- **End Sub:** Required. Indicates the end of the procedure.

1.1 Naming procedures

Every procedure must have a name. The rules governing procedure names are generally the same as those for variable names. Ideally, a procedure's name should describe what its contained processes do. A good rule is to use a name that includes a verb and a noun (for example, `ProcessDate`, `PrintReport`, or `Sort_Array`).

1.2 Scoping a Procedure

In Chapter 3, we noted that a variable's scope determines the modules and procedures in which you can use the variable. Similarly, a procedure's scope determines which other procedures can call it.

Public procedures

By default, procedures are public procedures; that is, they can be called by other procedures in any module in the workbook. It's not necessary to use the `Public` keyword, but programmers often include it for clarity.

Private procedures

Private procedures can be called by other procedures in the same module but not by procedures in other modules.

The following example declares a private procedure named `MySub`:

```
Private Sub MySub()  
    ' ... [code goes here] ...  
End Sub
```

Excel's macro recorder creates new Sub procedures called `Macro1`, `Macro2`, and so on. Unless you modify the recorded code, these procedures are all public procedures, and they will never use any arguments.

1.3 Executing a Procedure from Another Procedure

Sub procedures that do not have any argument (in `Procedure_Name([arglist])`) can be run directly. If we include arguments, these arguments need to be passed to Sub procedure, so this kind of Sub procedure cannot run independently, but should be run by **calling them** with arguments from other procedures.

You have two ways to do this.

- Enter the procedure's name, followed by its arguments (if any) separated by commas. Do not enclose the argument list in parentheses.
- Use the `Call` keyword followed by the procedure's name and then its arguments (if any) enclosed in parentheses and separated by commas. The `Call` keyword is technically optional, as you don't need it to run the specified procedure. However, many Excel developers still use it as a clear indicator that another procedure is being called.

Here's a simple Sub procedure that takes two arguments. The procedure displays the product of the two arguments.

Example:

```
Sub Caller()
    Dim R1, R2
    R1 = 10
    R2 = 20
    Call Called(R1, R2)
    '    Called R1, R2
    MsgBox R1
End Sub

Sub Called(D1, D2)
    Dim Total
    D1 = D1 * 10
    Total = D1 + D2
    Worksheets("Sheet1").Cells(3, 5).Value = Total
End Sub
```

In `Called`, `D1` and `D2` are the arguments. It takes the value passed from `Caller`. The value of the arguments assigned in `Caller` is `R1 = 10` and `R2 = 20`.

Notes:

- Called Sub procedure must also be a complete procedure.

- Arguments D1 and D2 are NOT variables that you need to declare in the called Sub procedure.
- If the called Sub procedure has 2 arguments, then the calling statement must also provide the same number of arguments.
- The names of argument being passed (R1 and R2) do not require having the same names in the called sub procedure (D1 and D2).
- We are allowed to specify the data types of the arguments of a Sub procedure; you need to do so in its declaration line. In our example, you can use

```
Sub Called(D1 as Integer, D2)
```

If we don't specify the data types, D1 and D2 are determined by the data type of the arguments being passed (R1 and R2).

If you call a procedure from another procedure then once control is returned to the calling procedure, the values of the procedure-level variables in the called procedure will not be available anymore, for example, for use during the next call.

Also, you can use the same variable names for procedure-level variables in different procedures within a module or different procedure of the same project (in one or more modules). However, for the sake of clarity, you should generally avoid such use.

1.4 Passing Arguments to Procedures

A procedure's arguments provide it with data that it uses in its instructions. The data that's passed by an argument can be any of the following:

- A variable
- A constant
- An expression
- An array
- An object

You can pass an argument to a procedure in two ways.

- **By reference:** Passing an argument by reference passes the memory address of the variable. Changes to the argument within the procedure are made to the original variable. This is the default method of passing an argument.
- **By value:** Passing an argument by value passes a copy of the original variable. Consequently, changes to the argument within the procedure are not reflected in the original variable.

The following example demonstrates this concept.

