The PowerShell Scripting & Toolmaking Book

Forever Edition

by Don Jones and Jeffery Hicks

The PowerShell Scripting and Toolmaking Book

Forever Edition

Don Jones and Jeff Hicks

This book is for sale at http://leanpub.com/powershell-scripting-toolmaking

This version was published on 2022-01-29

Tweet This Book!

Please help Don Jones and Jeff Hicks by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I got The #PowerShell #Toolmaking book http://leanpub.com/powershell-scripting-toolmaking @JeffHicks & @concentrateddon

The suggested hashtag for this book is #PowerShellToolmaking.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#PowerShellToolmaking

Also By These Authors

Books by Don Jones

Become Hardcore Extreme Black Belt PowerShell Ninja Rockstar

Instructional Design for Mortals

How to Find a Wolf in Siberia

Tales of the Icelandic Troll

The Culture of Learning

Alabaster

Power Wave

Onyx

Superior Wave

Shell of an Idea

Verdant

Books by Jeff Hicks

The PowerShell Practice Primer

The PowerShell Conference Book

#PS7Now

Contents

About This Book

The 'Forever Edition' of this book is published on LeanPub[1], an "Agile" online publishing platform. That means the book is published as we write it, and *that* means we'll be able to revise it as needed in the future. We also appreciate your patience with any typographical errors, and we appreciate you pointing them out to us - in order to keep the book as "agile" as possible, we're forgoing a traditional copyedit. Our hope is that you'll appreciate getting the technical content quickly, and won't mind helping us catch any errors we may have made. You paid a bit more for the book than a traditional one, but that up-front price means you can come back whenever you like and download the latest version. We plan to expand and improve the book pretty much forever, so it's hopefully the last one you'll need to buy on this topic!

You may also find this book offered on traditional booksellers like Amazon. In those cases, the book is sold as a specific edition, such as "Second Edition." These represent a point-in-time snapshot of the book, and are offered at a lower price than the Agile-published version. **These traditionally published editions do not include future updates**.

If you purchased this book, thank you. Know that writing a book like this takes hundreds of hours of effort, during which we're not making any other income. Your purchase price is important to keeping a roof over our families' heads and food on our tables. Please treat your copy of the book as your own personal copy - it isn't to be uploaded anywhere, and you aren't meant to give copies to other people. We've made sure to provide a DRM-free file (excepting any DRM added by a bookseller other than LeanPub) so that you can use your copy any way that's convenient for you. We appreciate your respecting our rights and not making unauthorized copies of this work.

If you got this book for free from someplace, know that you are making it difficult for us to write books. When we can't make even a small amount of money from our books, we're encouraged to stop writing them. If you find this book useful, we would greatly appreciate you purchasing a copy from LeanPub.com or another bookseller. When you do, you'll be letting us know that books like this are useful to you, and that you want people like us to continue creating them.

Please note that this book is not authorized for classroom use unless a unique copy has been purchased for each student. No-one is authorized or licensed to manually reproduce the PDF version of this book for use in any kind of class or training environment.

---

[1] https://leanpub.com

Dedication

This book is fondly dedicated to the many hardworking PowerShell users who have, for more than a decade, invite us into their lives through our books, conference appearances, instructional videos, live classes, and more. We're always humbled and honored by your support and kindness, and you inspire us to always try harder, and to do more. Thank you.

Acknowledgements

Thanks to Michael Bender, who has selflessly provided a technical review of the book. Any remaining errors are, of course, still the authors' fault, but Michael has been tireless in helping us catch many of them.

About the Authors

Don Jones received the Microsoft MVP Award recipient for 16 consecutive years for his work with Windows PowerShell and administrative automation. He has authored dozens of books, articles, white papers, and instructional videos on information technology, and today is a Vice President in the Content team at Pluralsight.com. Don was also a co-founder of The DevOps Collective[2], which offers IT education programs, scholarships, and which runs PowerShell.org and PowerShell + DevOps Global Summit[3] and other DevOps- and automation-related events.

