overview of basics of VBA

section on VBA topics should include:

- basics of variables
  - common variable types
  - difference between value and reference types
- basics of control structure
  - if
  - select case
  - for loop
  - foreach loop
  - do/while loops
  - goto
- error handling
- Subs and Functions

most of this section is going to be boilerplate explanation of these things

consider how to improve on that to avoid saying the same stuff as everyone else (or just power through it to get it on paper)

later on, there should be an advanced VBA section to handle:

- classes
- events
- adding references to other objects (specifically Office and Microsoft Runtime)

The pitfalls of running macros and how they destroy the undo functionality.

- using the debugger
- using the Immediate window

VBA 101

This section focuses on the basics of opening the Visual Basic Editor (VBE) and starting with VBA. This is called VBA 101 because these are the steps that you have to know before you can start programming. Hopefully these steps are already done or you can work through them quickly.

There are multiple ways to open the VBE, including:

- Use the keyboard shortcut ALT + F11
- Use the button on the Developer tab of the Ribbon (if enabled) (TODO: include steps for how to enable this menu if needed)
- Hit the Debug button if an existing code sample fails
- Hit the edit button on the run macro button, again off the developer tab (also available with `SHIFT`+`F11`)

Of these, the first two are more common.

Several of these methods require the Developer tab to be showing. If you are working with VBA, you should enable it. To enable it, follow steps:

1. XXX – you need to go through the settings to Customize the Ribbon. TODO: add some steps and images for the Developer tab on the Ribbon

Once you have the VBE open, you can add or edit code for your spreadsheets. The basics of the simple program are included below.

TODO: add the steps for a Hello World… how to create Sub and output the text

TODO: add pictures for these steps too

This section of the book is one of the few where the steps will be so clearly specified. I am going to this level of detail to ensure that you are able to get things started. This book will not include such detailed steps later for how to type and run code.

Introduction to VBA

This chapter will focus on the basics of VBA that are essential to using VBA to work with Excel. The upside of VBA is that it has a very simple instruction set. The downside of VBA is that it has a very simple instruction set. Fortunately, the vast majority of Excel/VBA interaction can be handled with very simple instructions. **The main difficulty with using Excel/VBA is not the VBA side of things, it's managing the object model for Excel**. This object model does not introduce new commands, but it does add a large number of interrelated objects, properties, and Functions that need to be known at some level to do anything. If you compare the length of this chapter to the length of the book, you will get a sense of what is meant by this.

An important thing to remember about VBA is that it exists outside of Excel, in some sense. VBA (Visual Basic for Applications) is derived from VB6 which is a legitimate programming language that (previously) was used for serious programming. These days (ca. 2019), no one starts a new project looking to use VB6; it doesn't offer the features of modern programming languages. That VBA exists outside of Excel means that

there are certain parts of the language that are independent of anything Excel has to offer. These aspects of VBA are the core parts of the language, and you need to understand them before you can do anything related to Excel.

Having said all of that, VBA consists of several key instructions:

- Declaring and setting variables
- Declaring and calling Subs and Functions
- Logic structures
- Loop structures
- Other control structures (Errors and Goto)

In addition to those aspects of using the language, there are a handful of details related to programming in general that are worth hitting:

- VBA 101, opening the VBE and getting started
- Adding references (how and why)
- Debugging code and using the tools provided

The flow of this chapter will hit on the VBA 101 question first. From there, we'll hit the language basics, and then touch on the 2 more advanced aspects of using VBA and Excel together.

Finally, it's worth noting that this basic overview misses a couple parts of VBA (TODO: like what) that might come up from time to time. They will be mentioned at the end of the chapter in passing, but this book is not a VBA reference. This book is designed to get you using VBA in sa professional setting with confidence. Knowing every nook and cranny of the language is not critical for that goal.

Declaring and Setting Variables

One of the core tasks when programming via VBA is working with variables. Variables are used to reference the Excel object model and to guide control structures. Within the Excel object model, the objects hold variables which point to other objects. Working with these objects is critical to using VBA. You will need to understand variables to do that.

This section is split into two areas: declaring variables and setting variables. The code for these two topics is simple. The complexity comes in planning out the best structure for managing variables. The variable declaration will directly shape how the control structures will work.

Declaring Variables

Declaring variables is straight forward. VBA offers a simple command to declare a new variable: `Dim`.

When declaring a variable, there are two components to it: variable name and variable type. Variable names are your choice with some constraints. You are not allowed to duplicate the name of an internal command, and you should go to some length to avoid using the same name as an Excel object model name. Beware that naming a variable has certain conventions, but these do not have any effect on the program execution. The main concern with names is that they will directly affect your ability to work with and maintain your code. Naming things is hard. Pick a strategy that works for you and your coworkers and get on it with it. There is no single answer here about how to name things.

The second part of the puzzle is to declare the type of the variable. This is THE core part of variables. When declaring a variable, you decide if the type should be the generic `Variant` or if you need a more specific type. There are times when you have to use Variant, but you should aim to use the most specific type that is possible. These types draw from VBA, from the Excel Object Model, or from your own created types. When thinking of variable types, there are two major groups of types:

- Value types = a number, string, or boolean
- Reference types = objects

TODO: find better place:

Note that you can technically use a variable before declaring it, but you should really avoid this practice. It leads to the potential to create all sorts of bugs later. Just don't do it. To better avoid this, setting the flag in the settings (TODO: add a picture of that).

TODO: add code sample for declaring a variable (show an object, primitive, and array)

Setting variables

Setting a variable is straight forward. The rule is: **for reference types, you must use `Set`; for value types, you must not.**

The real problem then is to determine whether or not you are working with a reference type. The rule is: if you are working with an object, it is a reference type. If you are working with a value (number, string, boolean), then you have a value type. Another approach, if you intend to use a `.` to call out some property of your variable, then it is a reference type. The exception here is arrays: they are set without using Set.

