THE EXPERT'S VOICE® IN EXCEL

Advanced Excel Essentials

TAKE YOUR EXCEL SKILLS TO THE NEXT LEVEL

Jordan Goldmeier

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

About the Author xiii

About the Technical Reviewer xv

Acknowledgments xvii

■ Part I: Core Advanced Excel Concepts 1

■ Chapter 1: Introduction to Advanced Excel Essentials 3

■ Chapter 2: Visual Basic for Applications for Excel, a Refresher 11

■ Chapter 3: Introducing Formula Concepts 31

■ Chapter 4: Advanced Formula Concepts 49

■ Chapter 5: Working with Form Controls 67

■ Part II: A Real World Example 93

■ Chapter 6: Getting Input from Users 95

■ Chapter 7: Storage Patterns for User Input 115

■ Chapter 8: Building for Sensitivity Analysis 137

■ Chapter 9: Perfecting the Presentation 165

Index 191

PART I

Core Advanced Excel Concepts

In this part, I'll review the core concepts that make up the essentials of advanced Excel.

Chapter 1 explains what is meant by advanced Excel development, and how this book differs from many others. For instance, several books place significant emphasis on Visual Basic for Applications code, believing macros to be the most important feature of Excel development. This chapter will challenge that notion and present advanced concepts as a product of many different Excel features, including code. Additionally, I discuss the most important required skill—creativity.

Chapter 2 provides a brief Visual Basic for Applications refresher. I'll discuss how best to set up the coding environment to make it conducive to headache-free coding. I'll also challenge conventional coding conventions and propose alternatives that will prove more effective.

Chapter 3 introduces the formula concepts that will be used in this book. The chapter starts with tips that will make your experience developing advanced formulas run more smoothly. I'll then show you how to perform advanced calculations by simply using range operators. You'll develop advanced alternatives to the `IF` function that will prove more powerful in practice and more readable later on. In addition, you'll investigate the full extent of Excel's Boolean logic features.

Chapter 4 continues the discussion of formulas by demonstrating how they can be used with advanced applications. I take you through several examples applying these formula concepts and demonstrate how they can be understood with a little bit of algebra. The chapter concludes by introducing the notion of reusable components, which are spreadsheet mechanics that can be easily reused for other projects.

Chapter 5 shows how advanced capabilities can be built into spreadsheets by using the humble form control. In this chapter, I argue against using ActiveX and UserForms. Instead, you'll rely on the flexibility of form controls combined with the speed and prowess of formulas. Chapter 5 concludes with several practical reusable components that you can start using in your own work right away.

CHAPTER 1

Introduction to Advanced Excel Essentials

I set out to write a book on the *essentials* of Excel development—that is, a book that concisely presents many of the development principles and practices I've discovered through my work and consulting experience.

But whether on purpose or by accident, this book has become something considerably more than that. Indeed, another name for this book could be *A Contrarian's Guide To Excel Development*. You see, this book will push back against the wisdom of other terrific Excel books, including my favorite book, *Professional Excel Development* (Addison-Wesley 2005). To be sure, the information in those books is terrific, and whatever merits this book might achieve, it will likely never come close to the impact of *Professional Excel Development*.

At the same time, much of the information in these books, I believe, is somewhat dated. For instance, let's take the case of Hungarian Notation. Hungarian Notation is a variable naming convention encouraged by virtually all Excel development books. Even if you've never heard of Hungarian Notation, you've likely seen and used it, if you've ever looked at or learned from example code. It basically says a variable's name should start with a prefix of the variable's type. For instance, `lblCaption`, `intCounter`, and `strTitle` are all examples of Hungarian Notation: the `lbl` in `lblCaption` tells us we're working with a Label object; the `int` in `intCounter` tells us we're working with an integer type, and the `str` in `strTitle` tell us we're working with a string type. If you've done any VBA coding before, this is likely not new information.

You might not know this, however: most modern languages have all but abandoned Hungarian Notation. Microsoft's .NET style guidelines, for instance, even discourage its use. More than a decade has passed since Microsoft last recommended Hungarian Notation. I argue that it's time for a more modern naming style, which I introduce in Chapter 2.

But this book is concerned with more than just naming conventions. I argue that we should change the way we think about development. Previous books have placed significant emphasis on user interface with ActiveX objects and UserForms. This book will eschew these bloated controls; rather, this book will show you how to develop complex interactivity using the spreadsheet as your canvas. You'll see that it's easier and provides for more control and flexibility compared to conventional methods from other books.

In addition, I'll place less emphasis on code and a stronger emphasis on formulas (Chapters 3, 4, and 5). Many books have narrowly defined the principles of advanced Excel in terms of VBA code. But formulas can be powerful. And often they can be used in place of VBA code. You might be surprised by how much interactivity you can create without writing a single line of code. And how much quicker your spreadsheet runs because of it.

This book is divided into two parts. Part I (Chapters 1-5) deals with concepts that are likely already familiar to you. Specifically they concern VBA code and formulas—but I present these concepts in new ways. Part II makes up the last four chapters of the book (Chapters 6-9). These chapters apply concepts from Part I to a real-world example product I built in my consulting experience. Futhermore, in Part II, you'll learn how to input form data without making your spreadsheet bloated. You'll also apply some data analytics used in the field of management science.