Don's recent writing focuses on business, instructional design, self-improvement, and fiction, and can be found at http://leanpub.com/u/donjones[4]. You can follow Don on Twitter @concentratedDon. He blogs at DonJones.com.

Jeff Hicks is a grizzled IT veteran with almost 30 years of experience, much of it spent as an IT infrastructure consultant specializing in Microsoft server technologies with an emphasis in automation and efficiency. He is a multi-year recipient of the Microsoft MVP Award. He works today as an independent author, teacher and consultant. Jeff has taught and presented on PowerShell and the benefits of automation to IT Pros worldwide for over a decade. Jeff has authored and co-authored a number of books, writes for numerous online sites, is a Pluralsight author, and a frequent speaker at technology conferences and user groups world-wide.

You can keep up with Jeff on Twitter (@JeffHicks) and on his blog at https://jdhitsolutions.com[5].

Additional Credits

Technical editing has been helpfully provided not only by our readers, but by Michael Bender. We're grateful to Michael for not only catching a lot of big and little problems, but for fixing most of them for us. Michael rocks, and you should watch his Pluralsight videos. However, anything Michael didn't catch is still firmly the authors' responsibility.

[2] https://devopscollective.org
[3] https://events.devopscollective.org/
[4] http://leanpub.com/u/donjones_
[5] https://jdhitsolutions.com

Foreword

After the success of *Learn PowerShell in a Month of Lunches*, Jeff and I wanted to write a book that took people down the next step, into actual scripting. The result, of course, was *Learn PowerShell Toolmaking in a Month of Lunches*. In the intervening years, as PowerShell gained more traction and greater adoption, we realized that there was a lot more of the story that we wanted to tell. We wanted to get into help authoring, unit testing, and more. We wanted to cover working with different data sources, coding in Visual Studio, and so on. These were really out of scope for the *Month of Lunches* series' format. And even in the "main" narrative of building a proper tool, we wanted to go into more depth. So while the *Month of Lunches* book was still a valuable tutorial in our minds, we wanted something with more tooth.

At the same time, this stuff is changing *really* fast these days. Fast enough that a traditional publishing process - which can add as much as four months to a book's publication - just can't keep up. Not only are we kind of constantly tweaking our narrative approach to explaining these topics, but the topics themselves are constantly evolving, thanks in part to an incredibly robust community building add-ons like Pester, Platyps, and more.

So after some long, hard thinking, we decided to launch this effort. As an Agile-published book on LeanPub, we can continuously add new content, update old content, fix the mistakes you point out to us, and so on. We can then take major milestones and publish them as "snapshots" on places like Amazon, increasing the availability of this material. We hope you find the project as exciting and dynamic as we do, and we hope you're generous with your suggestions - which may be sent to us via the author contact form from this book's page on LeanPub.com. We'll continue to use traditional paper publishing, but through a self-publishing outlet that doesn't impose as much process overhead on getting the book in print. These hard copy editions will be a "snapshot" or "milestone edition" of the electronic version.

It's important to know that we still think traditional books have their place. *PowerShell Scripting in a Month of Lunches*, the successor to *Learn PowerShell Toolmaking in a Month of Lunches*, covers the kind of long-shelf-life narrative that is great for traditionally published books. It's an entry-level story about the right way to create PowerShell tools, and it's very much the predecessor to this book. If *Month of Lunches* is about getting your feet under you and putting them on the right path, this book is about refining your approach and going a good bit further on your journey.

Toolmaking, for us, is where PowerShell has always been headed. It's the foundation of a well-designed automation infrastructure, of a properly built DSC model, and of pretty much everything else you might do with PowerShell. Toolmaking is understanding what PowerShell is, how PowerShell *wants* to work, and how the world engages with PowerShell. Toolmaking is a big responsibility.

