Microsoft

Microsoft Excel VBA and Macros

Your guide to efficient automation

Tracy Syrstad and Bill Jelen

Data sets and code files on the web

FREE SAMPLE CHAPTER

Microsoft

Microsoft Excel VBA and Macros: Your guide to efficient automation

Tracy Syrstad
Bill Jelen

Microsoft Excel VBA and Macros: Your guide to efficient automation

Published with the authorization of Microsoft Corporation by:

Pearson Education, Inc.

Hoboken, New Jersey

ISBN-13: 978-0-13-541023-3

ISBN-10: 0-13-541023-1

Library of Congress Control Number: 2025943857

$PrintCode

Trademarks

Warning and Disclaimer

EXECUTIVE EDITOR

Loretta Yates

ACQUISITIONS EDITOR

Shourav Bose

DEVELOPMENT EDITOR

Rick Kughen

MANAGING EDITOR

Sandra Schroeder

SENIOR PROJECT EDITOR

Tracey Croom

COPY EDITOR

Rick Kughen

INDEXER

Rachel Kuhn

PROOFREADER

Donna E. Mulder

TECHNICAL EDITOR

Bob Umlas

COVER DESIGNER

Twist Creative, Seattle

COMPOSITOR

codeMantra

COVER ILLUSTRATION

ImageFlow/Shutterstock

Dedication

For Tracy Syrstad. Thanks for being a great coauthor for 20 years!

–Bill Jelen

To the clients who made this book possible—Marlee Jo J., Dale W., Eddie G., and all the others over the years—thank you for the challenging projects, sharp questions, and unexpected curveballs. You didn't just keep me on my toes—you made me better.

–Tracy Syrstad

Contents at a Glance

Contents

Acknowledgments

They say it takes an army to get a celebrity ready for the red carpet—stylists, coaches, and a whole lot of behind-the-scenes magic. Writing this book wasn't so different. I had my own team making sure the seams didn't show.

To Loretta: The calm in the storm, the keeper of continuity, and the person who managed to steer this whole thing without ever once raising her voice—at least not where I could hear it.

To Bill Jelen: Thank you for opening the door, showing me the path, and walking it with me. You've made this journey possible.

To Robert L., for keeping me fueled with cookies; to Chris H., for giving me the stage to share my VBA bag of tricks; to Geoff W., for tossing me puzzles that demanded real out-of-the-box thinking; and to Ash—technically, a client but really a collaborator—for helping me think in versions, not just answers. Your clarity cut through my fog every time. Consulting has introduced me to people across every kind of industry, and it's the variety—the weird bugs, the clever fixes, the unexpected questions—that make this work so much fun.

To Icarus, MeltedCake, and Chilly: You reminded me that "It wasn't a phase, Mom" can be a way of life. Thanks for the chaos, the loyalty, and the perfectly timed distractions.

To Jareth, Luna, Sam, and Anita: Your daily antics are worth the 2 a.m. feedings and zoomies.

And to John: Thank you for being my steady in the storm, from the moment I left the corporate grind and started building this strange, wonderful path of my own.

—Tracy

At Pearson, Loretta Yates is an excellent executive editor. Thanks to Rick Kughen for guiding this book through production.

Along the way, I've learned a lot about VBA programming from the awesome community at the MrExcel.com message board. VoG, Richard Schollar, and Jon von der Heyden all stand out as having contributed posts that led to ideas in this book. Thanks to Pam Gensel for Excel macro lesson #1. Mala Singh taught me about creating charts in VBA. Suat Özgür keeps me current on new VBA trends and contributed many ideas to Chapter 18.

My family was incredibly supportive during this time. Thanks to Mary Ellen Jelen.

—Bill

This page intentionally left blank

About the Authors

Tracy Syrstad is the author of 11 Excel books and a veteran Excel developer with decades of experience building custom Excel solutions for businesses and individuals. Since 1997, she's helped clients streamline workflows, automate reporting, and tame messy data using the full power of Excel and VBA. Her writing reflects the same approach she brings to her consulting—clear, practical, and focused on helping real people solve real problems.

Bill Jelen, Excel MVP and the host of MrExcel.com, has been using spreadsheets since 1985, and he launched the MrExcel.com website in 1998. Bill was a regular guest on *Call for Help* with Leo Laporte and has produced more than 2,500 episodes of his video podcast, *Learn Excel from MrExcel*. He is the author of 71 books about Microsoft Excel and writes the monthly Excel column for *Strategic Finance* magazine. Before founding MrExcel.com, Bill spent 12 years in the trenches—working as a financial analyst for finance, marketing, accounting, and operations departments of a $500 million public company. He lives in Merritt Island, Florida, with his wife, Mary Ellen.

This page intentionally left blank

Introduction

In this Introduction, you will:

- Find out what is in this book.
- Have a peek at the future of VBA and Windows versions of Excel.
- Learn about special elements and typographical conventions in this book.
- Learn where to find the code files for this book.

As corporate IT departments have found themselves with long backlogs of requests, Excel users have discovered that they can produce the reports needed to run their businesses themselves using the macro language *Visual Basic for Applications* (VBA). VBA enables you to achieve tremendous efficiencies in your day-to-day use of Excel. VBA helps you figure out how to import data and produce reports in Excel so that you don't have to wait for the IT department to help you.

Is TypeScript a threat to VBA?

Your first questions are likely: "Should I invest time in learning VBA? How long will Microsoft support VBA? Will the TypeScript language released for Excel Online replace VBA?"

Your investments in VBA will serve you well until at least 2049. The last macro language change—from XLM to VBA—happened in 1993. XLM is still supported in Excel to this day. That was a case where VBA was better than XLM, but XLM is still supported 28 years later. Microsoft introduced TypeScript for Excel Online in February 2020. I expect that they will continue to support VBA in the Windows and Mac versions of Excel for the next 28 years.

In the Excel universe today, there are versions of Excel running in Windows, in macOS, on mobile phones powered by Android and iOS, and in modern browsers using Excel Online. In my world, I use Excel 99% of the time on a Windows computer. There's perhaps 1% of the time I use Excel Online. But if you are in a mobile environment where you are using Excel in a browser, then the TypeScript UDFs might be appropriate for you.

For an introduction to TypeScript UDFs in Excel, read Suat M. Özgür's *Excel Custom Functions Straight to the Point* (ISBN 978-1-61547-259-8).

However, TypeScript performance is still horrible. If you don't need your macros to run in Excel Online, the VBA version of your macro will run eight times more quickly than

the TypeScript version. For people who plan to run Excel only on the Mac or Windows platforms, VBA will be your go-to macro language for another decade.

The threat to Excel VBA is the new Excel Power Query tools found in the Get & Transform group of the Data tab in Excel for Windows. If you are writing macros to clean imported data, you should consider cleaning the data once with Power Query and then refreshing the query each day. Bill has a lot of Power Query workflows set up that would have previously required VBA. For a primer on Power Query, check out *Master Your Data with Excel and Power BI: Leveraging Power Query to Get & Transform Your Task Flow* by Ken Puls and Miguel Escobar (ISBN 978-1-61547-058-7).

Is Python a threat to VBA?

Python in Excel is not made to automate chores like VBA. It doesn't control the workbook, respond to events, or handle formatting and file tasks. Instead, it's a powerful alternative to analyzing worksheet data—ideal for advanced calculations, reshaping data with pandas, or creating custom visuals. Think of it as a next-gen formula engine, not a macro tool.

Python runs in Excel as a special formula type using the `=PY()` function. You write Python code directly in a cell, and Excel sends that code to a sandbox-secure cloud environment to execute. The result is returned to the worksheet.

The function supports libraries like pandas, matplotlib, NumPy, and seaborn. This makes it a great option for complex analysis, but Python can't modify the workbook or sheet structure or trigger actions. For now, that means VBA is still your go-to for automating tasks, controlling Excel behavior, and building interactive tools.

What is in this book?

You have taken the right step by purchasing this book. We can help you reduce the learning curve so that you can write your own VBA macros and put an end to the burden of generating reports manually.

Reducing the learning curve

This Introduction provides a case study about the power of macros. Chapter 1, "Unleashing the power of Excel with VBA," introduces the tools and confirms what you probably already know: The macro recorder does not work reliably. Chapter 2, "This sounds like BASIC, so why doesn't it look familiar?" helps you understand the crazy syntax of VBA. Chapter 3, "Referring to ranges, names, and tables," cracks the code on how to work efficiently with ranges, cells, defined names, and tables.

Chapter 4, "Laying the groundwork with variables and structures," lays the groundwork for writing flexible, maintainable VBA code by introducing key concepts like procedures, variables, object references, and code structure tools such as `With` blocks and compiler directives. These building blocks will help you understand—and eventually master—the code you'll encounter throughout the book.

Chapter 5, "Looping and flow control," covers the power of looping using VBA. The case study in this chapter demonstrates creating a program to produce a department report and then wrapping that report routine in a loop to produce 46 reports.

Chapter 6, "R1C1 style formulas," covers, obviously, R1C1 style formulas. Chapter 7, "Event programming," shows how user interaction and automatic Excel functionality, such as calculations, can be used to trigger procedures. Chapter 8, "Arrays," covers arrays. Chapter 9, "Creating custom objects and collections," covers creating custom objects and collections, great tools for organizing data. Chapter 10, "Userforms: An introduction," introduces custom dialog boxes that you can use to collect information from a human using Excel.

Excel VBA power

Chapters 11, "Data mining with Advanced Filter," and 12, "Using VBA to create pivot tables," provide an in-depth look at Filter, Advanced Filter, and pivot tables. Report automation tools rely heavily on these concepts. Chapters 13, "Excel power," and 14, "Sample user-defined functions," include dozens of code samples designed to exhibit the power of Excel VBA and custom functions. Chapter 13 includes the section, "Leveling up: Real project issues, real solutions," which features various project issues I run into and how I resolve them.