However, if you learn anything from my book, it should be that the process of development never stops. The most important skill you'll need is creativity. Just as I saw different ways to approach a problem than my predecessors, so too should you analyze what's being presented to you. Undoubtedly, you'll find even better approaches than I did. I don't expect everyone to agree with my approaches, but what's important is that you understand them, so you can see what works, what doesn't, and why. Because you won't become an advanced Excel developer through rote memorization of the material presented herein; you must learn to think like an advanced developer. This book will teach you the essentials of doing just that.

What to Expect from this Book

This is not a beginner level book. I assume you have intermediate level experience with formulas and Visual Basic for Applications. At the very least, you should be able to understand and write both formulas and code. Complete mastery isn't necessary; because the topics presented in this book are somewhat new, a mastery in these topics might not even help you. All that being said, if you're an experienced Excel user—and you have the aptitude and thirst to learn new things—there's no reason you won't be successful in reading this book! Again, the most important (and cherished) skill that will guarantee your success is creativity.

What's considered "advanced" may mean different things to different people. Here, we're interested in the principles that help us become better spreadsheet users and developers. That said, this book will make use of Excel features such as formulas, tables, conditional formatting, Visual Basic for Applications code, form controls, and charts. For the most part, I will present a brief refresher on what these features do and how they are used. However, you'll find this book moves at a quicker pace than beginner level treatments for these items. Features such as PivotTables, PowerPivot, Power Map, and data tables are not discussed in this book. But you'll find that the principles presented in these pages are extendable to these topics.

Indeed, this book is most concerned with teaching Excel development as first principles. I will explain what they are and how best they are used in practice. Once you learn underlying concepts, extending their use into applications becomes trivial.

Example Files Used in This Book

This book comes with many examples as a complement to the material presented herein. The example files are organized by chapter. Whenever there is a corresponding example file for the material presented, I'll provide you the name of the example file in the text. All example files are freely available to download from the book's Apress web page (`www.apress.com/9781484207352`). The files are designed to work in Excel 2007 and newer.

The Two Most Important Principles

There are many different ideas and concepts presented in this book. But I'll be daring and attempt to sum them up as two key concepts:

1. When it make sense, do more with less.
2. Break every rule.

■ **Note** The two most important principles are (1) when it makes sense, do more with less, and (2) break every rule.

When It Makes Sense, Do More with Less

You don't need VBA to do everything. Many times, the reason a spreadsheet is slow is because there is too much reliance on code. Similarly, too many formulas—especially volatile functions like `OFFSET` and `INDIRECT`—will almost always slow down a spreadsheet. There are better alternatives to these methods. Often, they require less code and can get more done.

However, we should be wary of brevity for the sake of it. Bill "MrExcel" Jelen and I have a friendly disagreement[1] on whether to use `Option Explicit` in your code. He says he doesn't need it because he always writes perfect code to start with—and that its use needlessly adds more lines of code. I, of course, respectfully disagree. I strongly encourage you to use `Option Explicit.` `Option Explicit` requires that you declare your variables before they're used. That means that you cannot introduce a new variable in your code on the fly. Listing 1-1 shows code without `Option Explicit`; Listing 1-2 shows code with `Option Explicit`.

Listing 1-1. No Option Explicit

```
Public Sub MyResponse()
    ResponseMessage = "Code Executed Successfully!"
    MsgBox ResponseMessage
End Sub
```

Listing 1-2. With Option Explicit

```
Option Explicit

Public Sub MyResponse()
    Dim ResponseMessage as String

    ResponseMessage = "Code Executed Successfully!"
    MsgBox ResponseMessage
End Sub
```

Bill argued using `Option Explicit` required at least one additional line of code for every variable. And it might appear Listing 1-1 is indeed doing more (or at least *the same*) with less code. But, as I show in Chapter 2, not using `Option Explicit` might be more trouble than it is worth. Debugging is much harder without `Option Explicit`, and not using it even encourages sloppy code. From my standpoint, leaving out `Option Explicit` (and the required variable declaration) is simply getting less done with less code. But however you feel on this particular issue, it's worth testing your opinion against that first principle: ask yourself, am I really doing more with less?

Break Every Rule

I truly believe, and stand by, the material presented in this book. But I would have never discovered any of it without departing from conventional wisdom. Again, I'll keep hammering this point until I am blue in the face: the most important takeaway from this book is creativity. And you cannot be creative without pushing a few boundaries. Don't be scared to crash a spreadsheet or two in the pursuit of learning.

You'll see in later chapters that some techniques won't always be the best choice for every scenario. For instance, a complex formula that is much faster in practice than a conventional formula might be useless if you must share your spreadsheet and you're the only one who understands it. There will always be an economy between formula readability and utility. I present complex formulas in this book, but I also argue that readability should be a factor in choosing when and where to use them.

[1]Watch Bill and I fight about this on Excel.TV: `www.youtube.com/watch?v=yJRLzN3Dzmw`.

Most important, you shouldn't be satisfied with Excel's perceived limitations. Over the last several years, I've been blown away by what I've seen others accomplish with Excel. There is a thriving online community dedicated to helping people realize their imaginations with spreadsheets. Whenever I need inspiration, I look to the community.