My first job out of high school was as an apprentice for the US Navy. In our first six weeks, we rotated through various shops - electronics, mechanical, and so on - to find a trade that we thought

we'd want to apprentice for. For a couple of weeks, I was in a machine shop. Imagine a big, not-climate-controlled warehouse full of giant machines, each carving away at a piece of metal. There's lubrication and metal chips flying everywhere, and you wash shavings out of yourself every evening when you go home. It was disgusting, and I hated it. It was also boring - you set a block of metal into the machine, which might take hours to get precisely set up, and then you just sat back and kind of watched it happen. Ugh. Needless to say, I went into the aircraft mechanic trade instead. Anyway, in the machine shop, all the drill bits and stuff in the machine were called *tools* and *dies*. Back in the corner of the shop, in an enclosed, climate-controlled room, sat a small number of nicely-dressed guys in front of computers. They were using CAD software to *design new tools and dies* for specific machining purposes. These were the *tool makers*, and I vowed that if I was ever going to be in this hell of a workplace, I wanted to be a *toolmaker* and not a *tool user*. And that's really the genesis of this book's title. All of us - including the organizations we work for - will have happier, healthier, more comfortable lives as high-end, air-conditioned *toolmakers* rather than the sweaty, soaked, shavings-filled tool users out on the shop floor.

Enjoy!

Don Jones

Feedback

We'd love your feedback. Found a typo? Discovered a code bug? Have a content suggestion? Wish we'd answered a particular question? Let us know.

Zeroth, make sure you're not using Leanpub's online reader, as it omits some of the front matter from the book. Use the PDF, EPUB, or MOBI version, or a printed edition you bought someplace else.

First, please have a chapter name, heading reference, and a brief snippet of text for us to refer to. We can't easily use page numbers, because our source documents don't have any.

Second, understand that due to time constraints like having full-time jobs, we can't personally answer technical questions and so forth. If you have a question, please hop on the forums at PowerShell.org[6], where we and a big community of enthusiasts will do our best to help.

Third, keep in mind that the EPUB and MOBI formats in particular allow little control over things like code formatting. So we can't usually address those for you.

Then, head to the LeanPub website and use their email link[7] to email us. We can't always reply personally to every email, but know that we're doing our best to incorporate feedback into the book.

Finally, accept our thanks!

[6] http://powershell.org
[7] https://leanpub.com/powershell-scripting-toolmaking/email_author/new

Introduction

Pre-Requisites

We're assuming that you've already finished reading an entry-level tutorial like, *Learn Windows PowerShell in a Month of Lunches*, or that you've got some solid PowerShell experience already under your belt. Specifically, nothing on this list should scare you:

- Find commands and learn to use them by reading help
- Write very basic "batch file" style scripts
- Use multiple commands together in the pipeline
- Query WMI/CIM classes
- Connect to remote computers by using Remoting
- Manipulate command output to format it, export it, or convert it, using PowerShell commands to perform those tasks

If you've already done things like written functions in PowerShell, that's marvelous - but, you may need to be open to un-learning some things. Some of PowerShell's best practices and patterns aren't immediately obvious, and especially if you know how to code in another language, it's easy to go down a bad path in PowerShell. We're going to teach you the right way to do things, but you need to be willing to re-do some of your past work if you've been following the Wrong Ways.

We also assume that you've read *PowerShell Scripting in a Month of Lunches*, a book we wrote for Manning. It provides the core narrative of "the right way to write PowerShell functions and tools," this book essentially picks up where that one leaves off. Look for that book in late 2017 from Manning or your favorite bookseller. Part 1 of this book *briefly* slams through this "the right way" narrative just to make sure you've got it in your mind, but the *Month of Lunches* title really digs into those ideas in detail.

Versioning

This book is primarily written against Windows PowerShell v5/v5.1 running on Microsoft Windows. In January 2018, Microsoft announced the General Availability of PowerShell Core 6.0, which is a distinct cross-platform "branch" of PowerShell. This branch has now become PowerShell 7, which was released in early 2020. As far as we can tell, everything we teach in this book applies to PowerShell 7, too - although some of our specific examples may still only work on Windows PowerShell, the concepts and techniques are applicable to PowerShell 7. However, PowerShell 7 includes some new scripting features which we'll cover in a dedicated chapter or two.