A simple example will illustrate this clearly:

```
Sub CallingProcedure()
```

```
    Dim A As Long
    Dim B As Long
    A = 123
    B = 456
    Call CalledProcedure(A, B)
    Debug.Print A
    Debug.Print B
End Sub

Sub CalledProcedure(ByRef X As Long, ByVal Y As Long)
    X = 321
    Y = 654
End Sub
```

(See Lec4_SubFunction.xlsm, Module A_CallSub)

In the `CallingProc`, the variables `A` and `B` are assigned the values 123 and 456, respectively. Then, the `CalledProc` is called passing arguments `A` to `X` and `B` to `Y`. Within `CalledProc` the parameters `X` and `Y` are assigned the values 321 and 654 respectively, and control is returned to the procedure `CallingProc`.

Since the parameter `X` was declared with `ByRef`, a reference to `A` was passed to `CalledProc` and any modification to the `X` parameter in `CalledProc` affects the variable `A` in `CallingProc`. Now `A` has value 321 (updated).

The parameter `Y` was declared with `ByVal`, so only the actual value of `B` was passed to `CalledProc`. Changes made to the parameter `Y` are not made to the variable `B`. Now, the `B` still has value 456 (not updated).

2. Function Procedure

A VBA Function is a procedure that performs calculations and returns a value. You can use these functions in your Visual Basic for Applications (VBA) code or in worksheet formulas.

VBA enables you to create `Sub` procedures and `Function` procedures. You can think of a `Sub` procedure as a command that either the user or another procedure can execute.

`Function` procedures, on the other hand, usually return a single value (or an array), just like Excel worksheet functions and VBA built-in functions. As with built-in functions, your `Function` procedures can use arguments.

`Function` procedures are versatile, and you can use them in two situations.

- As part of an expression in a VBA procedure
- In formulas that you create in a worksheet

In fact, you can use a `Function` procedure anywhere you can use an Excel worksheet function or a VBA built-in function. The only exception is that you can't use a VBA function in a data validation formula. You can, however, use a custom VBA function in a conditional formatting formula.

2.1 An Introductory Function Example

Without further ado, this section presents an example of a VBA Function procedure. The following is a custom function defined in a VBA module. This function, named `TestSumSqr(a, b As Single)`

```
Function TestSumSqr(a, b As Single)
    a = a ^ 2
    b = b ^ 2
    TestSumSqr = a + b
End Function
```

This function certainly isn't the most useful function, but it demonstrates some key concepts related to functions.

Function procedures can be as complex as you need them to be. Most of the time, they're more complex and much more useful than this sample procedure. Nonetheless, an analysis of this example may help you understand what is happening.

The procedure starts with the keyword `Function`, rather than `Sub`, followed by the name of the function (`TestSumSqr`). This custom function uses two arguments (`a, b`), enclosed in parentheses. Argument `a` has data type of `Variant`, and `b` has data type of `Single`. Excel uses the `Variant` data type if no data type is specified. The returned value of the function has data type of `Variant`.

This procedure does not use additional variables. So `Dim` statement, which declares the variable used in the procedure is not included in this procedure.

Before the procedure ends, we assigned calculation results to the name of the `Function` procedure. In this procedure, we return a numeric value to `TestSumSqr`. In another function procedure `ArrayOut`, we assign an array variable to the function name, then it is an array function.

2.2 Structure of Function Procedures

Structure of a Function procedure looks like this:

```
[Private|Public] [Static] Function name([arglist]) [As type]
    [Statements]
    [name = expression]
    [Exit Function]
    [Statements]
    [name = expression]
End Function
```

Here's a description of the elements that make up a Function procedure:

- **Private:** Optional. Indicates that the Function procedure is accessible only to other procedures in the same module.
- **Public:** Optional. Indicates that the Function procedure is accessible to all other procedures in all other modules in the workbook.
- **Static:** Optional. Indicates that the values of variables declared in the Function procedure are preserved between calls.
- **Function:** Required Indicates the beginning of a procedure that returns a value or other data.
- **name:** Required. Any valid procedure name.
- **arglist:** Optional. Represents a list of variables, enclosed in parentheses, that receive arguments passed to the procedure. Use a comma to separate arguments. If the procedure uses no arguments, a set of empty parentheses is required.
- **type:** Optional. The data type returned by the Function procedure.
- **Statements:** Optional. Represents valid VBA statements.
- **Exit Function:** Optional. Forces an immediate exit from the procedure prior to its formal completion.
- **End Function:** Required. Indicates the end of the procedure.