Chapters 15, "Creating charts," through 20, "Automating Word," handle charting, data visualizations, web queries, sparklines, and automating Word.

Techie stuff needed to produce applications

Chapter 21, "Using Access as a back end to enhance multiuser access to data," handles reading and writing to Access databases and SQL Server. The techniques for using Access databases enable you to build an application with the multiuser features of Access while keeping the friendly front end of Excel.

Chapter 22, "Advanced userform techniques," shows you how to go further with userforms. Chapter 23, "The Windows Application Programming Interface (API)," teaches some tricky ways to achieve tasks using the Windows API. Chapters 24, "Handling errors," through 26, "Creating Excel add-ins," deal with error handling, custom menus, and add-ins. Chapter 27, "An introduction to creating Office Add-ins," provides a brief introduction to building your own Office Script application within Excel.

Does this book teach Excel?

Not exactly. We assume you already use Excel and are ready to go further with VBA. While we don't walk through every feature, we do point out powerful tools, such as pivot tables and Power Query, when they matter to the code. If you haven't used them before, you'll know what to explore.

Bill regularly presents a Power Excel seminar for accountants—people who use Excel 30 to 40 hours a week. Yet two things happen at every seminar. First, half the room gasps when they see how fast a feature like automatic subtotals or pivot tables can be. Second, someone always trumps Bill. He'll answer a question, and someone in the second row chimes in with a better solution.

The point? Both the authors and readers of this book know Excel—but we still assume that in any given chapter, many readers haven't used pivot tables, and even fewer have tried features like the Top 10 Filter. So, before we automate something in VBA, we'll briefly show how to do it manually in Excel. This isn't a how-to on pivot tables, but it will let you know when it's time to look one up.

Case study: Monthly accounting reports

This is a true story. Valerie is a business analyst in the accounting department of a medium-sized corporation. Her company recently installed an overbudget $16 million enterprise resource planning (ERP) system. As the project ground to a close, there were no resources left in the IT budget to produce the monthly report that this corporation used to summarize each department.

However, Valerie had been close enough to the implementation to think of a way to produce the report herself. She understood that she could export general ledger data from the ERP system to a text file with comma-separated values. Using Excel, Valerie was able to import the general ledger data from the ERP system into Excel.

Creating the report was not easy. As in many other companies, there were exceptions in the data. Valerie knew that certain accounts in one particular cost center needed to be reclassified as expenses. She knew that other accounts needed to be excluded from the report entirely. Working carefully in Excel, Valerie made these adjustments. She created one pivot table to produce the first summary section of the report. She cut the pivot table results and pasted them into a blank worksheet. Then she created a new pivot table report for the second section of the summary. After about three hours, she had imported the data, produced five pivot tables, arranged them in a summary, and neatly formatted the report in color.

Becoming the hero

Valerie handed the report to her manager. The manager had just heard from the IT department that it would be months before they could get around to producing "that convoluted report." When Valerie created the Excel report, she became the instant hero of the day. In three hours, Valerie had managed to do the impossible. Valerie was on cloud nine after some well-deserved recognition.

More cheers

The next day, Valerie's manager attended the monthly department meeting. When the department managers started complaining that they could not get the report from the ERP system, this manager pulled out his department's report and placed it on the table. The other managers were amazed. How was he able to produce this report? Everyone was relieved to hear that someone had cracked the code. The company president asked Valerie's manager if he could have the report produced for each department.

Cheers turn to dread

You can probably see what's coming. This particular company had 46 departments. That means 46 one-page summaries had to be produced once a month. Each report required importing data from the ERP system, backing out certain accounts, producing five pivot tables, and then formatting the reports in color. It had taken Valerie three hours to produce the first report, but after she got into the swing of things, she could produce the 46 reports in 40 hours. Even after she reduced her time per report, though, this is horrible. Valerie had a job to do before she became responsible for spending 40 hours a month producing these reports in Excel.

VBA to the rescue

Valerie found Bill's company, MrExcel Consulting, and explained her situation. In the course of about a week, Bill was able to produce a series of macros in Visual Basic that did all the mundane tasks. For example, the macros imported the data, backed out certain accounts, made five pivot tables, and applied the color formatting. From start to finish, the entire 40-hour manual process was reduced to two button clicks and about 4 minutes.

Right now, either you or someone in your company is probably stuck doing manual tasks in Excel that can be automated with VBA. We are confident that we can walk into any company that has 20 or more Excel users and find a case just as amazing as Valerie's.

Versions of Excel

This eighth edition of *VBA and Macros* is designed to work with Microsoft 365 features released up through July 2025. The previous editions of this book covered code for Excel 97 through Excel 365 (2021). In 80 percent of the chapters, the code today is identical to the code in previous versions.

Differences for Mac users

Although Excel for Windows and Excel for the Mac are similar in terms of user interface, there are a number of differences when you compare the VBA environment. Certainly, nothing in Chapter 23 that uses the Windows API will work on the Mac. That said, the overall concepts discussed in this book apply to the Mac. You can find a general list of differences as they apply to the Mac at *http://www.mrexcel.com/macvba.html*. The VBA Editor for the Mac does not let you design UserForms (Chapter 10). It also has a bug that makes it difficult to create event handler macros (Chapter 7). Excel throws an error when you try to select from the drop-downs at the top of the Code window. You have to first copy and paste an empty event procedure; then the drop-downs will work.

Special elements and typographical conventions

The following typographical conventions are used in this book:

- *Italic*—Indicates new terms when they are defined, special emphasis, non-English words or phrases, and letters or words used as words.
- `Monospace`—Indicates parts of VBA code, such as object or method names.
- **Bold**—Indicates user input.

In addition to these typographical conventions, there are several special elements. Many chapters have at least one case study that presents a real-world solution to common problems. The case study also demonstrates practical applications of the topics discussed in the chapter.

In addition to the case studies, you will see Level Up, Note, Tip, and Caution sidebars.

Level Up!

Level Up sidebars offer deeper insights or clever techniques that go beyond the basics. They're optional side explorations—perfect if you're ready to stretch your skills a bit further than what the current section assumes.

Note Notes provide additional information outside the main thread of the chapter discussion that might be useful for you to know.

Tip Tips provide quick workarounds and time-saving techniques to help you work more efficiently.

Caution Cautions warn about potential pitfalls you might encounter. Pay attention to the Cautions; they alert you to problems that might otherwise cause you hours of frustration.

About the companion content

As a thank-you for buying this book, we have put together a set of 50 Excel workbooks that demonstrate the concepts included in this book. This set of files includes all the code from the book, sample data, and additional notes from the authors.

To download the code files, visit this book's webpage at *MicrosoftPressStore.com/XLVBAAuto/downloads*.

Errata, updates, and book support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed at

MicrosoftPressStore.com/XLVBAAuto/errata.

If you find an error that is not already listed, you can report it to us through the same page.

For additional book support and information, please visit *MicrosoftPressStore.com/Support*.

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to *support.microsoft.com*.

Stay in touch

Let's keep the conversation going! We're on X:

x.com/MicrosoftPress

x.com/MrExcel

bsky.app/profile/tsyrstad.bsky.social

CHAPTER 2

This sounds like BASIC, so why doesn't it look familiar?

In this chapter, you will:

- Find out how VBA is different from BASIC
- Understand the parts of VBA "speech"
- Find out that learning VBA is not really hard
- Examine recorded macro code using the VB Editor and Help
- Use debugging tools to figure out recorded code
- Get to know the Object Browser
- Learn seven tips for cleaning up recorded code

As mentioned in Chapter 1, "Unleashing the power of Excel with VBA," if you have taken a class in a procedural language such as BASIC or COBOL, you might be confused when you look at VBA code. Even though VBA stands for *Visual Basic for Applications*, it is an *object-oriented* version of BASIC. Here is a bit of recorded VBA code:

```
Selection.End(xlDown).Select
Range("A11").Select
ActiveCell.FormulaR1C1 = "Total"
Range("E11").Select
Selection.FormulaR1C1 = _
  "=SUM(R[-9]C:R[-1]C)"
Selection.AutoFill _
  Destination:=Range("E11:G11"), _
  Type:=xlFillDefault
```

This code likely makes no sense to anyone who knows only procedural languages. Unfortunately, your first introduction to programming in school (assuming that you are more than 40 years old) would have been a procedural language.

Here is a section of code written in the BASIC language:

```
For x = 0 to 9
  Print Rpt$(" ",x)
  Print "*"
Next x
```

If you run this code, you get a pyramid of asterisks on your screen:

```
*
 *
  *
   *
    *
     *
      *
       *
        *
         *
```

If you have ever been in a procedural programming class, you can probably look at the code and figure out what is going on because procedural languages are more English-like than object-oriented languages. The statement `Print "Hello World"` follows the verb–object format, which is how you would generally talk. Let's step away from programming for a second and look at a concrete example.

Understanding the parts of VBA "speech"

If you were going to write code for instructions to play soccer using BASIC, the instruction to kick a ball would look something like this:

```
"Kick the Ball"
```

Hey, this is how you talk! It makes sense. You have a verb (*kick*) and then a noun (*ball*). The BASIC code in the preceding section has a verb (`Print`) and a noun (the asterisk, `*`). Life is good.

Here is the problem: VBA doesn't work like this. In fact, no object-oriented language works like this. In an object-oriented language, the objects (nouns) are most important, hence the name, "object-oriented." If you were going to write code for instructions to play soccer with VBA, the basic structure would be as follows:

```
Ball.Kick
```

You have a noun (`Ball`), which comes first. In VBA, this is an *object*. Then, you have the verb (`Kick`), which comes next. In VBA, this is a *method*.