The Journey

This book is laid out into seven parts:

1. A quick overview of "the right way" to write functions.
2. Professional-grade toolmaking, where you amp up your skills, comes next in a second narrative. This part is less tightly coupled than the first, so you can just read what you think you need, but we still recommend reading the chapters in order.
3. Moving on from toolmaking for a moment, we'll cover different kinds of controller scripts that can put your tools to use. Read these in whatever order you like.
4. Data sources are often a frustrating point in PowerShell, and so this part is dedicated to those. Again, read whichever ones you think you need.
5. More advanced topics complete the book, and again you can just read these as you encounter a need for them.
6. A high-level introduction to using Pester in your toolmaking development.
7. Scripting for the PowerShell 7 and cross-platform world.

Following Along

We've taken some pains to provide review Q&A at the end of most chapters, and to provide lab exercises (and example answers) at the end of many chapters. We strongly, strongly encourage you to follow along and complete those exercises - *doing* is a lot more effective than just *reading*. And if you get stuck, hop onto the Q&A forums on PowerShell.org and we'll try and unstick you. We've tried to design the labs so that they only need a Windows client computer - so you won't need a complex, multi-machine lab setup. Of course, if you *have* more than one computer to play with, some of the labs can be more interesting since you can write tools that query multiple computers and so forth. But the code's the same even if you're just on a single Windows client, so you'll be learning what you need to learn.

Providing Feedback

Finally, we hope that you'll feel encouraged to give us feedback on this book. There's a "Contact the Authors" form on this book's page[8] on LeanPub.com, and you're also welcome to contact us on Twitter @concentratedDon and @JeffHicks. You can also post in the Q&A forums on PowerShell.org, which frankly is a lot easier to respond to than Twitter. If you purchased the "Forever Edition" of this book on LeanPub, then you'll see us incorporating suggestions and releasing a new build of the book all the time. If you obtained the book elsewhere, we can't turn your purchase into a LeanPub account for you. However, when the book changes enough for us to publish a new "edition" to other booksellers, that might be a time to pick it up on LeanPub instead, provided you understand the "Agile publishing" model and are comfortable with it.

[8] http://leanpub.com/powershell-scripting-toolmaking

A Note on Code Listings

The code formatting in this book only allows for about 60-odd characters per line. We've tried our best to keep our code within that limit, although sometimes you may see some awkward formatting as a result.

For example:

```
Invoke-CimMethod -ComputerName $computer-MethodName Change -Query "SELECT * FROM Win\
32_Service WHERE Name = '$ServiceName'"
```

Here, you can see the default action for a too-long line - it gets word-wrapped, and a backslash inserted at the wrap point to let you know. We try to avoid those situations, but they may sometimes be unavoidable. When we *do* avoid them, it may be with awkward formatting, such as using backticks (`):

```
Invoke-CimMethod -ComputerName $computer `
                      -MethodName Change `
       -Query "SELECT * FROM Win32_Service WHERE Name = '$ServiceName'" `
```

Here, we've given up on neatly aligning everything to prevent a wrap situation. Ugly, but oh well.

You may also see this crop up in `inline code` snippets, especially the backslash.

If you are reading this book on a Kindle, tablet or other e-reader, then we hope you'll understand that all code formatting bets are off the table. There's no telling what the formatting will look like due to how each reader might format the page. We trust you know enough PowerShell to not get distracted by odd line breaks or whatever.

When *you* write PowerShell code, you should not be limited by these constraints. There's no reason for you to have to use a backtick to "break" a command. Simply type out your command. If you want to break a long line to make it easier to read without a lot of horizontal scrolling, you can hit `Enter` after any of these characters:

- Open parenthesis (
- Open curly brace {
- Pipe |
- Comma ,
- Semicolon ;

- Equal sign =

This is probably not a complete list, but breaking after any of these characters makes the most sense.