TODO: add code sample showing variable setting

Using Variables

It seems somewhat obvious that you would want to use a variable after declaring and setting it. This is generally always the case (why else would you create the variable). To that end, there are a pair of ways to use variables depending on whether it is a reference or value type. Value types are easier since you can only do 1 thing with them: use them in an expression. This feels and usually looks like mathematical formulas. The more complicated example comes with reference types where the variable stores a reference to another object. These variables have the ability to access either a property of the type or the default `Value` of the type. The distinctions between reference and value types can become confusing with the Excel Object Model since so many properties of objects reduce to value types. An example is the value of a `Range` which will hold some number or string or Error depending on what the cell contains.

When accessing a property of the object, you use the `.` to access a property by name. In this way, you can chain together a series of commands accessing the properties of objects. It is often the case that the property is itself another object which makes it possible to use another `.` to keep going. If you are using the VBE and properly declaring your variables, the VBE will work to provide helpful suggestions of what may be possible to use next (this is called Intellisense). The one pitfall to Intellisense is when the return from a given property can be Variant or a combination of possible results. When this happens, Intellisense will not offer any suggestions and you are left guessing whether or not the command exists. This is where it can be quite helpful to do one of two things:

TODO: create a demo of these bullets

- Create a new variable with the type that you know the object will have and Set that reference before using it. This “cheats” and tells Intellisense exactly what you expect to exist.
- Read through the documentation and gain an understanding of what types are possible and just use them. There is no rule that the type must be suggested by Intellisense for it to be valid.

In general, I take a combination of those two approaches often. If I expect to use the variable a number of times, I will go with the new variable route to avoid guessing properties later. If I only need the variable once or am copying code from somewhere else (and know it works), I will just go with the code as is without Intellisense. The one upside of creating new variables is that it forces you to be more explicit with your declarations. It also clearly shows your intent to other developers that may see your code later.

Value Default

I mentioned it above, but it is worth digging into the default `Value` property a little more. This can be a source of confusion because very often, you will accidentally use the name of a variable without calling for a property. In other programming languages, this will result in a compile time or runtime error. In VBA, your code will run and even worse will return something from the object that may not be what you want. When this happens, it can be incredibly difficult to track down the source of the error. To avoid this, you could never use the variable name as a shortcut to the `.Value` property. In practice this is a pain to manage and I will often mix and match whether or not Value is called. Sometimes, I am tired of typing out Value and just let the default work. Other times, I am being very diligent about calling everything explicitly to avoid some unforeseen error later. You will find that this comes down to your own preference and the preferences of others working on your code.

using Subs and Functions

The basic building blocks of your VBA efforts will be the Sub and the Function. It's possible that they are your only top level components if you do not use Class Modules. In all my years of using VBA, I've used Class modules only a couple of times, so they're not common.

Having said that, Subs and Functions are actually far more similar than different. The only real difference between the two is that Function can return a result back to the caller. A Sub on the other hand is meant to execute without returning anything back to the caller. It's possible to have a Sub manipulate a variable with can approximate returning a value for a little more work. If you're using a Function as a UDF (see chapter XXX, TODO: add link), then there are further limitations on what your Function can do. If you are not using it as a UDF, then there are no limitations that make a Sub distinct from a Function. The only difference is how you call them (if you want the return value) and that a Function is made to return something.

If you have a Function that does not actually return a value, it is the same as a Sub with the same code.

TODO: add an example of a Sub

TODO: add an example of a Function

declaring the parameters (Subs and Functions)

When creating a new Sub or Function you are able to determine the inputs to your new creation. There are a handful of ways of handling the inputs:

- Put the inputs into the parameters of the Sub/Function and allow the caller to provide them
- Use knowledge of the spreadsheet to determine the inputs (or prompt the user for an input)

The main split here is: do you require the person typing the VBA to give you the inputs? Or, do you use some other approach like asking the user or just pulling the inputs from the spreadsheet.

The most common approach is to pull the inputs out of the spreadsheet. This seems counter intuitive, but if you consider that the vast majority of VBA code is purposes written for a single use, then it stands to reason that code will not be built on a large number of Subs/Functions accepting parameters. The reason for this is that generally someone writes VBA to handle *their* spreadsheet and so the VBA just reflects that spreadsheet. This works great for individual cases but can become a burden when building larger workflows. The main thing to consider for lager workflows is that as the complexity grows, there will be a large amount og code that is called multiple times or could be called separately from the main workflow. When the sis the case, you are often served by pulling that code out into its own Sub/Function with parameters.

To create a Sub or Function with parameters, you simply add them to the definition line:

```
1 Sub WithSomeName(firstParameter as String)
2
3 End Sub
```

This approach is very simple. You give the parameter a name and a type declaration. This is very nice because it nearly exactly matches the `Dim` statement with a Sub. That correspondence makes it very easy to start with an internally declared variable and then upgrade it to parameter. You can also go the other way: take a parameter and inline it into the Sub with some default or determined value. This is less common.