The basic structure of VBA is a bunch of lines of code with this syntax:

```
Object.Method
```

Needless to say, this is not English. If you took a romance language in high school, you will remember that those languages use a "noun–adjective" construct. However, no one uses "noun–verb" to tell someone to do something:

```
Water.Drink
Food.Eat
Girl.Kiss
```

That is why VBA is confusing to someone who previously took a procedural programming class.

Let's carry the analogy a bit further. Imagine that you walk onto a grassy field, and there are five balls in front of you: a soccer ball, basketball, baseball, bowling ball, and tennis ball. You want to instruct a kid on your soccer team to "kick the soccer ball."

If you tell them to kick the ball (or `Ball.Kick`), you really aren't sure which one of the five balls they will kick. Maybe they will kick the one closest to them, which could be a problem if they are standing in front of the bowling ball.

For almost any noun or object in VBA, there is a collection of that object. Think about Excel. If you can have one row, you can have a bunch of rows. If you can have one cell, you can have a bunch of cells. If you can have one worksheet, you can have a bunch of worksheets. The only difference between an object and a collection is that you add an *s* to the name of the object:

`Row` becomes `Rows`.

`Cell` becomes `Cells`.

`Ball` becomes `Balls`.

When you refer to something that is a collection, you have to tell the programming language to which item you are referring. There are a couple of ways to do this. You can refer to an item by using a number. For example, if the soccer ball is the second ball, you might say this:

```
Balls(2).Kick
```

This works fine, but it could be a dangerous way to program. For example, it might work on Tuesday. However, if you get to the field on Wednesday and someone has rearranged the balls, `Balls(2).Kick` might be a painful exercise.

A much safer way to go is to use a name for the object in a collection. You can say the following:

```
Balls("Soccer").Kick
```

With this method, you always know that it will be the soccer ball that is being kicked.

So far, so good. You know that a ball will be kicked, and you know that it will be a soccer ball. For most of the verbs or methods in Excel VBA, there are *parameters* that tell *how* to do the action. These parameters act as adverbs. You might want the soccer ball to be kicked to the left and with a hard force.

In this case, the method would have a number of parameters that tell how the program should perform the method:

```
Balls("Soccer").Kick Direction:=Left, Force:=Hard
```

When you are looking at VBA code, the colon–equal sign combination (`:=`) indicates that you are looking at the parameters of how the verb should be performed.

Sometimes, a method will have a list of 10 parameters, some of which are optional. For example, if the `Kick` method has an `Elevation` parameter, you would have this line of code:

```
Balls("Soccer").Kick Direction:=Left, Force:=Hard, Elevation:=High
```

Here is the confusing part: Every method has a default order for its parameters. If you are not a conscientious programmer and happen to know the order of the parameters, you can leave off the parameter names. The following code is equivalent to the previous line of code:

```
Balls("Soccer").Kick Left, Hard, High
```

This throws a monkey wrench into our understanding. Without `:=`, it is not obvious that you have parameters. Unless you know the parameter order, you might not understand what is being said. It is pretty easy with `Left`, `Hard`, and `High`, but when you have parameters like the following:

```
ActiveSheet.Shapes.AddShape Type:=1, Left:=10, Top:=20, Width:=100, Height:=200
```

It gets confusing if you instead have this:

```
ActiveSheet.Shapes.AddShape 1, 10, 20, 100, 200
```

The preceding line is valid code. However, unless you know that the default order of the parameters for this `Add` method is `Type`, `Left`, `Top`, `Width`, `Height`, this code does not make sense. The default order for any particular method is the order of the parameters, as shown in the Help topic for that method.

To make life more confusing, you are allowed to start specifying parameters in their default order without naming them, and then you can switch to naming parameters when you hit one that does not match the default order. If you want to kick the ball to the left and high but do not care about the force (that is, you are willing to accept the default force), the following two statements are equivalent:

```
Balls("Soccer").Kick Direction:=Left, Elevation:=High
Balls("Soccer").Kick Left, Elevation:=High
```

However, keep in mind that as soon as you start naming parameters, they have to be named for the remainder of that line of code.

Some methods simply act on their own. To simulate pressing the F9 key, you use this code:

```
Application.Calculate
```

Other methods perform an action and create something. For example, you can add a worksheet by using the following:

```
Worksheets.Add Before:=Worksheets(1)
```

However, because `Worksheets.Add` creates a new object, you can assign the results of this method to a variable. In this case, you must surround the parameters with parentheses:

```
Set MyWorksheet = Worksheets.Add(Before:=Worksheets(1))
```

Note Don't worry if the use of `Set` isn't clear yet. This command and other foundational concepts are covered in more detail in Chapter 4, "Laying the groundwork with variables and structures."

One final bit of grammar is necessary: adjectives. Just as adjectives describe a noun, *properties* describe an object. Because you are an Excel fan, let's switch from the soccer analogy to an Excel analogy. There is an object to describe the active cell. Fortunately, it has a very intuitive name:

```
ActiveCell
```

Suppose you want to change the color of the active cell to red. There is a property called `Interior.Color` for a cell that uses a complex series of codes. However, you can turn a cell to red by using this code:

```
ActiveCell.Interior.Color = 255
```

You can see how this can be confusing. Again, there is the *noun*-dot-*something* construct, but this time, it is `Object.Property` rather than `Object.Method`. How you tell them apart is quite subtle: There is no colon before the equal sign. A property is almost always set equal to something, or perhaps the value of a property is assigned to something else.

To make this cell color the same as cell A1, you might say this:

```
ActiveCell.Interior.Color = Range("A1").Interior.Color
```

`Interior.Color` is a property. Actually, `Interior` is a property of the `Range` object, and `Color` is a property of the `Interior` property. By changing the value of a property, you can make things look different. It is kind of bizarre: Change an adjective, and you are actually doing something to the cell. Humans would say, "Color the cell red," whereas VBA says this:

```
ActiveCell.Interior.Color = 255
```

Table 2-1 summarizes the VBA "parts of speech."

TABLE 2-1 Parts of the VBA programming language

VBA Component	Analogous To	Notes
Object	Noun	Examples include cell or sheet.
Collection	Plural noun	Usually specifies which object: `Worksheets(1)`.
Method	Verb	Appears as `Object.Method`.
Parameter	Adverb	Lists parameters after the method. Separate the parameter name from its value with :=.
Property	Adjective	You can set a property (for example, `ActiveCell.Height=10`) or store the value of a property (for example, `x = ActiveCell.Height`).

VBA is not really hard

Knowing whether you are dealing with properties or methods helps you set up the correct syntax for your code. Don't worry if it all seems confusing right now. When you are writing VBA code from scratch, it is tough to know whether the process of changing a cell to yellow requires a verb or an adjective. Is it a method or a property?

This is where the macro recorder is especially helpful. When you don't know how to code something, you record a short little macro, look at the recorded code, and figure out what is going on.

VBA Help files: Using F1 to find anything

Excel VBA Help is an amazing feature, provided that you are connected to the Internet. If you are going to write VBA macros, you absolutely *must* have access to the VBA Help topics installed. Follow these steps to see how easy it is to get help in VBA:

1. Open Excel and switch to the VB Editor by pressing Alt+F11. From the Insert menu, select Module.
2. Type these three lines of code:

```
Sub Test()
  MsgBox "Hello World!"
End Sub
```

3. Click inside the word `MsgBox`.
4. With the cursor in the word `MsgBox`, press F1. If you can reach the Internet, you will see the Help topic for the `MsgBox` function.

Using Help topics

If you request help on a function or method, the Help topic walks you through the various available arguments. If you browse to the bottom of a Help topic, you can see a great resource: code samples under the Example heading (see Figure 2-1).

It is possible to select the code, copy it to the Clipboard by clicking the Copy shortcut in the top-right corner of the code box, and then paste it into a module by pressing Ctrl+V.

After you record a macro, if there are objects or methods about which you are unsure, you can get help by inserting the cursor in any keyword and pressing F1.

Example

This example uses the **MsgBox** function to display a critical-error message in a dialog box with **Yes** and **No** buttons. The **No** button is specified as the default response. The value returned by the **MsgBox** function depends on the button chosen by the user. This example assumes that `DEMO.HLP` is a Help file that contains a topic with a Help context number equal to `1000`.

VB Copy

```
Dim Msg, Style, Title, Help, Ctxt, Response, MyString
Msg = "Do you want to continue ?"    ' Define message.
Style = vbYesNo Or vbCritical Or vbDefaultButton2    ' Define buttons.
Title = "MsgBox Demonstration"    ' Define title.
Help = "DEMO.HLP"    ' Define Help file.
Ctxt = 1000    ' Define topic context.
        ' Display message.
Response = MsgBox(Msg, Style, Title, Help, Ctxt)
If Response = vbYes Then    ' User chose Yes.
    MyString = "Yes"    ' Perform some action.
Else    ' User chose No.
    MyString = "No"    ' Perform some action.
End If
```

FIGURE 2-1 Most Help topics include code samples.

Examining recorded macro code: Using the VB Editor and Help

Let's take a look at the code you recorded in Chapter 1 to see whether it makes more sense now that you know about objects, properties, and methods. You can also see whether it's possible to correct the errors created by the macro recorder.

Figure 2-2 shows the first code that Excel recorded in the example from Chapter 1.

Now that you understand the concept of `Noun.Verb` or `Object.Method`, consider the first line of code that reads `Workbooks.OpenText`. In this case, `Workbooks` is a collection object, and `OpenText` is a method. Click the word `OpenText` and press F1 for an explanation of the `OpenText` method (see Figure 2-3).