For your own consideration, I'll provide two examples of my own work that show what can be done with Excel when we think creatively. Figure 1-1 shows a three dimensional maze I created. It might surprise you to learn there is very little code involved. And the "maze" is simply an area chart formatted to look like a three dimensional plane.

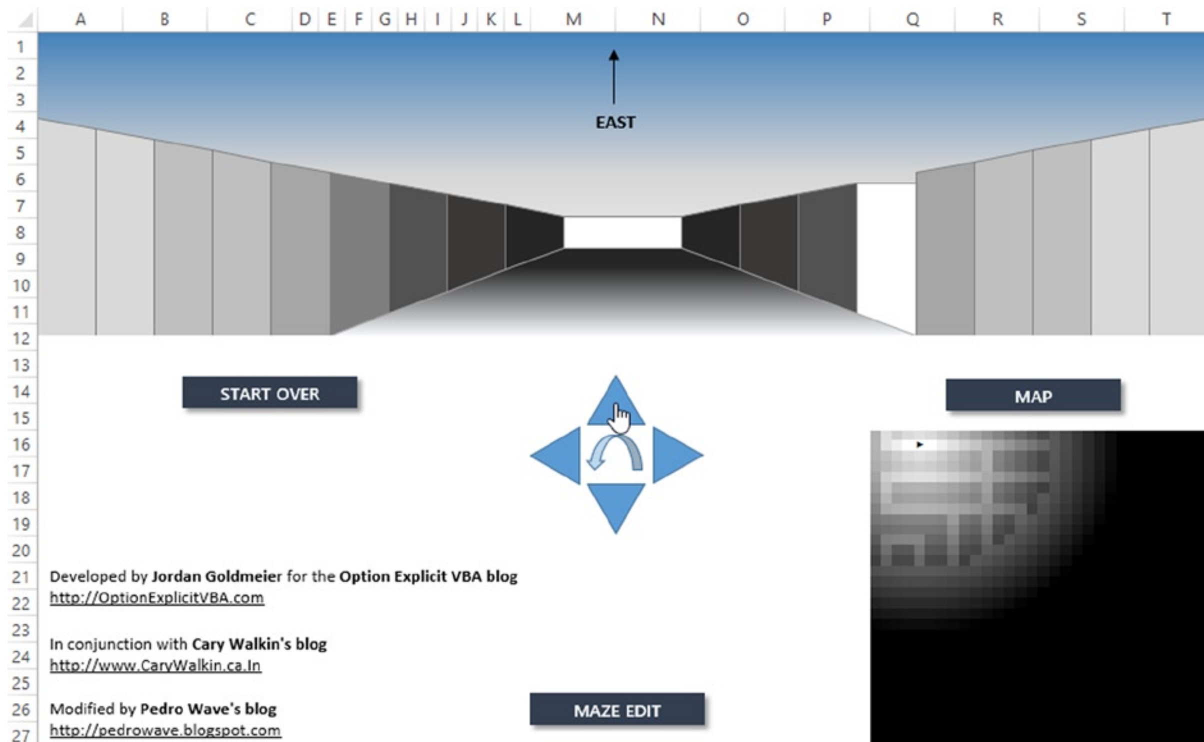

Figure 1-1. *A three dimensional maze, made with Excel*

The second item I would like to present is a periodic table of elements with Excel, shown in Figure 1-2. The periodic table uses a mouseover capability. When the user hovers their mouse over a cell, a macro is executed that updates information about the element. However, the macro uses only a few lines of code, and besides that update, the functionality is largely driven by formula functions. Moreover, that mouseover capability is one I discovered by accident. Before I first wrote about it on my blog, it had been considered impossible.

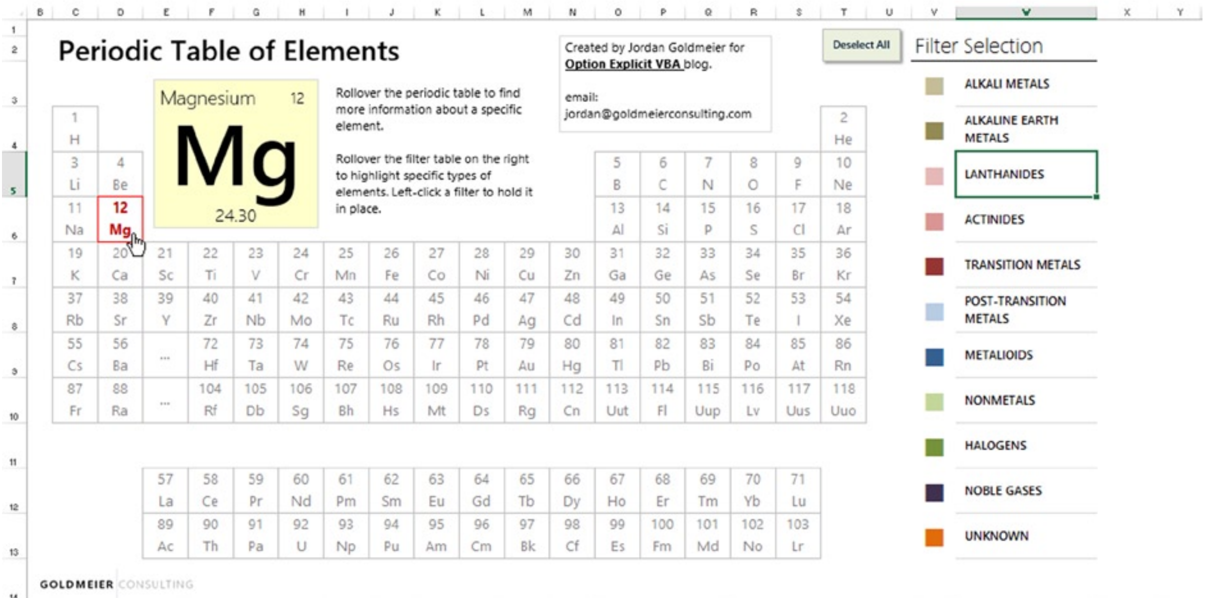

Figure 1-2. *A periodic table of elements with interactivty previously thought impossible with Excel*