Once the parameter has been given a name and a type, you can simply use it within the Sub like any other variable. In this regard, your code will look the exact same. IF you are the person typing the VBA to use this Sub, then you will have to provide an appropriate variable as the parameter to make it all work.

declaring an Optional parameter

The one additional thing to consider is that of `Optional` parameters. An optional parameter is one who is not strictly required. In lieu of a value, you can either leave the parameter missing or provide a default value. In either case, you can use the VBA specific function `IsMissing()` to determine if the parameter was entered. An Optional parameter can be a very nice feature when you are trying to determine whether

or not to make a Sub take parameters or just use defaults. You can provide the defaults in the parameter declaration and then allow the user (person typing the VBA) to override them if needed. This is a very common approach when writing library type code; provide sensible defaults that can be overwritten.

calling a Sub or Function

When you are calling a Sub or Function, there are a couple of ways to do it. The preferred approach is to simply type the name of the Sub/Function along with any required parameters. This will call the Sub. Another approach is to use `Call SubName` which is the same as `SubName`. This is an older approach that some people prefer. It can sometimes be the case that Sub names are not particularly clear in the VBA and using `Call` has the effect of making it obvious that code flow is being directed into a Sub.

WHen calling a Function, you have the same approaches available. You can just use the Function name or use call (TODO: is that right?).

ONe thing to be aware of with Functions is how to properly handle the return from the function (assuming it actually returns something). This is where VBA gets a bit weird. THe rules here split on whether the Function returns an Object or Value type.

For either type, you are required to call the Function with parentheses. This signals to VBA: please retain and use the return of this Function. For a reference type, you will need to use `Set` as required. For a value type, you will omit `Set`. See the code example below.

If you ever get the compile time error `Object reference not set` this means that you have not used a `Set` somewhere that is required. A good place to check are spots where you are using the return from a function. The same thing happens if you omit the parentheses. (TODO: is this right?)

```
Sub ExampleOfCallingCode()
    Dim rngReference as Range
    Set rngReference = someFunctionThatReturnsARange()

    Dim dblValue as Double
    dblValue = someFunctionThatReturnsADouble()

End Sub
```

declaring the return type (Function only)

For a Function, the only extra step is to declare the return type of the Function. This is done after the normal parameters, with an extra `as Type` where `Type` is the actual type that you want to return. Note that this type must be compatible with all possible Types that you could return. Sometimes this means that you need to return a Variant in order to have all possible return Types available to you. There are times where this makes sense (and a large part of the Excel object model does this), but note that using Variant will make it hard to use Intellisense to figure out what your VBA is capable of doing.

TODO: is this a Variant by default?

TODO: give some examples of Function returns (or link to examples of them)

returning from a Function

If you want to take advantage of a Function, you need to return a value from your Function. This returned value can then be consumed by the caller (or not). To return a value from a Function, you simply use the Function name as a variable and set its value appropriate. If the return type is an object or reference type, then you need to use Set to return the object. If it is a value type instead, you can simply set the return with an equal statement like any other value type. Once you have made the return statement, you can call Exit Function to break out of the Function.

For the caller, there are two things to keep in mind when using Functions. The first is that you must call the Function with parentheses in order to access the return value. The corollary of this is that if you call a Function with parentheses, you must use that return value to set the value of a variable. You will get an error if you do not do this correctly. Note that if you do not want the return value for some reason, you can avoid using parentheses in the same way you call a Sub. The second part is that you must call Set if the variable is an object/reference and not a value.

TODO: give an example of the return type and returning

logic structures

The logic structures are the backbone of nearly all VBA programs. There are a handful of times where you can just run through some commands with no branching logic, but in general, your program will need to make decisions based on some condition that it encounters. In order to make those decisions, you use the

logic structures of VBA. There is really only a single logic structure in VBA, the If-Then structure, but VBA provides a handful of useful additions to the basic If-Then to make programming a little easier.

The main logic structures then are:

- If-Then
- If-ElseIf-Then
- Select-Case

The If-Then is the main building block that allows you to do something if a condition is true or do something else otherwise. The If-ElseIf-Then allows you to add additional conditions to check before defaulting to the Else. If any of the ElseIf statements evaluate True, then the branch will stop traversing the conditions. The Select-Case is an extension of the If-ElseIf-Then that always compares a given variable against different possible values.

TODO: add example of the different forms of If-Then

For logic evaluation, there are always a handful of ways to arrive at the same result. You are allowed to evaluate multiple conditions in a single If-Then statement by using And and Or. You can also "nest" different logic blocks inside of each other to create the same sort of logic. In this way, you can either use If-ElseIf-Then or you can do an If-Then and stick a second If-Then in the Else clause. These will be equivalent. Sometimes one version looks better or makes more sense than the other.

helpful logic functions

TODO: add an overview of the functions And/Or (is there a XOR?) along with the logic operators <,>,<>, = etc.

the Select Case

The Select Case makes it possible to compare the value of a variable against multiple values without having to type the variable name every time. This is a purely syntactic feature that makes programming easier in certain cases. You can completely duplicate a Select-Case with an If-ElseIf-Then, but you may have to type more code.

TODO: add an example of a Select Case

Note that this section is fairly short. Logic structures are so prevalent in normal VBA code that should look for examples of these in the respective chapters instead of this section.

loop structures

The loop structures are an integral part of VBA programming. You are pretty much guaranteed to use them immediately. In some cases, you are more likely to use loops than logic structures. The reason that loops are so critical is that they allow you to perform an action multiple times or across multiple objects. Given the nature of a spreadsheet (where you have a high multitude of cells) and the reasons for using VBA (you want to perform some action multiple times) you really can't avoid loops. Gaining an understanding and comfort with loops is critical to your skill with VBA.

There are several types of loops that work similarly but have different use cases. Those include:

- For Each - useful when you have a collection and want to do something for each object in that collection (this is the most common loop to use since you will nearly always have a collection of Ranges or some other object to iterate through)
- For - useful when you want to do something a specific number of times
- Do/While - run a loop until a condition is met which is useful when you do not know in advance how many times to run the loop and you don't have a finite collection

It is worth noting that all loops can be written as a Do/While loop, but you will nearly never do this. There are good reasons that the For Each and For loops exist.