The Help file confirms that `OpenText` is a method, or an action word. The default order for all the arguments that can be used with `OpenText` appears in a Parameters table. Notice that only one argument is required: `Filename`. All the other arguments are listed as optional.

```
(General)
Sub ImportInvoice()
'
' ImportInvoice Macro
' Import Invoice.txt. Add Total Row. Format.
'
' Keyboard Shortcut: Ctrl+i
'
    Workbooks.OpenText Filename:="G:\2016VBA\SampleFiles\invoice.txt", Origin:= _
        437, StartRow:=1, DataType:=xlDelimited, TextQualifier:=xlDoubleQuote, _
        ConsecutiveDelimiter:=False, Tab:=False, Semicolon:=False, Comma:=True _
        , Space:=False, Other:=False, FieldInfo:=Array(Array(1, 3), Array(2, 1), _
        Array(3, 1), Array(4, 1), Array(5, 1), Array(6, 1), Array(7, 1)), TrailingMinusNumbers _
        :=True
    Selection.End(xlDown).Select
    Range("A11").Select
    ActiveCell.FormulaR1C1 = "Total"
    Range("E11").Select
    Selection.FormulaR1C1 = "=SUM(R[-9]C:R[-1]C)"
    Selection.AutoFill Destination:=Range("E11:G11"), Type:=xlFillDefault
    Range("E11:G11").Select
    Rows("1:1").Select
    Selection.Font.Bold = True
    Rows("11:11").Select
    Selection.Font.Bold = True
    Selection.CurrentRegion.Select
    Selection.Columns.AutoFit
End Sub
```

FIGURE 2-2 Here is the recorded code from the example in Chapter 1.

Parameters

Filename	Required	**String**	Specifies the file name of the text file to be opened and parsed.
Origin	Optional	**Variant**	Specifies the origin of the text file. Can be one of the following **XlPlatform** constants: **xlMacintosh**, **xlWindows**, or **xlMSDOS**. Additionally, this could be an integer representing the code page number of the desired code page. For example, "1256" would specify that the encoding of the source text file is Arabic (Windows). If this argument is omitted, the method uses the current setting of the **File Origin** option in the **Text Import Wizard**.
StartRow	Optional	**Variant**	The row number at which to start parsing text. The default value is 1.
DataType	Optional	**Variant**	Specifies the column format of the data in the file. Can be one of the following XlTextParsingType constants: **xlDelimited** or **xlFixedWidth**. If this argument is not specified, Microsoft Excel attempts to determine the column format when it opens the file.
TextQualifier	Optional	XlTextQualifier	Specifies the text qualifier.
ConsecutiveDelimiter	Optional	**Variant**	**True** to have consecutive delimiters considered one delimiter. The default is **False**.

FIGURE 2-3 This shows part of the Help topic for the `OpenText` method.

Optional parameters

The Help file can tell you what happens if you skip an optional parameter. For `StartRow`, the Help file indicates that the default value is 1. If you leave out the `StartRow` parameter, Excel starts importing at row 1. This is fairly safe.

Now look at the Help file note about `Origin`. If this argument is omitted, you inherit whatever value was used for `Origin` the last time someone used this feature in Excel on this computer. That is a recipe for disaster. For example, your code might work 98 percent of the time. However, immediately after someone imports an Arabic file, Excel remembers the setting for Arabic and thereafter assumes that this is what your macro wants if you don't explicitly code this parameter.

Defined constants

Look at the Help file entry for `DataType` in Figure 2-3, which says it can be one of these constants: `xlDelimited` or `xlFixedWidth`. The Help file says these are the valid `xlTextParsingType` constants that are predefined in Excel VBA. In the VB Editor, press Ctrl+G to bring up the Immediate window. In the Immediate window, type this line and press Enter:

```
Print xlFixedWidth
```

The answer appears in the Immediate window. `xlFixedWidth` is the equivalent of saying `2` (see Figure 2-4). In the Immediate window, type **Print xlDelimited**, which is really the same as typing **1**. Microsoft correctly assumes that it is easier for someone to read code that uses the somewhat English-like `xlDelimited` term than to read `1`.

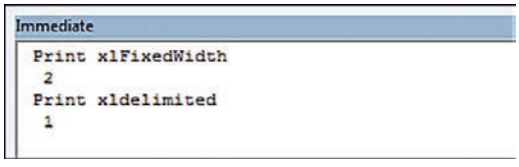


FIGURE 2-4 In the Immediate window of the VB Editor, you can query to see the true value of constants such as `xlFixedWidth`.

If you were an evil programmer, you could certainly memorize all these constants and write code using the numeric equivalents of the constants. However, the programming gods (and the next person who has to look at your code) will curse you for doing so.

In most cases, the Help file either specifically calls out the valid values of the constants or offers a hyperlink that opens the Help topic, showing the complete enumeration and the valid values for the constants (see Figure 2-5).

If you read the Help topic on `OpenText`, you can surmise that it is basically the equivalent of opening a file using the Text Import Wizard. In step 1 of the wizard, you normally choose either Delimited or Fixed Width. You also specify the file origin and at which row to start. This first step of the wizard is handled by these parameters of the `OpenText` method:

```
Origin:=437
StartRow:=1
DataType:=xlDelimited
```

XlColumnDataType enumeration (Excel)

06/08/2017 • 2 minutes to read •

Specifies how a column is to be parsed.

Name	Value	Description
xlDMYFormat	4	DMY date format.
xlDYMFormat	7	DYM date format.
xlEMDFormat	10	EMD date format.
xlGeneralFormat	1	General.
xlMDYFormat	3	MDY date format.
xlMYDFormat	6	MYD date format.
xlSkipColumn	9	Column is not parsed.
xlTextFormat	2	Text.
xlYDMFormat	8	YDM date format.
xlYMDFormat	5	YMD date format.

FIGURE 2-5 Click the hyperlink to see all the possible constant values. Here, the 10 possible `xlColumnDataType` constants are revealed in a new Help topic.

Step 2 of the Text Import Wizard enables you to specify that your fields be delimited by commas. Because you do not want to treat two commas as a single comma, the Treat Consecutive Delimiters As One checkbox should not be selected. Sometimes, a field may contain a comma, such as "XYZ, Inc." In this case, the field should have quotes around the value, as specified in the Text Qualifier box. This second step of the wizard is handled by the following parameters of the `OpenText` method:

```
TextQualifier:=xlDoubleQuote
ConsecutiveDelimiter:=False
Tab:=False
Semicolon:=False
Comma:=True
Space:=False
Other:=False
```

Step 3 of the wizard is where you actually identify the field types. In this case, you leave all fields as General except for the first field, which is marked as a date in MDY (Month, Day, Year) format. This is represented in code by the `FieldInfo` parameter.

The third step of the Text Import Wizard is fairly complex. The entire `FieldInfo` parameter of the `OpenText` method duplicates the choices made in this step of the wizard. If you happen to click the Advanced button on the third step of the wizard, you have an opportunity to specify something other than the default decimal and thousands separators, as well as the setting Trailing Minus For Negative Numbers.

Note Note that the macro recorder does not write code for `DecimalSeparator` or `ThousandsSeparator` unless you change these from the defaults. The macro recorder does, however, always record the `TrailingMinusNumbers` parameter.

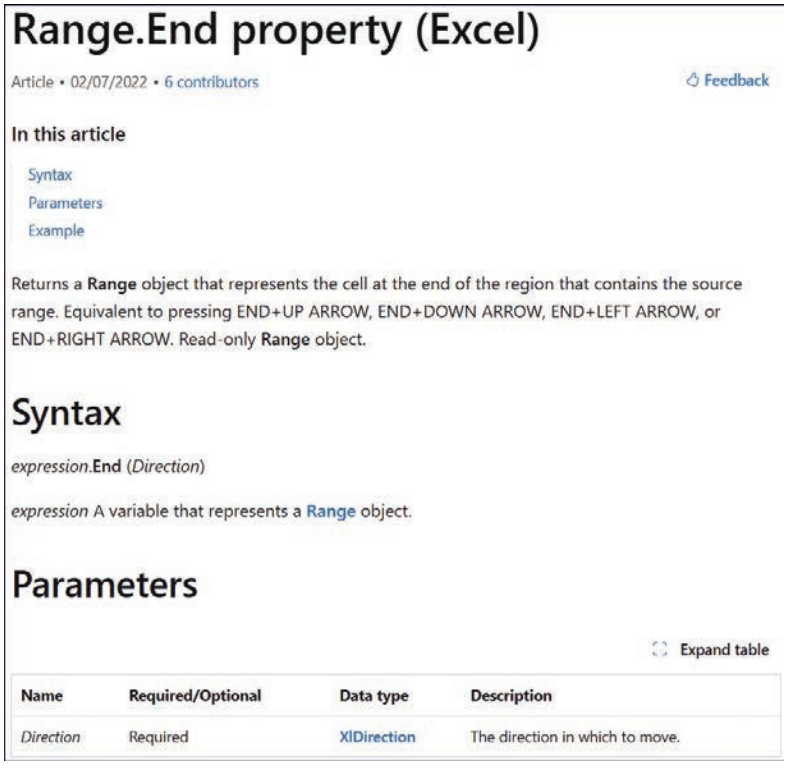
Remember that every action you perform in Excel while recording a macro gets translated to VBA code. In the case of many dialogs, the settings you do not change are often recorded along with the items you do change. When you click OK to close the dialog, the macro recorder often records all the current settings from the dialog in the macro.