Both the three dimensional maze and periodic table are available for you to investigate in the project files included with this book. While it's beyond the scope of this book to explain in detail how these particular spreadsheets were created, they are the direct product of the material I present in the rest of the book. However, if you're interested in reading how these items were developed, see the links in the sidebar.

LINKS ON DEVELOPING A MAZE AND MOUSE OVER MECHANISM

How to Create a Rollover Effect in Excel: Execute a Macro When Your Mouse is Over a Cell

http://optionexplicitvba.blogspot.com/2011/04/rollover-b8-ov1.html

Roll Over Tooltips and Web Actions on a Microsoft Excel Dashboard

www.clearlyandsimply.com/clearly_and_simply/2012/11/roll-over-tooltips-and-web-actions-on-a-microsoft-excel-dashboard.html

Development Principles for Excel Games and Applications

http://optionexplicitvba.com/2013/09/16/development-principles-for-excel-games-and-applications/

Your First Maze

http://optionexplicitvba.com/2013/09/17/your-first-maze-2/

Available Resources

As I said in the previous section, sometimes you need some inspiration to help get you going. Here's a list of resources I use regularly.

Google

Google...Google...Google! Google is your best friend. If you're ever stuck on a problem, simply ask Google the same way you might your friend. Usually, you'll find the results in Excel forums where folks have asked the very same questions.

Chandoo

This site, by Purna "Chandoo" Duggirala, is a phenomenal resource for every Excel developer, from novice to professional. Chandoo covers many topics including dashboards, VBA, data visualization, and formula techniques. His site is also host to a thriving online forum community.

`www.chandoo.org`

Cleary and Simply

Clearly and Simply is a site by Robert Mundigl. The site is mainly focused on dashboards and data visualization techniques with Excel and Tableau.

`www.ClearlyAndSimply.com`

Contextures

Debra Dalgleish runs the Contextures web site, which focuses on Excel development and dashboards, particularly with PivotTables. Her approach to dashboards and the use of PivotTables is different from mine, but well worth a read. She is also the author of these Apress Books:

- *Excel Pivot Tables Recipe Book: A Problem-Solution Approach*
- *Beginning PivotTables in Excel 2007: From Novice to Professional*

`www.contextures.com`

Excel Hero

Excel Hero was created by Daniel Ferry. While his blog is not very active anymore, you will find his older content incredibly useful. Several of his articles have served as the inspiration for the content found in these pages.

`www.ExcelHero.com`

Peltier Tech

Jon Peltier is a chartmaster. His web site is full of charting tutorials and examples. He provides sage wisdom on data visualization and proper data analysis. His web site covers every conceivable thing you might want to do with a chart in Excel.

`www.peltiertech.com`

The Last Word

Above all, advanced development is about thinking creatively. You'll see this in practice in the chapters to come. Because some of the material is new, it may appear challenging at first. You may even find yourself frustrated at times. In these moments, it's best to take a break for a moment, find your bearings, and start from the beginning of the section in which you left off. The material is complex, but well within your grasp. I urge you to push through to the end of the book. The material is worth it; but more important, you're worth it. What will you learn in this book will distinguish you. We're only still scratching the surface of what Excel can do. By the time you're finished with this book, you'll be developing work that might even surprise you.

CHAPTER 2

Visual Basic for Applications for Excel, a Refresher

Of course, no advanced book on developing anything in Excel would be complete without a chapter on the interpreter language housed within Excel, Visual Basic for Applications–or better known by its shorthand moniker, VBA.

This chapter won't be an introduction to VBA but rather a review of VBA programming techniques and development principles found in this book and practiced throughout most of my career. What follows may appear unconventional, at first. Indeed, it may differ somewhat from what you've been previously taught. However, I don't leave you with a few instructions and no guidance. Instead, I'll explain in detail why I believe what I believe—and why you should believe as I do. If you find that you don't—and I certainly welcome disagreement—consider the other important—actually, more important—takeaway from this chapter: the code choices and styles we use should always follow from a set of principles, guidelines, and convention. When you code, do so with structure and meaning. Know *why* you believe what you believe.

But, the most important thing to do right now is to ready yourself to begin coding. This requires that you set the right conditions in your coding environment.

Making the Most of Your Coding Experience

I tend to get more done when I'm less frustrated. I'll be so daring to suggest you're probably the same way. And let's not kid ourselves: coding in VBA can be a frustrating experience. For instance, have you ever been halfway through writing an IF statement and then realized you needed to fix something on another line? So you click that other line and Excel stops everything to pop up a message box saying that you've written a syntax error, like in Figure 2-1. Chances are, you already knew that. In fact, you wanted to change an earlier line in the code to prevent another error from happening.