It is also worth mentioning here that I typically try to avoid For loops whenever possible. I always prefer to use a For Each loop if it is appropriate for the application. This is not an approach that I used from the beginning but have begun to value the use of For Each loops. If you are coming from a programming language that does not value collection iteration, then you might avoid the For Each loop at first. I'd strongly recommend you learn to use the For Each and appreciate it. Your code will be much cleaner and easier to read with For Each loops instead of the alternatives. Especially when dealing with Ranges, it is tempting to iterate through them as a nested For loop. You should really avoid this.

TODO: add an example of a bad For loop

For Each loop

It is not traditional to start with the For Each instead of the For loop, but I personally use the For Each far more so I'll start there.

The For Each loop is used whenever you have an utterable collection. An utterable collection can come from either the Excel object model or your own code. In general, most of the Excel object model returns an utterable collection. This is especially true for Ranges.

TODO: add a list of utterable collections that can be used here

You are not required to put the variable name in the Next line. I recommend not including the variable unless you have tons of code in the loop and are nesting loops. Typically you will rename the variable and then get a compile time error because the variable names don't match. I've never found the variable name in the Next line to help much.

TODO: add an example of a For Each loop

For loop

Another style of loop that exists is the "bare" `For` loop. This is one of the simplest loops to understand and control. The idea is simple: iterate through a chunk of code a given number of times. The most common forms of the `For` loop work through a fixed number of iterations. One example is easy: if you want to output the numbers 1 through 10 into a column of cells, you can easily use a For loop to output the number. This is a bad example though since it can easily be done with normal Excel functions, but it is quite common to do the equivalent sort of task when writing a larger macro. In that sense, it is easy to forget how versatile the For loop can be when needing to do something some number of times.

Compared to a While loop (discussed below) there are a number of advantages to the For loop:

- Much easier to control the "exit" strategy and avoid infinite loops
- Can be wired up with constants that are intuitive and just work
- Can be extended to use variables instead of constants to provide more flexibility

When moving beyond the simple 1 to 10 For loop, there are a handful of options which can be pushed into ever complex strategies:

- Use a variable for either the starting index, increment, or end point
- Use a negative step to go backwards through a list of numbers
- Use `Exit For` statements to control execution and kick out of the loop

On that last point is where you will see VBA is woefully underpowered compared to "modern" programming languages. VBA does not provide a simple command to **`continue`** a loop. There is a `Exit For` which can be used to kick out of the loop, but to **`continue`** you must create a `:LABEL` and use a `Goto LABEL` statement to jump there. There is nothing necessarily wrong with this, but it is an approach that is annoying and prone to some mistakes. The biggest issue is accidentally moving the label or having some other position issue. THe annoyance is having to create a label and use a `Goto`. There is nothing inherently wrong with a `Goto` but they provide create power which means awful bugs later.

It is worth noting that the `For Each` loop is a simplification of a `For` loop for a number of instances. In most cases, you could create an index and then iterate through an object by index, storing a reference to the object being stored. Behind the scenes, I believe this is how the majority of internal commands are handled. Despite the 1:1 translation between the two loops, it is typically MUCH simpler to use a `For Each` loop if you just need access to the underlying object in the collection. I will nearly always create an index outside of the loop and use it alongside a `For Each` instead of creating a `For` loop and storing a reference to an object. It always seems vastly simpler to store an Integer than an object. Aside from the marginal advantage of not calling `Set`, there is a immediate payoff of a `For Each` loop that if you name the variables correctly, you can typically read exactly what the code will do. This pays dividends for yourself and others later when reviewing the code.

It is also worth mentioning that there are a handful of instances where you are typically required to use a For loop even if you want to use the object being used. The standard example here is if you will eb modified the collection that you are iterating. In this case, you will rapidly create iteration issues trying to modify the collection inside the loop using it. In some of those cases, you will get runtime errors, but in others you will just get unintended consequences. The most common example of doing this is when you want to delete items from a collection while iterating through it. Let's say you need to check whether an item meets some criteria before deleting it. There are two ways to handle this:

- Use a For loop and run thought eh collection BACKWARDS. The direction is critical because it means that at worst, you are working at the end of the list which will not affect future operations.
- Use a "dual loop" approach.

An example of item 1 is shown below.

TODO: add example of backward loop

The dual loop approach is worth mentioning further since sometimes it can give you an elegant way out of a bind. The idea is that instead of modifying the collection while you iterate it, you store some amount of information outside of the collection and then use that to determine what to delete. This typically only works if the items being deleted exist independently of the collection that holds them. This happens often enough with Excel that it is worth giving a concrete example: deleting Rows from a Worksheet. To handle this dual loop approach, there are two possible options:

- Use one loop to create a collection that stores the rows to be deleted and then iterate that collection in a new loop
- Build a larger Range to delete as you go and then use an Excel function to handle the actual deletion

The latter option is only technically a "dual" loop. Technically Excel will use some sort of internal loop to

actually delete the Range. You are only required to cleverly create the Range which allows this internal process to be kicked off.

TODO: add an example of the Collection approach for deleting Ranges

TODO: add an example of the UNION-DELETE approach for deleting ranges.

TODO: add some examples of For loops and how they might be used

Do/While loop

The final style of loop is the Do/While loop. Although it is mentioned last, it is ultimately the simplest type of loop that exists. The idea is: run until some condition is meeting. This loop matches very nicely when your looping strategy involves some condition. You simply put than tocndifiton int he loop and let it work. The downside to a Do?while loop comes down to the possibility of an infinite loop. This leads to the common problem of a macro that hangs Excel and requires intervention to shut down. Infinite loops are technically easy to avoid, but it is far more common in practice to skip the steps that help avoid infinite loops.