Here is another example. The next line of code in the macro is this:

```
Selection.End(xlDown).Select
```

You can click to get help for three topics in this line of code: `Selection`, `End`, and `Select`. Assuming that `Selection` and `Select` are somewhat self-explanatory, click in the word `End` and press F1 for Help.

This Help topic says that `End` is a property. It returns a `Range` object that is equivalent to pressing End+up arrow or End+down arrow in the Excel interface (see Figure 2-6). If you click the blue hyperlink for `xlDirection`, you see the valid parameters that can be passed to the `End` function.



Range.End property (Excel)

Article • 02/07/2022 • 6 contributors Feedback

In this article

Syntax
Parameters
Example

Returns a **Range** object that represents the cell at the end of the region that contains the source range. Equivalent to pressing END+UP ARROW, END+DOWN ARROW, END+LEFT ARROW, or END+RIGHT ARROW. Read-only **Range** object.

Syntax

expression.**End** (*Direction*)

expression A variable that represents a Range object.

Parameters

Expand table

Name	Required/Optional	Data type	Description
Direction	Required	XlDirection	The direction in which to move.

FIGURE 2-6 The correct Help topic for the End property.

Properties can return objects

In VBA, some properties give you simple values—like a number, a string, or True/False. But other properties give you something more powerful—another object. This is called *returning an object*. For example, the `End(xlDown)` property doesn't just give you a value—it gives you a `Range` object that represents a specific cell. Once you have that object, you can do things with it, like select it, read its contents, or format it.

Consider the line of code currently under examination:

```
Selection.End(xlDown).Select
```

The `End` keyword is a property, but from the Help file, you see that it returns a `Range` object. You then call `Select` on that returned object. In the end, you're calling a method (`Select`) on the result of a property (`End`). You'll do this often in VBA when a property gives you an object you can immediately act on.

`Selection` is actually a property of `Application`

`Selection` might look like an object, but if you bring up the help topic for `Selection`, you'll see that it is actually a property. In reality, the proper code would be `Application.Selection`. However, when you are writing code within Excel, VBA assumes you are referring to the Excel object model, so you can leave off the `Application` prefix. If you were to automate Excel from another program, like Word, you'd need to include an object reference before the `Selection` property, such as `ExcelApp.Selection`.

`Selection` can return different types of objects. It returns the object of whatever is selected, such as cells, charts, etc.

Using debugging tools to figure out recorded code

The following sections introduce some awesome debugging tools that are available in the VB Editor. These tools are excellent for helping you see what a recorded macro code is doing.

Stepping through code

Generally, a macro runs quickly: You start it, and less than a second later, it's done. If something goes wrong, you don't have an opportunity to figure out what the macro is doing. However, using Excel's Step Into feature makes it possible to run one line of code at a time.

To use this feature, make sure your cursor is in the procedure you want to run, such as the `ImportInvoice` procedure, and then from the menu, select Debug | Step Into, as shown in Figure 2-7. Alternatively, you can press F8.

The VB Editor is now in Break mode. The line about to be executed is highlighted in yellow, with a yellow arrow in the margin before the code (see Figure 2-8).

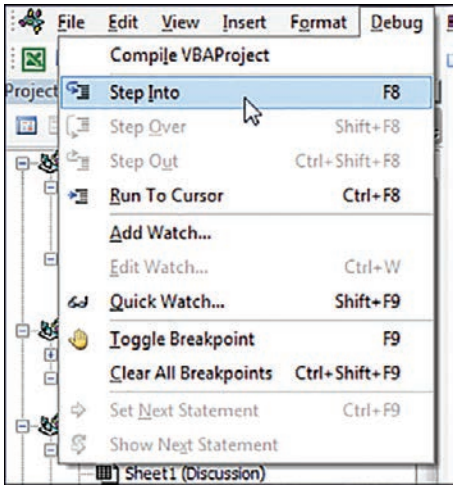


FIGURE 2-7 You can use the Step Into feature to run a single line of code at a time.

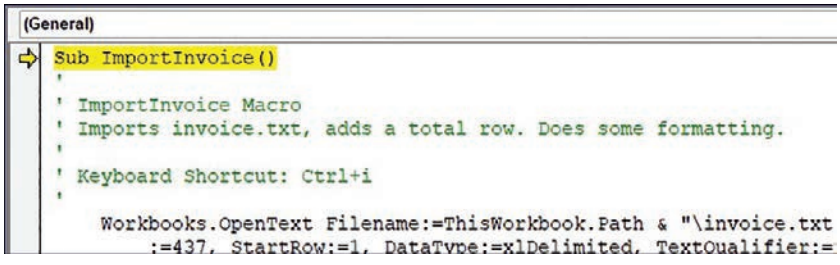


FIGURE 2-8 The first line of the macro is about to run.

In this case, the next line to be executed is the `Sub ImportInvoice()` line. This basically says, "You are about to start running this procedure." Press the F8 key to execute the line in yellow and move to the next line of code. The long code for `OpenText` is then highlighted. Press F8 to run this line of code. When you see that `Selection.End(xlDown).Select` is highlighted, you know that Visual Basic has finished running the `OpenText` command. At this point, you can press Alt+Tab to switch to Excel and see that the `Invoice.txt` file has been parsed into Excel. Note that A1 is selected.

Tip If you have a wide monitor, you can use the Restore Down icon at the top right of the VBA window to arrange the window so that you can see both the VBA window and the Excel window. (Restore Down is the two-tiled window icon between the Minimize "dash" and the Close Window *X* icon at the top of every maximized window).

This is also a great trick to use while recording new code. You can actually watch the code appear as you do things in Excel.

Switch back to the VB Editor by pressing Alt+Tab. The next line about to be executed is `Selection.End(xlDown).Select`. Press F8 to run this code. Switch to Excel to see that the last cell in your data set is selected.

Press F8 again to run the `Range("A11").Select` line. If you switch to Excel by pressing Alt+Tab, you see that this is where the macro starts to have problems. Instead of moving to the first blank row, the program moves to the wrong row.

Now that you have identified the problem area, you can stop the code execution by using the Reset command. You can start the Reset command either by selecting Run | Reset or by clicking the Reset button on the toolbar (the small blue square next to the Run and Pause icons). After clicking Reset, you should return to Excel and undo anything done by the partially completed macro. In this case, you need to close the Invoice.txt file without saving.

More debugging options: Breakpoints

If you have hundreds of lines of code, you might not want to step through each line one at a time. If you have a general idea that a problem is happening in one particular section of the program, you can set a breakpoint. You can then have the code start to run, but the macro pauses just before it executes the breakpoint line of code.

To set a breakpoint, click in the gray margin area to the left of the line of code on which you want to break. A large maroon dot appears next to this code, and the line of code is highlighted in brown (see Figure 2-9). (If you don't see the margin area, go to Tools | Options | Editor Format and select the Margin Indicator Bar checkbox.) Or select a line of code and press F9 to toggle a breakpoint on or off.

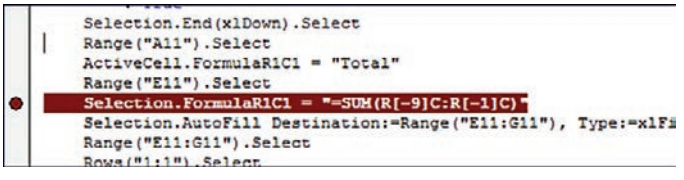


FIGURE 2-9 The large maroon dot signifies a breakpoint.

Next, from the Visual Basic menu, select Run | Run Sub/UserForm or press F5. The program executes but stops just before running the line in the breakpoint. The VB Editor shows the breakpoint line highlighted in yellow. You can now press F8 to begin stepping through the code.

After you have finished debugging your code, remove the breakpoints by clicking the dark brown dot in the margin next to each breakpoint to toggle it off. Alternatively, you can select Debug | Clear All Breakpoints or press Ctrl+Shift+F9 to clear all breakpoints that you set in the project.

Backing up or moving forward in code

When you are stepping through code, you might want to jump over some lines of code, or you might have corrected some lines of code that you want to run again. This is easy to do when you are working in Break mode. One favorite method is to use the mouse to grab the yellow arrow. The cursor changes

to a three-arrow icon, which enables you to move the next line up or down. Drag the yellow line to whichever line you want to execute next. The other option is to right-click the line to which you want to jump and then select Set Next Statement.

Not stepping through each line of code

When you are stepping through code, you might want to run a section of code without stepping through each line, such as when you get to a loop. You might want VBA to run through the loop 100 times so you can step through the lines after the loop. It is particularly monotonous to press the F8 key hundreds of times to step through a loop. Instead, click the cursor on the line you want to step to and then press Ctrl+F8 or select Debug | Run To Cursor. This command is also available in the right-click menu.

Querying anything while stepping through code

Even though variables have not yet been discussed, you can query the value of anything while in Break mode. However, keep in mind that the macro recorder never records a variable.

Using the Immediate window

Press Ctrl+G to display the Immediate window in the VB Editor. While the macro is in Break mode, you can ask the VB Editor to tell you the currently selected cell, the name of the active sheet, or the value of any variable. Figure 2-10 shows several examples of queries typed into the Immediate window.

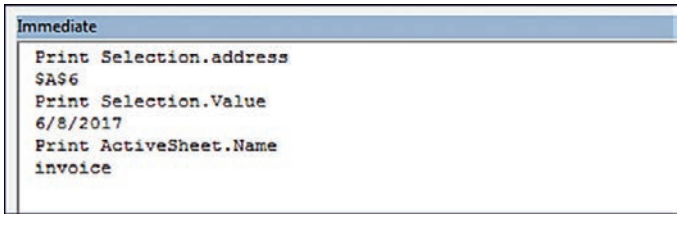


FIGURE 2-10 Queries that can be typed into the Immediate window while a macro is in Break mode, shown along with their answers.