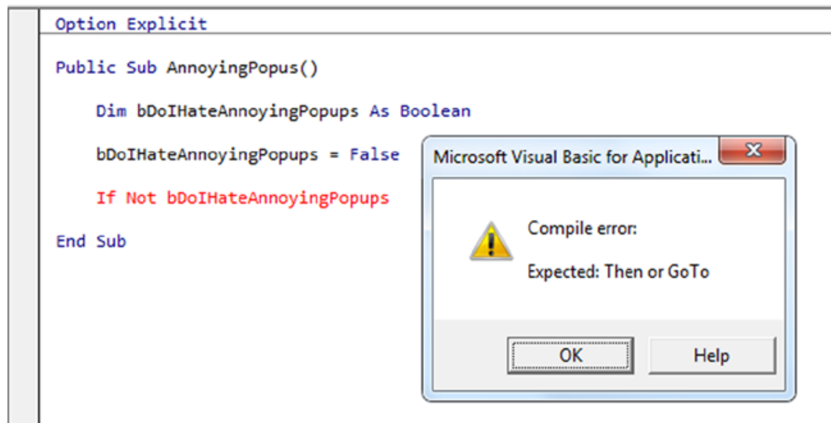

Figure 2-1. *That all-too-annoying popup error box telling you what you likely already know*

Tell Excel: Stop Annoying Me!

I mean, nobody's perfect, but you don't need this popup ruining your coding flow every time you click to another line. So, save yourself from unnecessary popups by disabling Auto Syntax Check from the Options dialog box, which you access by selecting Tools > Options (see Figure 2-2). This will only disable the popup. The offending syntax error is still highlighted in red—in other words, you don't lose any functionality, just the annoyance.

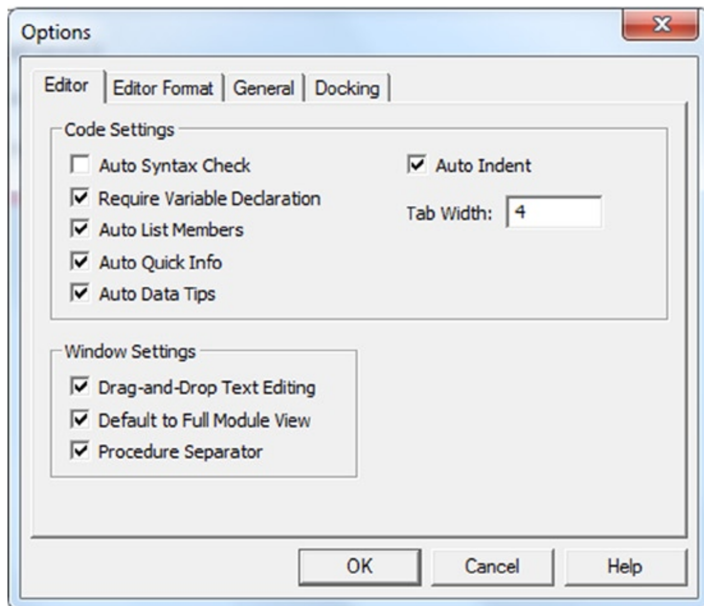

Figure 2-2. *Uncheck Auto Syntax Check for distraction-free coding*

Make Loud Comments

If you comment your code regularly—and you should—you've probably noticed comments don't "stand out" very much. In fact, I'll be the first to admit I've gone through code and missed comments because they've "blended in" with their surroundings. Figure 2-3 shows perhaps a more extreme example involving rather busy code, but the point remains: the two comment markers (') I've placed in the routine are not easily or immediately found.

```
Public Sub CommentTest()
    MsgBox "1"
    MsgBox "2          "
    MsgBox "3     "
    MsgBox "4                "
    MsgBox "        5   "
    MsgBox "6   " '
    MsgBox "    7   "
    MsgBox "       8   "
    MsgBox "               9   "
    MsgBox "   10   "
    MsgBox "     11   "
    MsgBox "      12   "
    MsgBox "14      " '
    MsgBox "    15   "
    MsgBox "16  "
End Sub
```

Figure 2-3. *Comment markers at 6 and 14 blend in with the code*

Luckily, you don't have to use the preset colors. In fact, you can make the comments stand out. Go back to the Options dialog box from the Tools menu. Click the Editor Format tab and select Comment Text from the Code Colors list box. Below the list box you can specify the foreground and background color, which are the text color and highlight properties, respectively (see Figure 2-4). Personally, I like using a dark blue foreground and light blue background (see Figure 2-5). You'll have to try this on your own to get the full effect; to that end, and to preserve the formatting guidelines of this book, the highlight does not appear in the code listings throughout the book.