It is worth mentioning at this point that all of the loop varieties can be recreated from the other loop varieties. From this standpoint, there are slight advantages to one style over another, but at the end of the day, you simply write the loop that works for the task at hand.

For a Do/While loop, there are two possible ways of writing it. You can either do a `Do...While` or a `While...Loop`. THe main difference is whether or not the loop will execute before the condition is checked. There are instances where one style makes more sense over the other. Typically you can always use the `While...Loop` variety, but you may be required to type an initialization statement before the loop that is repeated within the loop.

Some common examples where a While loop make sense include:

- Iterate down through a column of cells until some condition is met (typically a blank or non-blank cell). This is quite helpful when it is difficult to create the `Range` that might be used for a `For Each` loop.
- Iterate through the file system using the `Dir` command to find files to open and process

The WHile loop tends to make the most sense when you are not iterating through a fixed collection of objects because the `For Each` does a better job there. You also would avoid using it when you have a fixed number of iterations to run where a `For` loop makes a lot more sense. That then leaves the instances

where you want to loop through some action some number of times, but you're not sure how many times until you start going.

If you are particular adventurous, you can make use of the `Exit Do` command to exit out of the loop mid iteration. This pairs nicle with a `While True` at the start of the loop to ensure that nothing else will kick you out of the loop. There are instances where this can be a simple way to loop, but you have to be absolutely certain your `Exit Do` command will be triggered at some ppint or else you guarantee an infinite loop.

TODO: add an example of looping use `Dir`

TODO: add an example of a loop that works through a range using Offset

which loop and why

There are a handful of common reasons you might go for one loop instead of another. Worth knowing is that in general you can use any of the loops and get the same result. One approach is typically much easier to program, understand, and maintain.

A couple of good things to remember:

If you are going to modify a collection in the course of iterating through it, you should not use a For Each loop. The For Each does not update the utterable collection if you modify it during a loop. This is particularly important if you are looping through a collection to identify items to delete from the collection. You should never do this in a For Each loop. When deleting, you should typically use a For loop and iterate through the collection in reverse order. This makes it easy to handle deleting items since you cannot get out of order. You can achieve the same result with a Do/While, but I won't cover that.

TODO: add an example of deleting via a For loop

If you need an index/incrementing variable alongside your loop, you are not required to use a For loop. You can always create a new variable and increment it yourself inside the loop. This is sometimes preferable to switching to a For loop solely to get the counter/index variable.

If you are using a Do/While loop, you should give serious consideration to adding a counter and breaking the loop if the counter gets too large. It happens far too often where I use a While loop and end up freezing Excel because the loop never terminates. You can sometimes break the code and get Excel to respond, but that does not always work. This is especially important if you are generating code that others will use since they may be less familiar with how to break out of an infinite loop.

You may need to break out of a loop. Unfortunately, VBA does not have the normal Break and Continue commands that you might be familiar with from another language. The only way to break out of a loop on the spot is to add a label use a Goto command unless you are able to break out of the Function/Sub completely using Exit. This always feels dirty to me so instead I will typically structure the loop with a Boolean that can detect whether the next iteration should continue. This works for Continue, but it is not a good solution for getting a Break. The only way to do this is via a Goto. Just do it.

other control structures

With command

THe `With` command allows you to place a given variable within "scope" and avoid repeatedly typing that variable's name for each required call. The `With` command exists solely to reduce the number of times that a given object or variable name is typed. You are never required to ues a With command to accomplish a goal, but it can be helpful to clarify or avoid having too long of a code block. Having said that, a With block can be incredibly confusing to read especially when mixed with the always in scope function calls like `Range` or `Cells`. It is incredibly easy to avoid typing the required `.` to start a new line and accidentally refer to the globally scope object instead of your With scoped object. For this reason, I very rarely use the With command. When I do use it, I will typically only use it when I am working with a nested object that might be several levels deep. Having said that, I mostly avoid the With block by creating a variable which holds the object in question and using that instead. I have found that parsing a With block later can quickly become a confusing mess because of the difficulty of spotting the `.` which is critical.

If you read through some of the most common questions on the internet about "why my VBA no work?" you will quickly find issues with With blocks accidentally calling a globally scoped command. I have never asked those questions on the internet, but I have definitely been bitten by the same errors where a `.` is missed and the commang goes bonkers. It happens but is easily avoided by not using `With`.

GoTo statements

`GoTo` statements are used to force execution to jump to a specific Label regardless of anything else that the program is doing. A `GoTo` statement is required for error handling but is otherwise frowned upon by programmers with experience in other languages. The problem is that a bad `GoTo` statement allows you to do much damage within a program because you can quickly corrupt your program state by jumping around. Also, other programming languages tend to include all of the nice features that have replaced

places where `GoTo` was previously required. A good example of this is breaking out of a loop or skipping to the next item in a loop. The latter is typically handled with a **`continue`** statement in other languages. In VBA, this statement does not exist and you are required to use a `GoTo` if you want the functionality.

To make a GoTo statement work, you need to have a Label that the GoTo points to. An example looks like this:

```
Sub GoToExample()
    'doing some stuff

    If someConditiojn Then
        GoTo EndOfCode
    Else
        ' do some other stuff
    End if

EndOfCode:

End Sub
```

The rule for labels is that they are required to occur at the front of the lien (no indenting), they must be a single variable name without spaces, and they must end with a colon.

You should go to reasonable lengths to avoid using GoTo statements for anything other than error handling. They are the root of a lot of problems as execution order is concerned.