Notes about function's name:

- Don't give your defined function the same name as other VBA functions (such as NPV, RATE). Otherwise, user-defined function will overwrite VBA function. (But for Excel built-in function, such as SUM, there is not conflict. Because in VBA, you actually use `WorksheetFunction.SUM`.)

For more details about VBA Built-In Functions, please refer to the Excel file "**Appendix C Excel and VBA Built-In Functions.xls**" on Moodle.

- Don't give different functions the same name in your project (if the functions are project-level functions). When there are repeated function names, there will be error dialog window, warning "ambiguous names".
- Don't give your Function procedure the same name as other functions (including user-defined and VBA functions).
- Don't give your variables the same name as `Sub` and `Function` procedures.

A key point to remember about a custom function written in VBA is that a value is always assigned to the function's name a minimum of one time, generally when it has completed execution.

Using the Function in a Worksheet

When you enter a formula that uses the `TestSumSqr` function, Excel executes the code to get the result that's returned by the function. The function works like any built-in worksheet function.

Using the Function in a VBA Procedure

In addition to using custom functions in worksheet formulas, you can use them in other VBA procedures.

You can call custom functions from a VBA procedure the same way that you call built-in functions. For example, after you define a function called `TestSumSqr`, you can enter a statement like the following:

```
Total = TestSumSqr(var1, var2)
```

This statement executes the `TestSumSqr` function with `var1` and `var2` as its arguments, returns the function's result, and assigns it to the `Total`.

A Function with an Array Argument

A Function procedure also can accept one or more arrays as arguments, process the array(s), and return a single value. The array can also consist of a range of cells.

The following function accepts an array as its argument and returns the sum of its elements:

```
Function SumArray(List) As Double
    Dim Item As Variant
    SumArray = 0
    For Each Item In List
        If WorksheetFunction.IsNumber(Item) Then _
```

```
        SumArray = SumArray + Item
    Next Item
End Function
```

Excel built-in function `IsNumber` checks to see whether each element is a number before adding it to the total. Adding this simple error-checking statement eliminates the type-mismatch error that occurs when you try to perform arithmetic with something other than a number.

The following procedure demonstrates how to call this function from a sub procedure. The `MakeList` procedure creates a 100-element array and assigns a random number to each element.

Then the `MsgBox` function displays the sum of the values in the array by calling the `SumArray` function.

```
Sub MakeList()
    Dim Nums(1 To 100) As Double
    Dim i As Integer
    For i = 1 To 100
        Nums(i) = Rnd * 1000
    Next i
    MsgBox SumArray(Nums)
End Sub
```

Note that the `SumArray` function doesn't declare the data type of its argument (it's a `Variant` type). Because it's not declared as a specific numeric type, the function also works in your worksheet formulas in which the argument is a `Range` object. For example, the following formula returns the sum of the values in A1:C10:

```
= SumArray(A1:C10)
```

You might notice that, when used in a worksheet formula, the `SumArray` function works very much like Excel's `SUM` function. One difference, however, is that `SumArray` doesn't accept multiple arguments. Understand that this example is for educational purposes only. Using the `SumArray` function in a formula offers no advantages over the Excel `SUM` function.

A Function that Returns a VBA Array

VBA includes a useful function called `Array`. The `Array` function returns a variant that contains an array (that is, multiple values). If you're familiar with array formulas in Excel, you have a head start on understanding VBA's `Array` function. You enter an array formula

into a cell by pressing **Ctrl+Shift+Enter**. Excel inserts curly braces around the formula to indicate that it's an array formula.

The `MonthNames` function, which follows, is a simple example that uses VBA's `Array` function in a custom function:

```
Function MonthNames()  
    MonthNames = Array("Jan", "Feb", "Mar", "Apr", "May", _  
        "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")  
End Function
```

The `MonthNames` function returns a horizontal array of month names. You can create a multicell array formula that uses the `MonthNames` function. Here's how to use it:

1. Make sure that the function code is present in a VBA module.
2. In a worksheet, select multiple cells in a row (start by selecting 12 cells).
3. Enter the formula that follows (without the braces) and press **Ctrl+Shift+Enter**.
`{=MonthNames() }`

What if you'd like to generate a vertical list of month names? No problem; just select a vertical range, enter the following formula (without the braces), and then press **Ctrl+Shift+Enter**:

```
{=TRANSPOSE(MonthNames()) }
```

This formula uses the Excel `TRANSPOSE` function to convert the horizontal array to a vertical array.