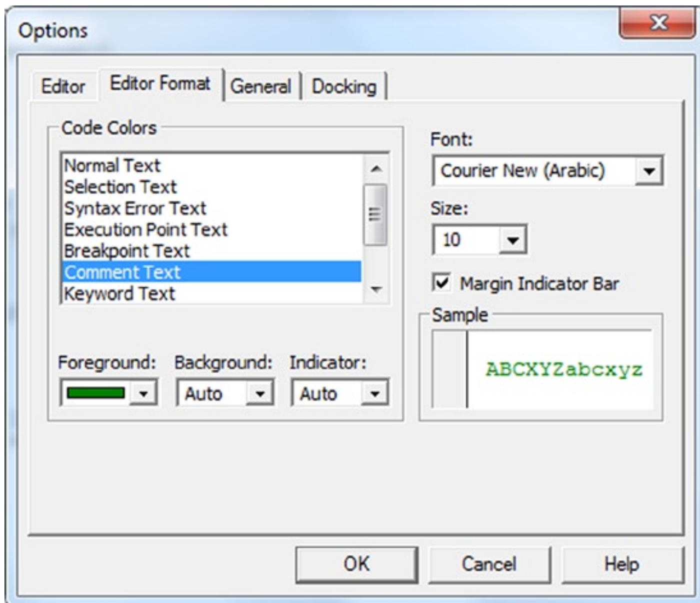

Figure 2-4. *The Editor Format dialog box*

```
Public Sub CommentTest()
    MsgBox "1"
    MsgBox "2        "
    MsgBox "3    "
    MsgBox "4              "
    MsgBox "        5  "
    MsgBox "6  " '
    MsgBox "    7  "
    MsgBox "       8  "
    MsgBox "               9  "
    MsgBox "   10  "
    MsgBox "     11  "
    MsgBox "      12  "
    MsgBox "14     " '
    MsgBox "    15  "
    MsgBox "16  "
End Sub
```

Figure 2-5. *Let your comments be heard with bold colors*

Pick a Readable Font

Leave that Options dialog box open because you'll need it once more. By default, Excel uses Courier New (Figure 2-6) as its default coding font. Again, this font, like the comment style defaults, doesn't emphasizes the clear readability. I prefer the font Consolas shown in Figure 2-7 because I think it does a much a better job in this regard.

```
Option Explicit

Public Sub CommentTest()
    MsgBox "Try Reading this Font."
End Sub
```

Figure 2-6. *Sample code with Courier New as the font*

```
Option Explicit

Public Sub CommentTest()
    MsgBox "Try Reading this Font."
End Sub
```

Figure 2-7. *More readable text with Consolas*

You can change the font by selecting Normal Text from the list box (Figure 2-4) and using the font dropdown on the side of the dialog box. Excel gives you lots of fonts to choose from, but the best fonts with which to code are those of fixed width. So if you choose something other than Consolas or Courier New, make sure to pick a readable, fixed-width font.

Start Using the Immediate Window, Immediately

The Immediate window is like a handy scratchpad with many uses. If the Immediate window is not already open, go to View ➤ Immediate Window in the Visual Basic Editor. You can type calculations and expressions directly into the Immediate window using the `print` keyword. Figure 2-8 provides some examples of typing directly into the Immediate window.

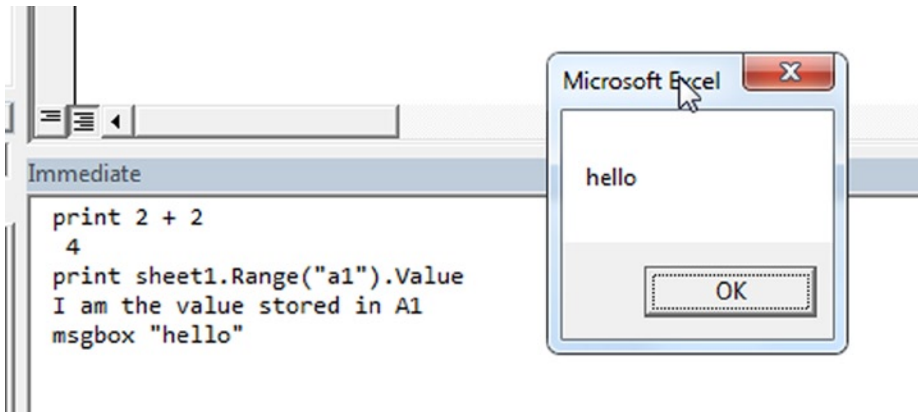

Figure 2-8. *The Immediate window*

In addition, you can also print the response of a loop or method directly into the Immediate window. To do this, use `Debug.Print`. Listing 2-1 shows you how.

Listing 2-1. Using Debug.Print to Write to the Immediate Window While in Runtime

```
For i = 1 to 100
    Debug.Print "Current Iteration: " & i
Next i
Debug.Print "Loop finished."
```

Opt for Option Explicit

VBA doesn't require you declare your variables before using them—that is, unless you place the words `Option Explicit` at the top of your code module. Without `Option Explicit`, the `For` loop from Listing 2-1 would run without problems. When you use `Option Explicit`, you must declare all variables before they are used. In Listing 2-2, I've used the `Dim` keyword to declare the integer `i`.

Listing 2-2. A For-Next Loop with Declared Variables

```
Dim i as Integer
For i = 1 to 100
    Debug.Print "Current Iteration: " & i
Next i
Debug.Print "Loop finished."
```

If you forgo `Option Explicit`, as I did in the first instance, Excel will simply create the variable `i` for you. However, that `i` won't be an integer; rather it will be of a *variant* type. This may not sound like such a bad thing at first, but letting Excel simply make variables for you is a recipe for trouble. What if you misspell a variable, like `RecordCount`, as I've done in Listing 2-3?