Error Handling

One final control structure that exists is related to error handling. It is an inevitable consequence that computer programs will eventually throw errors. There are a lot of techniques and good practice that can avoid errors, but sometimes you will be forced to deal with an error. The alternative to error handling is usually a pop up that informs the user that something went wrong. For an experienced user, they may be able to handle the `Debug` or `Continue` or `End` decision, but your typical user will assume that your code has failed catastrophically. It's entirely possible that the error has no effect on your intended outcome, or that the error could be resolved if the user just hit `Continue` but the take home message is that *if* something *has* to happen to respond to an error (or a possibly error), then you need error handling.

The elements of error handling are simple:

- Determine when to allow an error to be thrown
- Determine what happens with execution when an error occurs
- Determine where to go back to once the error state has been addressed

The first decision to make is whether or not to allow errors to interrupt execution. By default, the answer here is “yes”, an error will interrupt execution. If you want to handle this differently or reset it back to default, there are a pair of commands that can be used:

- `On Error Resume Next`, ignore all future errors, just keep trucking
- `On Error Goto 0`, stop execution immediately at the next error

If you are savvy about searching online for solutions to your problem, you will often see option 1 listed as the “go to” (or is it `GoTo`, ha!) solution for getting around an error. In the technical sense, yes, `On Error Resume Next` will absolutely get you around an error. It will by definition ignore the error and just keep going with execution. For the vast majority of workflows, this is an awful approach. Very often an error is indicating that something has gone awry from your expectations. If those expectations were reasonable, then it is very likely that future code will not work as intended. Therefore,e if you are getting an error, you should give serious consideration to finding the source of it before you `Resume Next` through it. Ignoring an error that should have been addressed nearly always causes more pain later.

THe other harsh approach to respond to an error is to force execution to stop immediately. This prompts the user with the popup about how to proceed. This prompt is helpful because it gives two options that may allow you to solve the problem. THe first is `Continue` which will attempt to run the line of code again that cause the issue. If the error still persists, then you will simply get it again. No harm. However, it is also possible to change the state of Excel while the prompt is visible. T his means that if your code was relying on an `ActiveChart` and you did not select one; you will be able to select a chart before hitting `Continue`. This can be a quick way out of a problem if you are confident where the error occurred. If you are programming only for yourself, this can also be a clean way around dealing with waiting for user input using another `GoTo` approach down below. Having said that, allowing a user to deal with an error prompt is absolutely awful in terms of usability.

The second way you can deal with these error prompts is by hitting `Debug`. This is likely the first response when an error occurs because you are very unlikely to know where the exact error occurs. Once you’ve seen it however, then you may be able to contue above. THe nice thing about debugging the error is that you get some powerful tools to try and solve the problem. For a full overview of debugging, check out the other section (TODO: add link). The specific features that are nice for dealing with error sinclude:

- Locals window, which will provide an overview of all the local variables and their current state

- Set next statement, which will allow you to skip over an error or rerun a line of code whose state may have changed between executions
- Immediate window, which will allow you to either run arbitrary commands or possibly output information about the program state.

All of those tools combined should make it possible for you to determine the source of an error. Once you have determine the source of an error, you can then set about resolving the error, again using the debug tools. Once you have solved the problem, you should give serious consideration to then adding that solution to the code using proper error handling techniques. Again, it is absolutely awful to present the user with an error dialog and expect them to be able to figure it out. Even if you are the user, you will absolutely tire of dealing with error prompts that can be handled with proper handling.

If you want to address an error, there are a couple of ways to handle that. They all rely on using the `On Error Goto LABEL` technique. This allows the code execution to jump to a specific place in your code. That area in your code is then able to do a couple of helpful things:

- Query the state of the `Err` object
- Attempt to address the error and then kick code back to the previous spot
- Provide the user with proper feedback before killing execution
- Log the issue accordingly before failing or prompting the user

With all of these approaches, the idea is simple: redirect execution to a known spot when the error has occurred. Once you are in a known spot, you can then step through possible problems and possible solutions. If you want, you are then able to send execution back to another spot to advance. IF you cannot resolve the error (or determine what caused it), you can then end execution all the same. Ideally you end execution with a better message than the normal prompt.

TODO: give an example of some error handling code

avoiding errors Although this section is about error handling, the best error handling is an approach hta tmake is very difficult for an error to occur in the first place. As you call into specific VBA and Excel Subs, you will gain a feel for which ones can cause problems. On the VBA side, there are a number of specific calls that will lead to errors:

- Indexing into an array with a index that is not valid: `Sheets("SomeSheetThatIsMissing")`
- Attempting to use a property on an object that does not exist
- Sending invalid parameters to a function

All of those items above have the nice poprerty that you may be able to provide checks for when you will enter an error state. The upside of this approach is that you can use an `If...Then` statement to check for an error causing state and then step around it. Before using `Range.Value`, you can check that `If Not Range Is Nothing`. `Nothing` is the default value for a reference type before it has been set to a poper reference. You are always going to get an error if you attempt to use a `Nothing`. You can avoid a ton of errors being thrown by simply checking for Nothing and avoiding its use when it appears.

For a lot of arrays and other utterable objects, you have different approaches for checking inf something is a valid index before accessing it. For a `Dictionary`, there is the `Exists` method. For `Worksheets` and other Excel arrays, you are always able to iterate through all of the items to check for existing before then using the index. TODO: add example of iterating sheets. It is very rare for the performances of VBA to be affected by these types of checks. There ar instances where it is not appropriate, but in general, these techniques work fine.

Application.XXX functions In some instances, it is possible to trade a runtime error for a return value that has a type of error. This occurs with the Application.XXX functions where XXX includes items in the list:

- Match
- TODO: any others?