Anyway, we apologize for these artifacts. Keep in mind that you can, and should, use `Install-Module PowerShell-Toolmaking` to download and install the code samples from the PowerShell Gallery. They'll end up in `\Program Files\WindowsPowerShell\Modules\PowerShell-Toolmaking`, typically broken down by chapter. We try to update that download often, so if you don't have the latest version installed, do that.

Lab Setup

We hope that you plan to follow along with us in this book, and to help you do so we've provided hands-on exercises at the end of most chapters. To complete those, you won't need much of a lab environment - just a Windows 10 (or later) computer to which you have Administrator access, and which has Internet connectivity. Business editions (not "Home") of Windows are recommended. We've built the labs so that there's no need for a domain controller, servers, or anything else.

We assume that your lab computer (or virtual machine) will have Internet access.

Create a Virtualized Environment

We know that most of our readers are technically savvy and more than likely have a preferred virtualization tool. You might prefer Hyper-V or big a VMware fan. Or maybe something completely different. If you would like to have a separate Windows 10 environment, we suggest downloading the evaluation edition of Windows 10 from https://www.microsoft.com/en-us/evalcenter/evaluate-windows-10-enterprise[9]. You should be able to use your virtualization tool of choice to create a new virtual machine from the ISO file.

Use the Windows 10 Sandbox

Another option if you are running a recent Pro or Enterprise version of Windows 10 is to enable the Windows Sandbox feature. You need a machine that supports virtualization and is enabled in the BIOS. The sandbox is a completely separate Windows 10 instance that you can play in without disturbing your main desktop. Although be aware that it is not persistent like a virtual machine. You get a fresh experience every time you launch it.

To install, open up Control Panel and go to Add/Remove Windows Features to enable the feature. Or run `Enable-WindowsOptionalFeature -FeatureName Containers-DisposableClientVM`. You'll most likely need to reboot.

Adding Lab Files and Configuring PowerShell

We have published all of the lab files for this book on the PowerShell Gallery, to make it easier to install them.

[9] https://www.microsoft.com/en-us/evalcenter/evaluate-windows-10-enterprise

1. On your Windows computer, press `Windows+R`, type *powershell*, and press `Enter`.
2. Right-click the PowerShell icon on the Task Bar, and select `Run as Administrator`.
3. Click `Yes`.
4. In the new PowerShell window (which must say "Administrator: Windows PowerShell" in the title bar), type `Install-Module PowerShell-Toolmaking` and press `Enter`.
5. You may be notified that "NuGet provider is required to continue." Type `Y` and press `Enter`.
6. You may be notified of an "Untrusted repository." Type `Y` and press `Enter`.
7. Type `Set-ExecutionPolicy Bypass` and press `Enter`.
8. Type `Y` and press `Enter`.

If the installation fails, or if you see an error or warning when setting the execution policy, ensure that PowerShell is running as Administrator and that the computer has unrestricted Internet access. On a company-owned computer, restrictions may be in place that prevent the installation of files or the changing of the execution policy. You will need to consult your company's IT administrators to remedy that.

If you are new to scripting, you may also want to install the free Visual Studio Code, often referred to as VS Code, from https://code.visualstudio.com/[10]. The PowerShell ISE that ships with Windows 10 isn't going away, and we used it a lot to develop the material in this book. But it should be considered deprecated. Microsoft is no longer developing it. All of their efforts are directed toward VS Code. This product is updated frequently and has a rich ecosystem of extensions. In fact, after you install it and launch it, click the gear icon in the lower left and then Extensions. Search for *powershell* and install the extension.

We won't lie. VS Code has a learning curve. But if you are new to scripting, you most likely would have to learn the PowerShell ISE. If you have to spend time learning a tool you might as well learn the tool that will be around for a while and as a bonus works cross-platform!