(See Function procedure `MonthNames`, `ArrayOut` in Module `C_FnArray`.)

A Function that Returns an Error Value

In some cases, you might want your custom function to return a particular error value. Consider the following `Factorial` function:

```
Function Factorial(n)  
    Dim fac, i  
    If n < 0 Then  
        Factorial = CVErr(xlErrNum)  
        Exit Function  
    End If  
    fac = 1  
    For i = 1 To Int(n)  
        fac = i * fac  
    Next i  
End Function
```



```
Next  
    Factorial = fac  
End Function
```

This function calculates the factorial of a non-negative integer. If the argument is negative, this function returns a #NUM! error.

To return a real error value from a function, use the VBA `CVErr` function, which converts an error number to a real error.

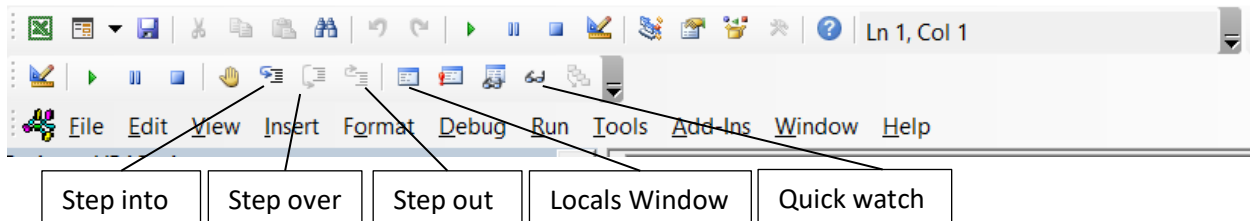
(See Function procedure `Factorial` in Module `B_FnBasic`.)

APPENDIX 1. DEBUGGING TOOL—LOCALS WINDOW, QUICK WATCH, STEP INTO/OVER/OUT

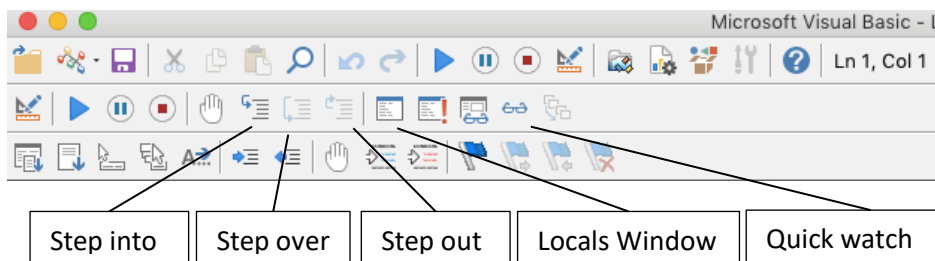
When you debug your sub procedures, one of the most useful methods is to **step through** a program one line at a time, possibly keeping a **watch** on one or more key variables.

VBE's **Debug toolbar** is very handy for doing this.

Windows VBE Debug Toolbar:



VBE Debug Toolbar in Excel for Mac:



If you want to run a sub procedure line by line, you click anywhere inside the program, then instead of pressing Run macro button, click **Step into** button. If the program calls another sub or function procedure, you will also step through that procedure line by line.

If you do not want to get into the called sub/function, you can click **Step over** button.

If you decide there is no point in stepping through the rest of code, click **Step out** button.

If you want to put a watch on a key variable, you put the cursor anywhere on the variable (anywhere it appears in the code) and click the **Quick watch** button.

The **Watch window** opens automatically. You can put watches on as many variables as you like. If the sub also calls other sub or function procedures, variables in those called procedures can also be watched. After the sub procedure is executed, the watched variables

are still listed in the Watch window, but values are reset. You can only check the value when the procedure is still running.

Locals Window automatically displays all the declared variables in the current procedure and their values. When the Locals window is visible, it is automatically updated every time there is a change from run time to break mode or you navigate in the stack display. The window is emptied when the procedure is finished.