> **Tip** Instead of typing `Print`, you can type a question mark: `? Selection.Address`. Read the question mark as, "What is."

When invoked with Ctrl+G, the Immediate window usually appears at the bottom of the code window. You can use the resize handle, which is located above the Immediate title bar, to make the window larger or smaller.

There is a scrollbar on the side of the Immediate window that you can use to scroll backward or forward through past entries.

It is not necessary to run queries only at the bottom of the Immediate window. For example, if you have just run a line of code, type `?Selection.Address` in the Immediate window to ensure that this line of code worked.

Next, press the F8 key to run the next line of code. Instead of retyping the same query, in the Immediate window, click anywhere in the line that contains the last query and press Enter. The Immediate window runs this query again, displays the results on the next line, and pushes the old results farther down the window.

You also can use this method to change the query by clicking to the right of the word `Address` in the Immediate window. Press the Backspace key to erase the word `Address` and instead type `Columns.Count`. Press Enter, and the Immediate window shows the number of columns in the selection.

This is an excellent technique to use when you are trying to figure out a sticky bit of code. For example, you can query the name of the active sheet (`?ActiveSheet.Name`), the selection (`?Selection.Address`), the active cell (`?ActiveCell.Address`), the formula in the active cell (`?ActiveCell.Formula`), the value of the active cell (`?ActiveCell.Value` or `?ActiveCell` because `Value` is the default property of a cell), and so on.

To dismiss the Immediate window, click the *X* in its upper-right corner.

Note Ctrl+G does not toggle the window off. Use the *X* at the top right of the Immediate window to close it.

Tip The Immediate window is a very useful tool, and I always have it and the Watches window open. If your monitor is big enough, I highly recommend this setup. If you have multiple monitors, you can move these windows to another monitor.

Querying by hovering

In many instances, you can hover the cursor over an expression in code and then wait a second for a tooltip to show the current value of the expression. This is incredibly helpful when you get to looping in Chapter 5, "Looping and flow control." It also comes in handy with recorded code. Note that the expression that you hover over does not have to be in the line of code just executed. In Figure 2-11, Visual Basic just selected E11, making E11 the active cell. If you hover the cursor over `ActiveCell.FormulaR1C1`, you see a tooltip showing that the formula in the active cell is `"=SUM(R[-9]C:R[-1]C)"`.

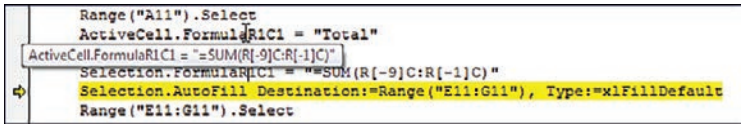


FIGURE 2-11 Hover the mouse cursor over any expression for a few seconds, and a tooltip shows the current value of the expression.

Sometimes, the VBA window seems to not respond to hovering. Because some expressions are not supposed to show values, it is difficult to tell whether VBA is not displaying a value on purpose or whether you are in the buggy "not responding" mode. Try hovering over something that you know should respond, such as a variable. If you get no response, hover, click into the variable, and continue to hover. This tends to wake Excel from its stupor, and hovering works again.

Are you impressed yet? This chapter started with a complaint that VBA doesn't seem much like BASIC. However, by now, you have to admit that the Visual Basic environment is great to work in and that the debugging tools are excellent.

Querying by using a Watches window

In Visual Basic, a watch is not something you wear on your wrist; instead, it allows you to watch the value of any expression while you step through code. Let's say that in the current example, you want to watch to see what is selected as the code runs. You can do this by setting up a watch for `Selection.Address`.

From the Debug menu, select Add Watch. In the Add Watch dialog, enter **Selection.Address** in the Expression text box and click OK (see Figure 2-12).

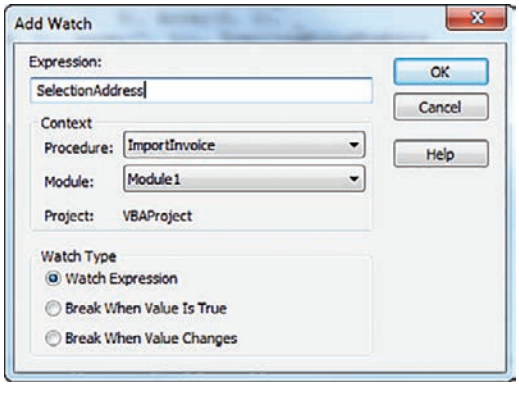


FIGURE 2-12 Setting up a watch to see the address of the current selection.

A Watches window is added to the busy Visual Basic window, usually at the bottom of the code window. When you start stepping through the code, it imports the file and then selects the last row with data. The Watches window confirms that `Selection.Address` is `$A$18` (see Figure 2-13).

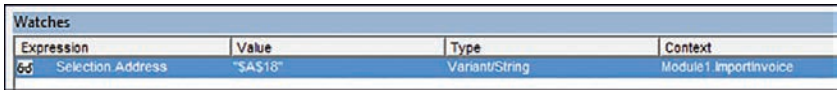


FIGURE 2-13 Without having to hover or type in the Immediate window, you can always see the value of watched expressions.

Press the F8 key to run the code to the line after `Rows("1:1").Select`. The Watches window is updated to show that the current address of the `Selection` is now `$1:$1`.

In the Watches window, the value column is read/write (where possible)! You can type a new value here and see it change on the worksheet. For example, if your watch expression is `Selection.Value`, you can click on the value and enter a new one.

Using a watch to set a breakpoint

Right-click the `Selection.Address` expression in the Watches window and select Edit Watch. In the Watch Type section of the Edit Watch dialog, select Break When Value Changes. Click OK.

The glasses icon to the left of the expression changes to a hand with a triangle icon. You can now press F5 to run the code. The macro starts running lines of code until something new is selected. This is very powerful. Instead of having to step through each line of code, you can now conveniently have the macro stop only when something important has happened. You also can set up a watch to stop when the value of a particular variable changes.

Using a watch on an object

In the preceding example, you watched a specific property: `Selection.Address`. It is also possible to watch an object such as `Selection`. In Figure 2-14, when a watch has been set up on `Selection`, you get the glasses icon and a + icon.

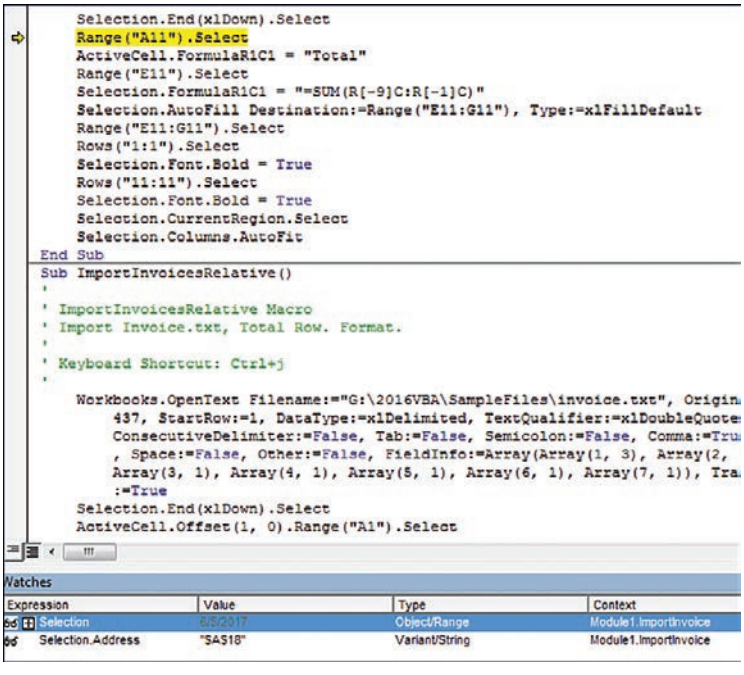
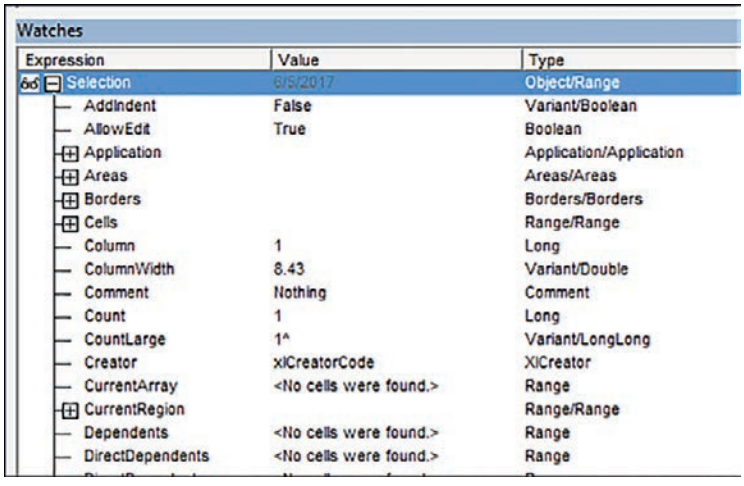


FIGURE 2-14 Setting a watch on an object gives you a + icon next to the glasses.

By clicking the + icon, you can see all the properties associated with `Selection`. When you look at Figure 2-15, you can see more than you ever wanted to know about `Selection`! There are properties

you probably never realized are available. You can see that the `AddIndent` property is set to `False`, and the `AllowEdit` property is set to `True`. There are useful properties further down in the list, such as the `Formula` of the selection.

In this Watches window, some entries can be expanded. For example, the `Borders` collection has a + next to it, which means you can click any + icon to see more details.