Assumptions Going Forward

Because scripting and toolmaking are not entry-level tasks, we assume that readers are already aware of the need to run PowerShell "as Administrator" when developing scripts and tools. We assume a basic level of familiarity with the PowerShell Integrated Scripting Environment (ISE), and we assume an intermediate or higher level of familiarity with PowerShell itself. If you don't feel you meet these expectations, we suggest first completing *Learn Windows PowerShell in a Month of Lunches*[11] available from most booksellers or from Manning.com.

[10] https://code.visualstudio.com
[11] https://www.manning.com/books/learn-windows-powershell-in-a-month-of-lunches-third-edition

Part 1: Review PowerShell Toolmaking

This first Part of the book is essentially a light-speed refresher of what *PowerShell Scripting in a Month of Lunches* covers. If you've read that book, or feel you have equivalent experience, then this short Part will help refresh you on some core terminology and techniques. If you haven't… well, we really recommend you get that fundamental information under your belt first.

Functions, the Right Way

This chapter is essentially meant to be a warp-speed review of the material we presented in the core narrative of *Learn PowerShell Toolmaking in a Month of Lunches* (and its successor, *PowerShell Scripting in a Month of Lunches*). This material is, for us, "fundamental" in nature, meaning it remains essentially unchanged from version to version of PowerShell. Consider this chapter a kind of "pre-req check;" if you can blast through this, nodding all the while and going, "yup," then you're good to skip to the next Part this book. If you run across something where you're like, "wait, what?" then a review of those foundational, prerequisite books might be in order, along with a thorough reading of this Part of this book.

By the way, you'll notice that our downloadable code samples for this book (the "PSToolmaking" module in PowerShell Gallery) contain the same code samples as the core "Part 2" narrative from *PowerShell Scripting in a Month of Lunches*. Those code samples also align to this book, and we use them in this chapter as illustrations.

Tool Design

We strongly advocate that you always begin building your tools by first *designing* them. What inputs will they require? What logical decisions will they have to make? What information will they output? What inputs might they consume from other tools, and how might their output be consumed? We try to answer all of these questions - often in writing! - up front. Doing so helps us think through the ways in which our tool will be used, by different people at different times, and to make good decisions about how to build the tool when it comes time to code.

Start with a Command

Once we know what the tool's going to do, we begin a console-based (never in a script editor) process of discovery and prototyping. Or, in plain English, we figure out the commands we're going to need to run, figure out how to run them correctly, and figure out what they produce and how we're going to consume it. This isn't a lightweight step - it can often be time-consuming, and it's where all of your experimentation can occur.

A user in PowerShell.org's forums once posted a request for help with the following:

I need a PowerShell script that will check a complete DFS Root, and report all targets and access based enumeration for each. I then need the script to check all NTFS permissions on all the targets and list the security groups assigned. I then need this script to search 4 domains and report on the users in these groups.

And yup - that's what "Start with a Command" means. We'd probably start by planning that out - inputs are clearly some kind of DFS root name or server name, and an output path for the reports to be written. Then the discovery process would begin: how can PowerShell connect to a DFS root? How can it enumerate targets? How can it resolve the target physical location and query NTFS permissions? Good ol' Google, and past experience, would be our main tool here, and we wouldn't go an inch further until we had a text file full of answers, sample commands, and notes.

Build a Basic Function and Module

With all the functional bits in hand, we begin building tools. We almost always start with a basic function (no `[CmdletBinding()]` attribute) located in a script module. Why a script module? It's the end goal for us, and it's easier to test. We'd fill in our parameters, and start adding the functional bits to the function itself. We tend to add things in stages. So, taking that DFS example, we'd first write a function that simply connected to a DFS root and spewed out its targets. Once that was working, we'd add the bit for enumerating the targets' physical locations. Then we'd add permission querying… and so on, and so on, until we were done. None of that along-the-way output would be pretty - it'd just be verifying that our code was working.

Adding CmdletBinding and Parameterizing

We'd then professional-ize the function, adding `[CmdletBinding()]` to enable the common parameters. If we'd hard coded any changeable values (we do that sometimes, during development), we'd move those into the `Param()` block. We'd also dress up our parameters, specifying data [types], mandatory-ness, pipeline input, validation attributes, and so on. We'd obviously re-test.