To open the Locals window, go to View ⇒ Locals Window, or click Locals Window button on the Debug toolbar.

APPENDIX 2. SIMPLE DEBUGGING

Finding and fixing errors in your code, called debugging, can be a simple to a very time-consuming and frustrating process because errors come in bewildering varieties. Here are some simple debugging methods:

- Insert one or a few **MsgBox** functions at strategic points in the procedure to display the values of a few key variables at certain points in your procedure, or to report errors if you get undesired result.
- Insert one or a few **Debug.Print** statements at strategic points in the procedure to display the values of a few key variables at certain points in your procedure in the Immediate Window. To open it from VBE window, go to View ⇨ Immediate Window.
- In the VBE, put your pointer in the gray margin (the left edge of the code window) on the left of a statement at a strategic point in the procedure and click. This will set a **breakpoint** in the code at that statement, which will be indicated by a dark circle in the margin and the statement itself will be highlighted. Now when you run the procedure, execution will be suspended when it gets to the breakpoint and you will be able to see the current value of any variable that has been read in or calculated so far by resting your cursor on it. Multiple breakpoints can be set in a procedure. To remove the breakpoint, point at the dark circle in the left margin and click.
Note: Breakpoints cannot be added in comment lines or `Dim` statement lines.
- You can also use a **Stop** statement to suspend execution, and then you can restart execution by clicking the Run Sub button on the toolbar.
- When you debug your sub procedures, you can also **step into** a program and execute one line at a time, possibly keeping a **watch** on one or more key variables. VBE's Debug toolbar provides very handy tools for doing this.
(See APPENDIX 1. DEBUGGING TOOL: LOCALS WINDOW, QUICK WATCH, AND STEP INTO/OVER/OUT)

MsgBox Function

The `MsgBox` function displays a dialog box and offers a simple way to provide the user a message and some short outputs, often to show intermediate results and to get simple user responses, if desired. It is also a useful debugging tool because you can insert `MsgBox` functions at strategic points in your code to see intermediate values of variables that can help you track down a problem.

The syntax of the `MsgBox` function is:

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

The argument `prompt` is required. It is the message you want displayed. The `buttons` argument, which is optional, allows you to specify which buttons (for example, OK) you want your dialog box to show to get the user's response. The optional `title` argument lets you specify a title for the dialog box. You will probably never use the other two optional arguments.

The `InputBox` also displays a dialog box in which you can provide a message, most often to explain what kind of input the user should provide in the text box. The result of the `InputBox` function is the input the user provides.

The syntax of the `InputBox` function is:

```
InputBox(prompt[, title] [, default] [, xpos] [, ypos] [, helpfile, context])
```

The required argument `prompt` is the message you want to display. The optional argument `title` lets you specify a title for the dialog box and the optional argument `default` lets you specify a string that will be displayed in the text box as a default option. (The user can accept it or overwrite it.) You are not likely to use the other arguments.

`Msgbox` and `Inputbox` are both VBA built-in functions. In VBA, function procedures are used to return a value. For example, `Inputbox` function return the value that you enter in the dialog box.

`MsgBox` function is very special. It does return a value, which is the value of the button that you select. But it also creates an action, which is to create a dialog box and present a message. When we use `MsgBox` function, we usually care about the message box only, but not the value returned. That is why you see codes like:

```
MsgBox "Interest expense is $" & intExpense
```

How is a function called in VBA? We can call it using `Call` statement ([check this website](#)).

The syntax of `Call` statement is [**Call**] *name* [*argumentlist*]. `Call` can be omitted. If you explicitly write `Call` in front of the function, arguments of the function must be included in the parentheses. For example:

```
Call MsgBox("Interest expense is $" & intExpense)
Call MsgBox("Interest expense is $" & intExpense, vbOKOnly,
"Result")
```

If you don't write `Call` explicitly and don't save the returned value of `MsgBox`, parentheses outside `MsgBox` function's arguments is not necessary:

```
MsgBox "Interest expense is $" & intExpense, vbOKOnly,  
"Result"
```