Watches

Expression	Value	Type
Selection	6/5/2017	Object/Range
AddIndent	False	Variant/Boolean
AllowEdit	True	Boolean
Application		Application/Application
Areas		Areas/Areas
Borders		Borders/Borders
Cells		Range/Range
Column	1	Long
ColumnWidth	8.43	Variant/Double
Comment	Nothing	Comment
Count	1	Long
CountLarge	1^	Variant/LongLong
Creator	xlCreatorCode	XlCreator
CurrentArray	<No cells were found.>	Range
CurrentRegion		Range/Range
Dependents	<No cells were found.>	Range
DirectDependents	<No cells were found.>	Range

FIGURE 2-15 Clicking the + icon shows a plethora of properties and their current values.

Object Browser: The ultimate reference

In the VB Editor, press F2 to open the Object Browser, which lets you browse and search the entire Excel object library. The built-in Object Browser is always available; you simply press the F2 key. The next few pages show you how to use it.

By default, the Object Browser opens where the code window normally appears. However, you can resize the window and reposition it anywhere you like, including on another monitor.

The topmost dropdown currently shows <All Libraries>. There are entries in this dropdown for Excel, Office, VBA, and each workbook that you have open, plus additional entries for anything you check in Tools | References. For now, go to the dropdown and select Excel.

In the bottom-left window of the Object Browser is a list of all classes available for Excel (see Figure 2-16). Click the `Application` class in the left window. The right window adjusts to show all properties and methods that apply to the `Application` object. Click something in the right window, such as `ActiveCell`. The bottom window of the Object Browser tells you that `ActiveCell` is a property that returns a range. It also tells you that `ActiveCell` is read-only (an alert that you cannot assign an address to `ActiveCell` to move the cell pointer).

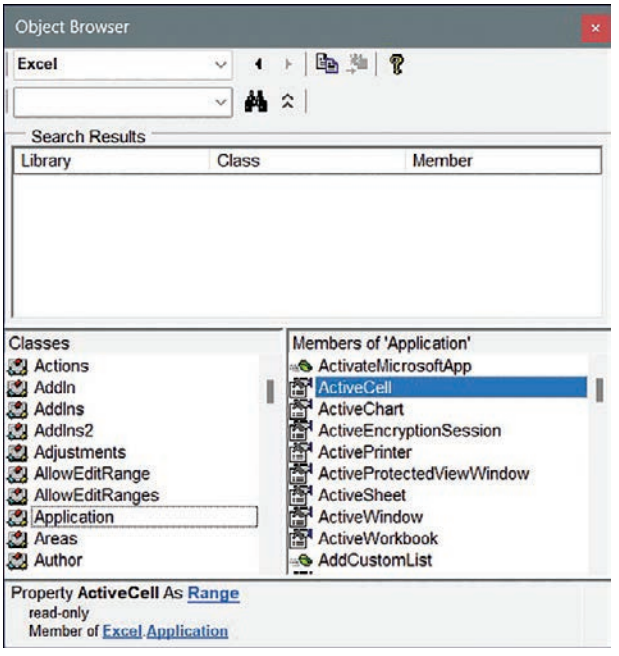


FIGURE 2-16 The Object Browser lets you explore objects, properties, methods, constants, and enumerations available in the VBA environment.

You have learned from the Object Browser that `ActiveCell` returns a range. When you click the hyperlink for `Range` in the bottom window, the Classes and Members windows update to show you all the properties and methods that apply to `Range` objects and, hence, to the `ActiveCell` property. Click any property or method and then click the yellow question mark near the top of the Object Browser to go to the online Help topic for that property or method.

Type any term in the text box in the Search field (next to the binoculars) and press Enter to find all matching members of the Excel library. Methods appear as green books with speed lines. Properties appear as index cards, each with a hand pointing to it.

The search capabilities and hyperlinks available in the Object Browser make it much more valuable than an alphabetic printed listing of all the information. Learn to make use of the Object Browser in the VBA window by pressing F2. To close the Object Browser and return to your code window, click the *X* in the upper-right corner.

Tip If you've maximized the Object Browser and are having trouble returning it to a floating window, try this: Drag it to the left or right edge of the VB Editor to dock it there. Once docked, move it back to the center of the code window—at that point, it should return to a resizable floating window.

Seven tips for cleaning up recorded code

Chapter 1 gave you four tips for recording code. So far, this chapter has covered how to understand the recorded code, how to access VBA help for any word, and how to use the excellent VBA debugging tools to step through your code. The remainder of this chapter presents seven tips to use when cleaning up recorded code.

Tip 1: Don't select anything

Nothing screams "recorded code" more than having code that selects things before acting on them. This makes sense in a way: In the Excel interface, you have to select row 1 before you can make it bold.

However, this is rarely done in VBA. There are a couple of exceptions to this rule. For example, you need to select a point on a chart before you can change its properties.

To streamline the code the macro recorder gives you, in many cases, you can remove the part of the code that performs the selection. The following two lines are macro recorder code before it has been streamlined:

```
Cells.Select
Selection.Columns.AutoFit
```

You can streamline the recorded code so it looks like this:

```
Cells.Columns.AutoFit
```

There are a couple of advantages to doing this streamlining. First, there will be half as many lines of code in your program. Second, the program will run faster because Excel does not have to redraw the screen after the lines that perform the selection.

After recording code, you can do this streamlining by highlighting the code from before the word `Select` at the end of one line all the way to the dot after the word `Selection` on the next line. Then, press Delete (see Figures 2-17 and 2-18).

```
    Range("E11:G11").Select
    Rows("1:1").Select
    Selection.Font.Bold = True
    Rows("11:11").Select
    Selection.Font.Bold = True
    Selection.CurrentRegion.Select
    Selection.Columns.AutoFit
End Sub
```

FIGURE 2-17 Select the part of the code highlighted here...

```
    Selection.End(xlDown).Select
    Selection.Offset(1, 0).Select
    Range("A11").FormulaR1C1 = "Total"
    Range("E11").FormulaR1C1 = "=SUM(R[-9]C:R[-1]C)"
    Range("E11").AutoFill Destination:=Range("E11:G11"), Type:=xlFillDefault
    Rows("1:1").Font.Bold = True
    Rows("11:11").Font.Bold = True
    Range("A1").CurrentRegion.Columns.AutoFit
End Sub
```

FIGURE 2-18 ...and press the Delete key. This is Cleaning Up Recorded Macros 101.

Tip 2: Use `Cells(2,5)` because it's more convenient than `Range("E2")`

The macro recorder uses the `Range()` property frequently. If you follow the macro recorder's example, you will find yourself building a lot of complicated code. For example, if you have the row number for the total row stored in a variable `TotalRow`, you might try to build this code:

```
Range("E" & TotalRow).Formula = "=SUM(E2:E" & TotalRow-1 & ")"
```

In this code, you are using concatenation to join the letter *E* with the current value of the `TotalRow` variable. This works, but eventually, you have to refer to a range where the column is stored in a variable. Say that `FinalCol` is 10, which indicates column J. The column in the `Range` property must always be a letter, so you have to do something like this:

```
FinalColLetter = MID("ABCDEFGHIJKLMNOPQRSTUVWXYZ",FinalCol,1)
Range(FinalColLetter & "2").Select
```

Alternatively, perhaps you could do something like this:

```
FinalColLetter = CHR(64 + FinalCol)
Range(FinalColLetter & "2").Select
```

These approaches work for the first 26 columns but fail for the remaining 99.85 percent of the columns.

You could start to write 10-line functions to calculate that the column letter for column 15896 is WMJ, but it is not necessary. Instead of using `Range("WMJ17")`, you can use the `Cells(Row,Column)` syntax.

Chapter 3, "Referring to ranges, names, and tables," covers this topic in complete detail. However, for now, you need to understand that `Range("E10")` and `Cells(10, 5)` both point to the cell at the intersection of the fifth column and the tenth row. Chapter 3 also shows you how to use `.Resize` to point to a rectangular range. `Cells(11, 5).Resize(1, 3)` is `E11:G11`.

Tip 3: Use more reliable ways to find the last row

It is difficult to trust data from just anywhere. If you are analyzing data in Excel, remember that the data can come from who-knows-what system written who-knows-how-long-ago. The universal truth is that eventually, some clerk will find a way to break the source system and enter a record without an invoice

number. Maybe it will take a power failure to do it, but invariably, you cannot count on having every cell filled in.

This is a problem when you're using the End+down arrow shortcut. This key combination does not take you to the last row with data in the worksheet. It takes you to the last row with data in the current range. In Figure 2-19, pressing End+down arrow would move the cursor to cell A7 rather than the true last row with data.

One better solution is to start at the bottom of the worksheet and look for the first non-blank cell by using this:

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

	A	B	C	D
1	Heading	Heading	Heading	Heading
2	Data	Data	Data	Data
3	Data	Data	Data	Data
4	Data	Data	Data	Data
5	Data	Data	Data	Data
6	Data	Data	Data	Data
7	Data	Data	Data	Data
8		Data	Data	Data
9	Data	Data	Data	Data
10	Data	Data	Data	Data

FIGURE 2-19 End+down arrow fails in the user interface if a record is missing a value. Similarly, `End(xlDown)` fails in Excel VBA.

This method could fail if the very last record happens to contain the blank row. If the data is dense enough that there will always be a diagonal path of non-blank cells to the last row, you could use this:

```
FinalRow = Cells(1,1).CurrentRegion.Rows.Count
```

If you are sure that there are not any notes or stray activated cells below the data set, you might try this:

```
FinalRow = Cells(1, 1).SpecialCells(xlLastCell).Row
```

The `xlLastCell` property is often wrong. Say that you have data in A1:F500. If you accidentally press Ctrl+down arrow from A500, you will arrive at A1048576. If you then apply Bold to the empty cell, it becomes activated. Or, if you type **Total** and then clear the cell, it becomes activated. At this point, `xlLastCell` will refer to F1048576.