Emitting Objects as Output

Next, we work on cleaning up our output. We remove any "development" output created by `Write-Output` or `Write-Host` (yeah, it happens when you're hacking away). Our function's only output would be an object, and in the DFS example it'd probably include stuff like the DFS root name, target, physical location, and a "child object" with permissions.

If you're really reading that DFS example, we'd probably stop our function at the point where it gets the permissions on the DFS targets. The results of that operation could be used to unwind the users who were in the resulting groups - a procedure we'd write as a separate tool, in all likelihood.

Using Verbose, Warning, and Informational Output

If we hadn't already done so, we'd take the time to add `Write-Verbose` calls to our function so that we could track its progress. *We* tend to do that habitually as we write, almost in lieu of comments a lot of the time, but *we* have built that up as a habit. We'd add warning output as needed, and potentially add `Write-Information` calls if we wanted to create structured, queryable "sidebar" output.

Comment-Based Help

We'd definitely "dress up" our code using comment-based help, if not full help (we cover that later in the book). We'd make sure to provide usage examples, documentation for each parameter, and a pretty detailed description about what the tool did.

Handling Errors

Finally, and again if we hadn't habitually done so already, we'd anticipate errors and try to handle them gracefully. "Permission Denied" querying permissions on a file? Handled - perhaps outputting an object, for that file, indicating the error.

Are You Ready

That's our process. The entire way through, we make sure we're conforming as much as possible to PowerShell standards. Input via parameters only; output only to the pipeline, and only as objects. Standardized naming, including Verb-Noun naming for the function, and parameter names that reflect existing patterns in native PowerShell commands. We try to get our command to look and feel as much like a "real" PowerShell command as possible, and we do that by carefully observing what "real" PowerShell commands do.

Ok, if you've gotten this far and you're still thinking, "yup, got all that and good to go," then you're… well, you're good to go. Proceed.

PowerShell Tool Design

Before you sit down and start whipping up a PowerShell function or class-based module, you need to seriously think about its *design*. We almost constantly see toolmaking newcomers start charging into their code, and before long they've made some monstrosity that is harder to work with than it should be. Or they are totally lost and don't know what to do next. In this chapter, we're going to lay out some core PowerShell tool design principles that we think, based on our experience, will help keep you the path of PowerShell Toolmaking Righteousness.

PowerShell Tools Do One Thing

When you buy a wrench or a hammer, they are designed to fill a specific need. You wouldn't use a wrench to pound in a nail much less use a hammer to tighten a bolt. This philosophy carries over into our world. The Prime Directive for a PowerShell tool is that *it onlyt does one thing*. You can see this in almost every single PowerShell tool - that is, *command* - that ships with PowerShell. `Get-Service` gets services. It doesn't stop them. It doesn't read computer names from a text file. It doesn't modify them. For that we can use `Set-Service` or `Stop-Service`. Commands in PowerShell do *one thing*.

This critical concept is one we see newcomers violate the most. For example, you'll see folks build a command that has a `-ComputerName` parameter for accepting a remote machine name, as well as a `-FilePath` parameter so that you can alternately read computer names from a file. That's Dead Wrong, because it means the tool is doing two things instead of just one. A correct design would be to stick with the `-ComputerName` parameter, and let it accept strings (computer names) from the pipeline. You could also feed it names from a file by using a `-ComputerName (Get-Content filename.txt)` parenthetical construct. The `Get-Content` command reads text files; you shouldn't duplicate that functionality without a really strong reason.

PowerShell Tools are Testable

Another thing to bear in mind is that - if you're trying to make tools like a real pro - you're going to want to create automated units tests for your tools. We'll actually get into how that's done later in this book, but from a design perspective, you want to make sure you're designing tools that are, in fact, testable. And trust us. In the long run you'll be glad that you've designed your tools with testing in mind.