You can also include the name of the function's arguments and assign value to the arguments using operator `:=`.

```
MsgBox Prompt:="Interest expense is $" & intExpense,  
Buttons:=vbOKOnly, Title:="Result"
```

Last, if you want to store the returned value of `MsgBox` function, you can store it in a defined variable:

```
msgReply = MsgBox("Do you want to increase the interest rate  
of " & intRate * 100 & "% by 1%?", vbYesNo)
```

APPENDIX 3. VBA Arrays and Worksheet Ranges

Data transfer between worksheet cells and VBA variables is an expensive operation that should be kept to a minimum. You can considerably increase the performance of your Excel application by passing arrays of data to the worksheet, and vice versa, in a single operation rather than one cell at a time. If you need to do extensive calculations on data in VBA, you should transfer all the values from the worksheet to an array, do the calculations on the array, and then, possibly, write the array back to the worksheet. This keeps the number of times data is transferred between the worksheet and VBA to a minimum. It is far more efficient to transfer one array of 100 values to the worksheet than to transfer 100 items at a time.

This appendix shows you how to transfer data between worksheet ranges and VBA arrays. You will find that with large amounts of data being transferred between the worksheet and the array, working with the array is much faster than working directly with worksheet cells.

Reading a Worksheet Range to a VBA Array

It is very simple to read a range on a worksheet and put it into an array in VBA. For example,

```
Dim Arr()                'declare an unallocated array
Arr = Range("A1:C5")     'Arr is now an allocated array
```

When you bring in data from a worksheet to a VBA array, the array is always 2 dimensional. The first dimension is the rows and the second dimension is the columns. So in the example above, `Arr` is implicitly sized as `Arr(1 To 5, 1 To 3)` where 5 is the number of rows and 3 is the number of columns. A 2D array is created even if the worksheet data is in a single row or a single column (e.g, `Arr(1 To 10, 1 To 1)`). The array into which the worksheet data is loaded always has a lower bound equal to 1, regardless of what `Option Base` directive you may have in your module. You cannot change this behavior. For example,

```
Dim Arr()
Arr = Range("A1:A10")
```

Here `Arr` is dimensioned automatically by VBA as `Arr(1 to 10, 1 To 1)`.

Writing a One Dimensional VBA Array to the Worksheet

Once you have calculated an array with the appropriate values, you can write it back to the worksheet. The array may be 1 or 2 dimensional.

A 1D array can be written back to a row vector in the worksheet.

```
Dim Arr(1 to 10)
For i = LBound(Arr) To UBound(Arr)
    Arr(i) = i
Next i
Range("A1:J1").Value = Arr
```

To write 1D array back to a column vector in the worksheet, you must transpose the array variable first.

```
Range("A1:A10") = WorksheetFunction.Transpose(Arr)
```

You can create a Range object, resize that range to the size of your array, and then write to the range.

Suppose we have a 1D array and want to write that out to the worksheet starting at cell K1. The code must first resize the destination range. For example,

```
' one row spanning several columns
Dim Destination As Range
Set Destination = Range("K1")
Set Destination = Destination.Resize(1, UBound(Arr))
Destination.Value = Arr
```

This code will write the values of Arr to range that is one row by UBound(Arr) columns, starting at range K1.

Range.Resize property:

<https://docs.microsoft.com/en-us/office/vba/api/excel.range.resize>

If you want the results passed to a range that is one column wide spanning several rows, use code like the following to resize the range and set the values.

```
Dim Destination As Range
Set Destination = Range("K1")
Set Destination = Destination.Resize(UBound(Arr), 1)
Destination.Value = WorksheetFunction.Transpose(Arr)
```


NOTE that the parameters to `Resize` are reversed and that the array `Arr` is transposed before being written to the worksheet.

Writing a Two-Dimensional VBA Array to the Worksheet

If you have a 2D array, you need to use `Resize` to resize the destination range to the proper size. The first dimension is the number of rows, and the second dimension is the number of columns. The code below illustrates writing an array `Arr` out to the worksheet starting at cell K1.

```
Dim Destination As Range
Set Destination = Range("K1")
Destination.Resize(UBound(Arr, 1), UBound(Arr, 2)).Value =
Arr
```

You can transpose the array when writing to the worksheet:

```
Set Destination = Range("K1")
Destination.Resize(UBound(Arr, 2), UBound(Arr, 1)).Value =
Application.Transpose(Arr)
```

Here, the parameters to `Resize` are reversed and the array `Arr` is transposed.