Another method is to use the Find method:

```
FinalRow = Cells.Find("*", SearchOrder:=xlByRows, _
 SearchDirection:=xlPrevious).Row
```

You will have to choose from these various methods based on the nature of your data set. If you are not sure, you could loop through all the columns. If you are expecting seven columns of data, you could use this code:

```
FinalRow = 0
For i = 1 to 7
  ThisFinal = Cells(Rows.Count, i).End(xlUp).Row
  If ThisFinal > FinalRow then FinalRow = ThisFinal
Next i
```

Note `ListObjects`—Excel tables created by selecting Insert | Table—have their own set of properties that make some things, such as finding the last row used in the table, a lot easier. See "Referencing tables aka `ListObjects`" in Chapter 3 for more information.

Tip 4: Use variables to avoid hard-coding rows and formulas

The macro recorder never records a variable. Variables are easy to use, but just as in BASIC, a variable can remember a value. Variables are discussed in more detail in Chapter 4, "Laying the groundwork with variables and structures."

It is recommended that you set the last row that contains data to a variable. Be sure to use meaningful variable names such as `FinalRow`:

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

When you know the row number of the last record, put the word *Total* in column A of the next row:

```
Cells(FinalRow + 1, 1).Value = "Total"
```

You can even use the variable when building this formula, which totals everything from E2 to the `FinalRow` of E:

```
Cells(FinalRow + 1, 5).Formula = "=SUM(E2:E" & FinalRow & ")"
```

Tip 5: Use R1C1 formulas that make your life easier

The macro recorder often writes formulas in an arcane R1C1 style. However, most people change the code back to use a regular A1-style formula. After reading Chapter 6, "R1C1 style formulas," you'll understand that there are times when you can build an R1C1 formula that is much simpler than the corresponding A1-style formula. By using an R1C1 formula, you can add totals to all three cells in the total row with the following:

```
Cells(FinalRow+1, 5).Resize(1, 3).FormulaR1C1 = "=SUM(R2C:R[-1]C)"
```

Tip 6: Copy and paste in a single statement

Recorded code is notorious for copying a range, selecting another range, and then doing an `ActiveSheet.Paste`. The `Copy` method, as it applies to a range, is actually much more powerful. You can specify what to copy and also specify the destination in one statement.

Here's the recorded code:

```
Range("E14").Select
Selection.Copy
Range("F14:G14").Select
ActiveSheet.Paste
```

Here's better code:

```
Range("E14").Copy Destination:=Range("F14:G14")
```

Tip 7: Use `With...End With` to perform multiple actions

If you are making the total row bold with double underline, a larger font, and a special color, you might get recorded code like this:

```
Range("A14:G14").Select
Selection.Font.Bold = True
Selection.Font.Size = 12
Selection.Font.ColorIndex = 5
Selection.Font.Underline = xlUnderlineStyleDoubleAccounting
```

For four of these lines of code, VBA must resolve the expression `Selection.Font`. Because you have four lines that all refer to the same object, you can name the object once at the top of a `With` block. Inside the `With...End With` block, everything that starts with a period is assumed to refer to the `With` object:

```
With Range("A14:G14").Font
  .Bold = True
  .Size = 12
  .ColorIndex = 5
  .Underline = xlUnderlineStyleDoubleAccounting
End With
```

See Chapter 4 for more information on using `With...End With` blocks.

Case study: Putting it all together—Fixing the recorded code

Using the seven tips discussed in the preceding section, you can convert the recorded code from Chapter 1 into efficient, professional-looking code. Here is the code as recorded by the macro recorder at the end of Chapter 1:

```
Sub FormatInvoice3()
'ImportInvoice Macro
Workbooks.OpenText Filename:="C:\Data\invoice.txt", Origin:=437, _
  StartRow:=1, DataType:=xlDelimited, TextQualifier:=xlDoubleQuote, _
  ConsecutiveDelimiter:=False, Tab:=False, Semicolon:=False, _
  Comma:=True, Space:=False, Other:=False, FieldInfo:=Array( _
  Array(1, 3), Array(2, 1), Array(3, 1), Array(4, 1), _
  Array(5, 1), Array(6, 1), Array(7, 1)), TrailingMinusNumbers:=True
Selection.End(xlDown).Select
ActiveCell.Offset(1, 0).Range("A1").Select
ActiveCell.FormulaR1C1 = "Total"
ActiveCell.Offset(0, 4).Range("A1").Select
Selection.FormulaR1C1 = "=SUM(R2C:R[-1]C)"
Selection.AutoFill Destination:=ActiveCell.Range("A1:C1"), Type:= _
  xlFillDefault
ActiveCell.Range("A1:C1").Select
ActiveCell.Rows("1:1").EntireRow.Select
ActiveCell.Activate
Selection.Font.Bold = True
Application.Goto Reference:="R1C1:R1C7"
Selection.Font.Bold = True
Selection.CurrentRegion.Select
Selection.Columns.AutoFit
End Sub
```

Follow these steps to clean up the recorded macro code:

1. Leave the `Workbook.OpenText` lines alone; they are fine as recorded.
2. Note that the following line of code attempts to locate the final row of data so that the program knows where to enter the total row:

   ```
   Selection.End(xlDown).Select
   ```

3. You do not need to select anything to find the last row. It also helps to assign the row number of the final row and the total row to a variable so that they can be used later. To handle the unexpected case in which a single cell in column A is blank, start at the bottom of the worksheet and go up to find the last-used row:

   ```
   'Find the last row with data. This might change every day
   FinalRow = Cells(Rows.Count, 1).End(xlUp).Row
   TotalRow = FinalRow + 1
   ```

   Note that these lines of code enter the word `Total` in column A of the total row:

   ```
   ActiveCell.Offset(1, 0).Range("A1").Select
   ActiveCell.FormulaR1C1 = "Total"
   ```

Better code uses the `TotalRow` variable to locate where to enter the word `Total`. Again, there is no need to select the cell before entering the label:

```
'Build a Total row below this
Cells(TotalRow,1).Value = "Total"
```

4. Note that these lines of code enter the `Total` formula in column E and copy it to the next two columns:

```
ActiveCell.Offset(0, 4).Range("A1").Select
Selection.FormulaR1C1 = "=SUM(R2C:R[-1]C)"
Selection.AutoFill Destination:=ActiveCell.Range("A1:C1"), Type:= _
  xlFillDefault
ActiveCell.Range("A1:C1").Select
```

   There is no reason to do all this selecting. The following line enters the formula in three cells:

```
Cells(TotalRow,5).Resize(1, 3).FormulaR1C1 = "=SUM(R2C:R[-1]C)"
```

   (The R1C1 style of formulas is discussed in Chapter 6.)

5. Note that the macro recorder selects a range and then applies formatting:

```
ActiveCell.Rows("1:1").EntireRow.Select
ActiveCell.Activate
Selection.Font.Bold = True
Application.Goto Reference:="R1C1:R1C7"
Selection.Font.Bold = True
```

   There is no reason to select before applying the formatting. The preceding five lines can be simplified to the two lines below. These two lines perform the same action and do it much more quickly:

```
Cells(1, 1).Resize(1, 7).Font.Bold = True
Cells(TotalRow, 1).Resize(1, 7).Font.Bold = True
```

6. Note that the macro recorder selects all cells before doing the `AutoFit` command:

```
Selection.CurrentRegion.Select
Selection.Columns.AutoFit
```

   There is no need to select the cells before doing the `AutoFit`:

```
Cells(1, 1).Resize(TotalRow, 7).Columns.AutoFit
```

   (The `Resize` method is discussed in Chapter 3.)

7. Note that the macro recorder adds a short description to the top of each macro:

```
'ImportInvoice Macro
```

   You have changed the recorded macro code into something that will actually work, so you should feel free to add your name as author to the description and mention what the macro does:

```
'Written by Bill Jelen. Import invoice.txt and add totals.
```

Here is the final macro with a declaration of variables (see Chapter 4), with all the changes discussed above:

```
Sub FormatInvoiceFixed()
'Written by Bill Jelen. Import invoice.txt and add totals.
Dim TotalRow as Long, FinalRow as Long
Workbooks.OpenText Filename:="C:\Data\invoice.txt", Origin:=437, _
  StartRow:=1, DataType:=xlDelimited, TextQualifier:=xlDoubleQuote, _
  ConsecutiveDelimiter:=False, Tab:=False, Semicolon:=False, _
  Comma:=True, Space:=False, Other:=False, FieldInfo:=Array( _
  Array(1, 3), Array(2, 1), Array(3, 1), Array(4, 1), _
  Array(5, 1), Array(6, 1), Array(7, 1))
FinalRow = Cells(Rows.Count, 1).End(xlUp).Row
TotalRow = FinalRow + 1
Cells(TotalRow, 1).Value = "Total"
Cells(TotalRow, 5).Resize(1, 3).FormulaR1C1 = "=SUM(R2C:R[-1]C)"
Cells(TotalRow, 1).Resize(1, 7).Font.Bold = True
Cells(1, 1).Resize(1, 7).Font.Bold = True
Cells(1, 1).Resize(TotalRow, 7).Columns.AutoFit
End Sub
```

Next steps

By now, you should know how to record a macro. You should also be able to use Help and debugging to figure out how code works. This chapter provides seven tools for making the recorded code look like professional code.

The next chapters go into more detail about referring to ranges, looping, and the crazy but useful R1C1 style of formulas that the macro recorder loves to use.

Index

Symbols

Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X