This can be beneficial because when the function returns an error, you can then turn around and deal with it by checking `IsError`. If the function throws an error instead, you are forced to use proper error handling to catch the error and attempt to resume state.

common VBA errors TODO: add section about 1004

TODO: add information about compile time errors vs. run time errors.

common Excel errors In addition to the VBA errors, there are also a number of Excel specific errors that happen often enough that they should by addressed. Some of those common examples include:

- Using `ActiveXXX` without have `XXX` selected. This is most common with `ActiveChart` where it is possible to not have a Chart selected. This is not possibly with ActiveWorkbook or ActiveSheet since one will always be active. TODO: what about ActiveCEll?
- Using `Selection` when the “wrong” thing is selected. It is quite common to `Set` some vriable equal to Selection. If the wrong thing is selected, you will get an error about `Type Mismatch`

- Attempting to make a selection when it is not valid per the UI. This is most often the case when you attempt to Select a cell when its Parent Worksheet is not selected.
- Attempting to build a Range across Worksheets using `Union`
- Attempting to iterate through a Range of cells by checking `Range.Value` if the Range can contain errors. If this is possibl you will instead have to check for errors first.
- Attempting to access or change the `AutoFilter` if is has not been enabled first

There are also a ton of instances where some function returns `Nothing` and you do not check for it. T his most commonly occurs with:

- `Range.Find` where nothing was found
- `Intersect` where the two Ranges do not overlap
- TODO: add some others?

As a final note, it is owrht mentioning that the sign of a good programmer is one who has a feel for when errors can and cannot occur. You will begin to appreciate when it is needed to add error handling code versus when you know you will not need it. Too often as a beginner, you will be excluding error handling because you are unaware of what can go wrong. As you get better, you will start to exclude error handling because you actually know that no errors can occur. Until you get good, the result may look the same (no error handling code) but the result to the user is prompts and halted execution in one case.

where Excel and VBA meet

The previous sections focused on the aspects of VBA that exist independent on Excel. It is worth ending this chapter with a section that discusses the general theme of where VBA and Excel actually do intersect. The main thing to remember is that VBA provides the programming constructs and language, while Excel exposes an object model to VBA that can be programmed against.

A good rule of thumb is that anything you can do in Excel can be done via VBA. This is probably not an exaggeration. The Excel object model is incredibly detailed and provides access to every nook and cranny of an Excel spreadsheet. This gives you enormous power to manipulate a spreadsheet in whatever way you can imagine, but it also means that it is easy to be overwhelmed by sheer volume of commands and objects that exist. Fortunately, there are only a handful of common/useful objects to start with and within those objects there is a significant amount of overlap. For example, the Range and Chart both expose formatting related properties (e.g. Border colors) but the ways of editing those are the same on both objects.

TODO: is that true about the Borders being the same?

TODO: finish this section… not sure where it's going

adding references to external code

This section will cover how to add References to other files and programming components. There are 2 main reasons for why you would need to do this:

- You want to access some code from an Excel file that you or someone else created
- You want to access code from an existing component on your computer

The latter reason on the list is the more common reason for adding a Reference. There are a handful of common references that are added if you want additional components that are not available by default. Of these, the most common include the Microsoft Scripting Runtime and the references to other Office programs. For example, if you want to create a Dictionary, you will need to reference the Scripting Runtime. In general, there are a number of references that are nearly guaranteed to exist on all Windows computers. Having said that, there are also a handful of references that are commonly made where the required file may not be available. This uncertainty about the files available on a system is the major downside of using these references.

TODO: add a list with other common references and what they might include

For the first item, there are times where you have created some code that would be useful to use somewhere else but that you don't want to copy. This can be common for using helper code that you know is included in another file. The major drawback to this approach is that you are creating a permanent link between the file and the reference. This means that the one file will quit working if the reference ever moves or becomes unavailable. Despite this drawback, there are times where this can be convenient and the drawbacks less significant.

To add a reference is relatively simple. You simply go to Tools -> References. You can then check the boxes for any references that you would like to add. To add a reference to an existing Excel file, you will have to browse to the file and select it that way.

TODO: add some images of how to add a component

debugging your VBA code

One of the most useful features of VBA and the VBE is the ability to debug your code simply and in place. It is easy to take for granted the power of the VBE debugger, but it is worth mentioning that it is a solid debugger. The debugger has a handful of specific uses related to debugging your code:

- Stepping through execution and watching the movement of values into and out of variables

- Using the Immediate Window to execute arbitrary code or output the results of some value
- Setting the next instruction to force VBA to jump to an arbitrary point in your code
- Viewing the call stack to see how you reached a given spot
- Breaking at an arbitrary breakpoint or after an error was thrown

entering the debugger

To enter the debugger, you need to either set a breakpoint, hit Step Into, hit the Break key, or have an error thrown that prompts for debugging. By default, you will not be using the debugger while your code is running. This si actually a good thing since debugging code adds a large overhead which will kill performance. The most common approaches to entering the debugger are to set a breakpoint or via an error. This lines up with the idea that you either want to debug a specific point in your code or that you want to be able to see what went wrong when an error is thrown.

When setting a breakpoint, there are a handful of reasons for choosing where to set one:

- Right before an important step so that you can see the before and after state
- Inside of a control structure so that you can see whether or execution enters that structure. Sometimes there is information to be had when the code does *not* reach a breakpoint.

When breakpoints, you can technically disable them instead of removing them if you do not want them to trigger. I never use that feature.

If you are entering the debugger through an error, you simply hit `Debug` on the prompt. You will be starting on the line that threw the error ready to execute it again.

The other ways to enter the debugger are by hitting the CTRL+BREAK shortcut. If the VBA is at a stoppable point, this will cause an interrupt which gives the same prompt as the error prompt. From here, you can hit `Debug`.

The final approach is to use the Step Into button on the code to run. TODO: is this true?

stepping through code

Once you have entered the debugger, there are a handful of ways to affect execution. They are:

- Run
- Step Into
- Step Over

TODO: add a picture of the toolbar icons

TODO: explain how to reach these commands along with the shortcuts

Run will tell the debugger to just keep running until it hits another error or breakpoint. This is the same as normal execution.