Array Sizing

When you read from a worksheet to an array variable, VBA will automatically size the array to hold the range on the worksheet. You don't have to concern yourself with sizing the array. However, when writing an array from VBA to the worksheet, you must resize the destination range to hold the array. We saw this earlier in the examples. Basically, you use code like the following.

```
Dim NumRows As Long
Dim NumCols As Long
NumRows = UBound(Arr, 1) - LBound(Arr, 1) + 1
NumCols = UBound(Arr, 2) - LBound(Arr, 2) + 1
Set Destination = Range("K1").Resize(NumRows, NumCols).Value =
Arr
```

If the array being passed to the worksheet is smaller than the Range to which it is written, the unused cells get a #N/A error. If the array being passed is larger than the range to which it is written, the array is truncated on the right or bottom to fit the range.

As you've seen in the examples, passing array between the worksheet and VBA is simple. Used correctly, the code snippets above can have a strong effect on increasing the performance of your VBA application.

(See Sub procedure `Data_v1`, `Data_v2`, `Data_v3` in Module `F_ReadData`.)

APPENDIX 4. ERROR HANDLING

Source:

Handle Run-Time Errors: <https://docs.microsoft.com/en-us/office/vba/access/concepts/error-codes/elements-of-run-time-error-handling>

On Error Statement: <https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/on-error-statement>
<http://www.cpearson.com/Excel/ErrorHandling.htm> (This note is from this website)

Trappable errors in VBA: [https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2008/ms234761\(v=vs.90\)](https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2008/ms234761(v=vs.90))

(Or, in VBE window -- "Help", search "trappable error", "on error statement")

Error handling refers to the programming practice of anticipating and coding for error conditions that may arise when your program runs. Errors in general come in three flavors:

(1) **Compiler errors** such as undeclared variables that prevent your code from compiling, spell something wrong, omit a keyword, or commit various other "grammatical" errors.

Such errors are usually easy to spot because the VBE typically detects them immediately, colors them in red, and displays a warning in a message box.

(2) **User data entry error** such as a user entering a negative value where only a positive number is acceptable;

(3) **Run time errors** that occur when VBA cannot correctly execute a program statement. The error is not discovered until you run the code.

We will concern ourselves here only with run time errors. Typical run time errors include attempting to access a non-existent worksheet or workbook, or attempting to divide by zero (Error 11).

Your application should make as many checks as possible during initialization to ensure that run time errors do not occur later. In Excel, this includes ensuring that required workbooks and worksheets are present and that required names are defined. The more checking you do before the real work of your application begins the more stable your application will be. It is far better to detect potential error situations when your application starts up before data is change than to wait until later to encounter an error situation.

If you have no error handling code and a run time error occurs, VBA will display its standard run time error dialog box. While this may be acceptable, even desirable, in a development environment, it is not acceptable to the end user in a production

environment. The goal of well-designed error handling code is to anticipate potential errors, and correct them at run time or to terminate code execution in a controlled, graceful method. Your goal should be to prevent unhandled errors from arising.

The On Error Statement

The heart of error handling in VBA is the On Error statement. This statement instructs VBA what to do when a run time error is encountered. The `On Error` statement takes three forms.

```
On Error Goto 0
On Error Resume Next
On Error Goto <label>:
```

The first form, `On Error Goto 0`, is the default mode in VBA. This indicates that when a run time error occurs VBA should display its standard run time error message box, allowing you to enter the code in debug mode or to terminate the VBA program. When `On Error Goto 0` is in effect, it is the same as having no enabled error handler. Any error will cause VBA to display its standard error message box.