Listing 2-3. An Example of a Variable Created on the Spot Because Option Explicit Wasn't Used

```
RecordCount = 1
Msgbox RecordCout
```

Excel won't alert you to an error. Instead, it will simply create `RecordCout` as a new variable. Do you trust your ability to find misspellings in your code quickly?

In practice, I've found using `Option Explicit` alleviates many potential headaches. So do yourself a favor, in the Option dialog box (Tools ➤ Options), check Require Variable Declaration to Excel to automatically (and proudly) display `Option Explicit` at the top of every module. And when the error in Figure 2-9 appears, give yourself a pat on the back for not having to scour your code to find your misspellings.

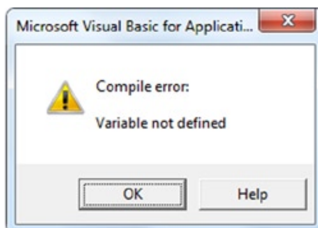

Figure 2-9. *Breath a sigh of relief! You have Option Explicit on the case!*

Seriously, I can't tell you how important `Option Explicit` is. I'd repeat "Always use Option Explicit!" 1,000 times here if I could. But I'll just let Excel do it for me instead. Paste the following formula into an empty cell before moving to the next section.

```
=REPT("Always use Option Explicit! ",1000)
```

Naming Conventions

A naming convention is a common identification system for variables, constants, and objects. By definition, then, a good naming convention should be sufficiently descriptive about the content and nature of the thing named. In the next subsections, I'll talk about two naming conventions. The first, Hungarian Notation, is the most common notation used for VBA coding. Indeed, I'm unaware of any book that has argued against its use—that is, until now. The second, my preferred notation, is what I call "loose" CamelCase notation, and it's similar to the standard for just about all modern object-oriented languages.

Hungarian Notation

In this section, I'll talk about Hungarian Notation. In this notation, the variable name consists of a prefix—usually an abbreviated description the variable's type—followed by one or two words describing the variable's function (e.g. its reason for existing). For example, in Listing 2-4, the "`s`" before `Title` is used to indicate the variable is of `String` type. The term "title," as I'm sure you can guess, describes to the string's *function*—in other words, its reason for existing.

Listing 2-4. An Example of Hungarian Notation

```
Dim sTitle as String
sTitle = “The new spreadsheet.!”
```

Table 2-1 shows some suggested prefixes for common variables and classes.

Table 2-1. *Prefixes Suggested by Hungarian Notation*

Prefix	Data Type
B	Boolean
D	Double
I	Integer
S	String
V	Variant
Rng	Excel.Range
Obj	Excel.Object
Chrt	Excel.Chart
Ws	Excel.Worksheet
Wb	Excel.Workbook

In this book, I will discourage the use of Hungarian Notation in your code. I'm not here to tell you that Hungarian Notation is terrible because it does have its uses. For instance, VBA code isn't known for having very strict data type rules. This means you can assign integers to strings without casting from one type to the other. So including the type in a variables name isn't a terrible idea at all.

But much of this type confusion can be resolved by using descriptive and proper variables names, as you'll see in the next few pages. For now, however, it's a good idea to at least familiarize yourself with Hungarian Notation if you haven't done so already. Hungarian Notation is still widely used in VBA to this day, so it's important that you can read it proficiently even if you decide in this moment to never use it again. (Good choice!)

The fact is, Hungarian Notation is old. Indeed, in many ways, it's a relic of a bygone era–namely, the era in which people still used Visual Basic 6.0. (Those were the days, right?) In fact, Microsoft's Design Guidelines for .NET libraries has discouraged its use for more than decade. So what I'm proposing in this next section might feel new, but it's actually been around for quite some time.

"Loose" CamelCase Notation

In this section, I'll talk about loose CamelCase notation as my preferred alternative. CamelCase notation begins with a description (with the first letter in the "lower case," when it's a local, private variable—hence the name "CamelCase") and usually ends with the object type *unabbreviated*. For example, the variable in Listing 2-5 refers to chart on a worksheet for sales.

Listing 2-5. A Demonstration of Camel Back Notation

```
Dim salesChart as Excel.Chart
Set salesChart = Sheet1.ChartObjects(1).Chart
```

I'll be honest and admit I'm not always such a stickler about that lower case descriptor, which is why I call my use of this notation "loose." The important takeaway when using this notation is to use *very descriptive names*. It's unlikely a variable name like `ChartTitle` will be confused for an integer in your code. Whether it's `recordCount` or `RecordCount`, you'll likely understand that count refers to a nonnegative integer.

My rule of thumb is, local primitive types should start with a lower case, if you feel so inclined. Variables that represent objects should end with the object name *unabbreviated*. Notice in Listing 2-5 that the variable name ends with `Chart`. Ranges should end with `Range`, etc.

Descriptive names are important. Use a variable name that describes what the variable does so when you come back to it later, you can remember what you did. If you have a test variable, then (please, for the love of God) call it "test,"; don't just call it "t." It's OK to use `i` in a `For/Next` loop where the `i` is simply an iterator and is not used later in the code, but don't name variables used to count objects with short names like `i,j,k,a,b,c`. Finally, there's really no good reason to use an underscore in your variable names. They're not easier to read.