Step Into and Step Over do the same thing with one difference. They both tell VBA to execute the current instruction and then resume debugging after it. The difference is how they handle whether or not to enter a `Sub` or `Function`. If you have a written a Sub or Function of your own and then call it, you have tow options while debugging. You can either enter that Sub and step through the commands in there. Or, you can treat that line with the Sub as a single step which can be processed as a single instruction. If you do that, you will `Step Over` all of the intermediate execution and resume debugging once code returns back to the level you started at. This is very important if you have a large number of nested Subs and Functions. The debugging steps allow you to decided how "deep" into the call stack you will go to pursue your deign. Sometimes, you will know that a given Sub works as intended and you do not want to step into it. Other times, you will reach a Sub being called and want to know exactly how it arrived at its output.

If you want to step through to a specific spot but cannot get there easily with the commands above, you can always just set a new breakpoint right there and hit `Run`. This will run until that line. You can also right click on a line nad do `Run until this point` and you will get the same effect. TODO: is that right?

viewing the state of your code

The whole point of debugging is generally to view the state of your code (or the Excel side of things) in process. The idea of viewing the state means a couple of concrete things:

- What are the values of specific variables?
- What was the order of execution? Which control structures were processed and in what way?
- What happens if I do "this" instead of "that"?

Each of those is hit below:

values of variables Typically, the most important aspect of debugging is seeing which variables hold which values. The idea is that if you can see what the variables hold at runtime, you can check that against your expectations and then gain insight into why your program is behaving the way it does. Other times, you want to see the values of things so that you can decide how to proceed from your current ppint. VBA provides a number of ways to check the value of a variable:

- Hover over the variable and allow the VBE to see you the value
- Using the Locals window
- Using the Immediate window with `?` added to the start (TODO: is that the same as Debug.pRint?)
- Using the Watch window after creating a watch
- Running a command where you put the value into the spreadsheet

The VBE is fairly helpful when debugging compared to other debuggers. It does about what you would expect. This means that you will get tooltips when you hover over variables. This works well for variables that hold a value and not an object. For an object, if you hover, you will get the `.Value` property of the object and not a drop down to explore. IN this regard, the debugger is inferior to a modern Visual Studio instance.

If you want to explore the properties of an object, or see a persistent value without hovering, you can use the Loacls or Watch window. They do the same thing: show the values of variables while also allowing you to click down into Objects and their properties. The Locals window works by giving you a list of all the local variables automatically. T eh Watch window works by requiring you to provide the variable name or command that you want to watch. I always start with the Locals window since typically local variable are what I want to see.

When reviewing the contents of an object, beware that VBA will not show you all of the properties of the object. In particular, it will not show you properties that are the result of a Function instead of a normal property. For a lot of Excel Object Model objects this is a key point. There are a large number of properties that you will need to add to the Watch window or query directly with the Immediate window to see their value. A common example: `Range.Address`.

TODO: add an example of using the Watch window

TO use the Immediate window, you first need to enable it via View (TODO: add this for others). Once enabled, you can use the Immediate window as a place to execute whatever code you want. It works by executing single lines at a time. IF you want the output of a command, use `?` at the start to print the result. You can use the Immediate window whevnerm, including during normal development (i.e. even when code is not running).

TODO: add an example of using the Immediate window

One particular thing that can be done (although not often) is that you can use the spreadsheet as a place to dump the results of your debugging. Sometimes, you will need to inspect some object and find that the VBE is just not than helpful. Maybe you have an array whose values you want to hceck. The simple approach here is to dump that array to the spreadsheet using the Immediate window (or actual code) and

then set a breakpoint to inspect it. This gives a nice back and forth between Excel and VBA that simply does not exist in other programming environments. Once you see Excel as a huge playgournd to dump arrays, you will find all sorts of using for that while programming.

forcing execution

In addition to watching the execution of a program, you also have the ability to change the execution. T his si done by using the `Set next command` TODO: name? while running. This is the "nuclear" option of debugging because it does exactly what it says. It will tell VBA to execute *whatever* line you want next. This allows you to completely ruin your execution while also providing you the power to step to a given spot. It's always the case when writing code that you end up on the wrong side of an If/Else while developing a loop. Sometimes, you just want to see what happens if you go down the other branch. T his option allows you to test that alternative execution path without having to modify your code. You just tell the debugger to execute that branch next and things will work. Very often however, you can using this feature to accidecanlly skip over code where variables are declared or Set and then you will have all sorts of errors because objects are set to `Nothing` instead of the values that are required.

Despite the pitfalls of moving execution arbitrarily, most people who know this feature exists are capable of using it appropriately. They typically are not surprised when things break.

viewing the call stack

One final feature which is useful is to check the Call Stack. The Call Stack is a list of all the proceduring Subs or Functions that are "active" preceding the current command. It gives you a list of all the places that came before your current line of code. The Call Stack is invaluable when you have started debugging following an error because oftentimes you will not know how you reached a given spot. This is especially true if you are debugging code that is used in multiple places.

To see the Call Stack do View->Call Stack. You can then double click on an item and jump back to that spot. Note that the VBE will attempt to show you the vales of variables at that location which can be very helpful.

The Call Stack can be very helpful if you are using recursive code that calls itself. This code can be very hard to debug because oftentimes a breakpoint will trigger more than you want. IF you are waiting for an error on the 8th time through a Function, then you don't want to skip the breakpoint 7 times. Instead, you can wait for the error, then use the Call Stack to step back through the previous iterations and see what happened.

TODO: add a picture of the call stack