The second form, `On Error Resume Next`, is the most commonly used and misused form. It instructs to VBA to essentially ignore the error and resume execution on the next line of code. It is very important to remember that `On Error Resume Next` does not in any way "fix" the error. It simply instructs VBA to continue as if no error occurred. However, the error may have side effects, such as uninitialized variables or objects set to Nothing. It is the responsibility of your code to test for an error condition and take appropriate action. You do this by testing the value of `Err.Number` and if it is not zero execute appropriate code. For example,

```
On Error Resume Next
N = 1 / 0      ' cause an error
If Err.Number <> 0 Then
    N = 1
End If
```

This code attempts to assign the value `1 / 0` to the variable `N`. This is an illegal operation, so VBA will raise an error 11 -- Division By Zero -- and because we have `On Error Resume Next` in effect, code continues to the `If` statement. This statement tests the value of `Err.Number` and assigns some other number to `N`.

The third form `On Error` of is `On Error Goto <label>:` which tells VBA to transfer execution to the line following the specified line label. Whenever an error occurs, code

execution immediately goes to the line following the line label. None of the code between the error and the label is executed, including any loop control statements.

```
On Error Goto ErrorHandler:
N = 1 / 0      ' cause an error
'
' more code
'
Exit Sub
ErrorHandler:
' error handling code
Resume Next
End Sub
```

APPENDIX 5. RETURNING ERRORS FROM VBA FUNCTIONS

If you use VBA to create User Defined Functions (functions that are called directly from worksheet cells) in a module or add-in, you likely will need to return an error value under some circumstances. For example, if a function requires a positive number as a parameter and the user passes in a negative number, you should return a #VALUE error. You might be tempted to return a text string that looks like an error value, but this is not a good idea. Excel will not recognize the text string, for example #VALUE, as a real error, so many functions and formulas may misbehave, especially ISERROR, ISERR, and IFERROR, and ISNA. These functions require a real error value.

VBA provides a function called CVerR that takes a numeric input parameter specifying the error and returns a real error value that Excel will recognize as an error. The values of the input parameter to CVerR are in the XLcVerR and are as follows:

- xlErrDiv0 (= 2007) returns a #DIV/0! error.
- xlErrNA (= 2042) returns a #N/A error.
- xlErrName (= 2029) returns a #NAME? error.
- xlErrNull (= 2000) returns a #NULL! error.
- xlErrNum (= 2036) returns a #NUM! error.
- xlErrRef (= 2023) returns a #REF! error.
- xlErrValue (= 2015) returns a #VALUE! error.

The only legal values of the input parameter to CVerR function are those listed above. Any other value causes CVerR to return a #VALUE. This means, unfortunately, that you cannot create your own custom error values. In order to return an error value, the function's return data type must be a Variant. If the return type is any other data type, the CVerR function will terminate VBA execution and Excel will report a #VALUE error in the cell. Note that these errors are meaningful only to Excel and have nothing at all to do with the Err object used to work with runtime errors in VBA code.

The following is an example using CVerR.

```
Function Test(D As Double) As Variant
    If D < 0 Then
        Test = CVerR(xlErrValue)
    Else
        Test = D * 10
    End If
End Function
```

This function will return a #VALUE! error if the input parameter is less than 0. Note that the return type of the function is Variant.

You can also use CErr to test whether a cell has a specific error value in it. However, you must first test whether the cell contains any sort of error, and then, if it does contain an error, test which type of error. For example,

```
Dim R As Range
Set R = Range("A1")
If IsError(R.Value) = True Then
    If R.Value = CErr(xlErrValue) Then
        Debug.Print "#VALUE error"
    Else
        Debug.Print "Some other error"
    End If
End If
```

If you attempt to compare a cell's value to a value produced by CErr, and the cell does not contain an error value, you will get a run-time error 13, Type Mismatch. For example, the following code will fail if A1 does not contain an error value.

```
Dim R As Range
Set R = Range("A1")
If R.Value = CErr(xlErrValue) Then 'error 13 if A1 has no
error
    Debug.Print "#VALUE error"
End If
```

You can use CErr in a Select Case statement to test the various error types. For example,

```
Dim R As Range
Set R = Range("A1")
If IsError(R.Value) = True Then
    Select Case R.Value
        Case CErr(xlErrValue)
            Debug.Print "#VALUE error"
        Case CErr(xlErrDiv0)
            Debug.Print "#DIV/0 error"
        Case CErr(xlErrName)
            Debug.Print "#NAME? error"
        Case Else
            Debug.Print "Some other error"
    End Select
End If
```

(From <http://www.cpearson.com/excel/ReturningErrors.aspx>)