Named Ranges

As I said above, naming convention goes beyond just VBA. Indeed, a proper naming convention should be applied to all Excel objects, including those that reside on a spreadsheet. Therefore, in this section, I'll talk about naming objects on the spreadsheet in the form of named ranges.

It's rather common to see Excel developers use the prefix "val" to refer to named cell ranges. This prefix is an attempt to extend the Hungarian Notation principles into the physical spreadsheet (as if we haven't already had enough of it!). However, I still prefer a more modern approach. Specifically, what I like to do is combine the name of the tab and the function of the variable in to be object-oriented-like. Figure 2-10 shows a good example of what I mean.

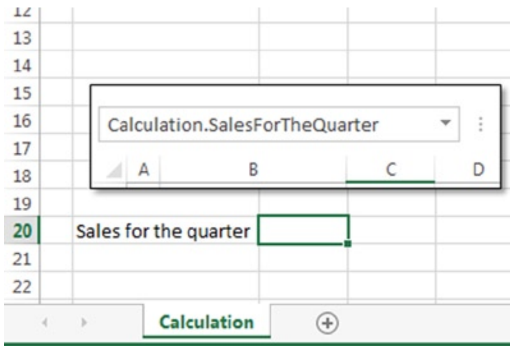

Figure 2-10. *An object-oriented like naming convetion for named ranges*

In Figure 2-10, the name of the tab is combined with the variable. Aside from being more object-oriented-ish, this type of naming brings other distinct advantages. For one, you can more easily and logically group named ranges that exist on the same worksheet tab. In addition, as you'll see in the next section, this type of convention works very well when interfacing between named ranges and VBA.

Sheet Objects

In this section, I'll focus on naming conventions for sheet objects. There's one property of the sheet object that I'm a big fan of changing, and it's the name of the object itself. When you change the name of a worksheet tab on the spreadsheet, you're actually changing the name of the tab (think of it as changing a caption); you are not, in fact, changing the name of the worksheet object itself.

If for nothing else, changing the name of the worksheet object is a great way to clear up confusion when looking at the Project Explorer window. For example, Excel seems to have a problem keeping the names of worksheet tabs and the names of the objects themselves straight, as I'm sure you've noticed before. Take a look at Figure 2-11 to see what I mean.

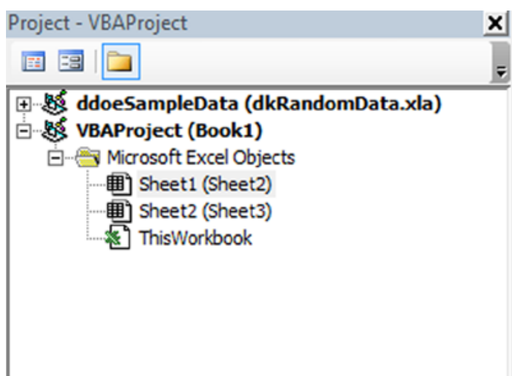

Figure 2-11. *The Project Explorer demonstrating a lack of consistency when it comes to worksheet object and tab names*

The *object* name is the item outside the parenthesis; the tab name is the one inside the parenthesis. If I were to write `MsgBox Sheet1.Name` in the Immediate window, I would see a response of "Sheet2."

To change the name of the object itself, go to the Properties window from within the editor (View ➤ Properties Window, if it's not already visible) and change the line that says (name). In Figure 2-12, my worksheet tab's caption is "Financial Data," so I'm going to change its object name to `FinancialData`.

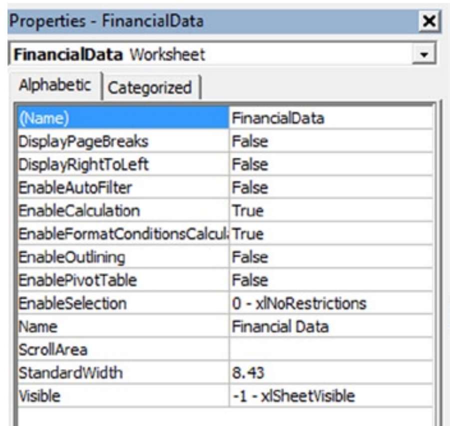

Figure 2-12. *The Properties Explorer showing how to change the worksheet object's name*

YES, I KNOW IT'S CONFUSING

If you look at the Project Explorer window (Figure 2-12, above), you'll see that the worksheet object name comes first and the tab name follows in parenthesis. The Properties Explorer window appears to do just the opposite; the first name in parenthesis, "(name)", refers to the object's name, while the second name item (under Enable Selection) refers to its name as it appears on the tab. Why did Microsoft choose to do it this way? Your guess is as good as mine.

Referencing

In this section, I'll talk about referencing. Referencing refers to interacting with other worksheet elements from within VBA code and also on the worksheet. This is where a good naming convention and proper coding style really makes the difference.

Let's take a made-up named range concerning Cost of Goods Sold. Hungarian Notation proponents would give the named range something like `valCoGS` (CoGS = Cost of Goods Sold). The notation I suggest would combine the tab name with a nicely descriptive title (you could make it shorter if you'd like, but I like long titles), something like `IncomeStatement.CostOfGoodsSold`. So let's take a look at why you might prefer a long named range such as this in the next section.