One way to do that is, again, to focus on tightly-scoped tools that do just one thing. The fewer pieces of functionality a tool introduces, the fewer things and permutations you'll have to test. The fewer logic branches within your code, the easier it will be to thoroughly test your code using automated

unit tests. This also means when someone reports a bug in your code, it will be easier to find and resolve.

For example, suppose you decide to design a tool that will query a number of remote computers. Within that tool, you might decide to implement a check to make sure each computer is online and reachable, perhaps by pinging it with `Test-NetConnection`. It makes sense in your head. But it might be a **bad idea**. First, your tool is now doing two things: querying whatever it is you're querying, but also pinging computers. That's two distinct sets of functionality. The pinging part, in particular, is likely to be code you'd use in many different tools, suggesting it should, in fact, be its own tool. Having the pinging built into the same querying tool will make testing harder, too, because you'll have to explicitly write tests to make sure that the pinging part works the way it's supposed to.

An alternate approach would be to write that "Test-PCConnection" functionality as a distinct tool. This code could use existing PowerShell commands wrapped up with features that make sense in your environment. So, if your "querying" tool is something like "Get-SystemData," you might concoct a pattern like:

```
Get-Content computernames.txt | Test-PCConnection | Get-SystemData
```

The idea being that `Test-PCConnection` would filter out whatever computers weren't reachable, perhaps logging the failed ones in some fashion, so that `Get-SystemData` could just focus on its one job of querying something. Both tools would then become easier to independently test, since they'd each have a tightly scoped set of functionality.

If you needed to take this a step further, say by automating this task for the help desk, you can put this pattern into a *control script*. Which is something we'll cover later.

You also want to avoid building functionality into your tools that will be difficult to test. For example, you might decide to implement some error logging in a tool. That's great - but if that logging is going to a SQL Server database, it's going to be trickier to test and make sure that the logging is working as desired. Logging to a file might be easier, since a file would be easier to check. An even better approach would be to write *a separate tool* that handles logging. You could then test that tool independently, and simply *use it* within your other tools. This gets back to the idea of having each tool do one thing, and one thing only, as a good design pattern.

PowerShell Tools are Flexible

You want to design PowerShell tools that can be used in a variety of scenarios. This often means wiring up parameters to accept pipeline input. For example, suppose you write a tool named `Set-MachineStatus` that changes some setting on a computer. You might specify a `-ComputerName` parameter to accept computer names. Will it accept one computer name, or many? Where will those computer names come from? The correct answers are, "always assume there will be more than one,

if you can" and "don't worry about where they come from." You want to enable, from a design perspective, a variety of approaches.

It can help to actually sit down and write some examples of using your command that you *intend to work*. These can become help file examples later, but in the design stage can help make sure you're designing to allow all of these. For example:

```
Get-Content names.txt | Set-MachineStatus
Get-ADComputer -filter * | Select -Expand Name | Set-MachineStatus
Get-ADComputer -filter * | Set-MachineStatus
Set-MachineStatus -ComputerName (Get-Content names.txt)
```

That third example is going to require some careful design, because you're not going to be able to pipe an Active Directory Computer object to the same `-ComputerName` parameter that also accepts a String object from `Get-Content`! You may have identified a need for two parameter sets, perhaps one using `-ComputerName <string[]>` and another using `-InputObject <ADComputer>` to accommodate both scenarios. Now, creating two parameter sets is going to make the coding, and the automated unit testing, a bit harder - so you'll need to decide if the tradeoff is worth it. Will that third example be used so frequently that it justifies the extra coding and test development? Or will it be a rare enough scenario that you can exclude it, and instead rely on the similar second example?

The point is that every design decision you make will have downstream impact on your tool's code, its unit tests, and so on. It's worth thinking about those decisions up front, which is why it's called the *design phase*!

Who and How

A lot of the ideas we're discussing here come down to a fundamental set of questions. Who will be using your tools and what are their expectations? Are you writing a command for the help desk to use but they have minimal PowerShell experience? Are you creating something that you will be using? Will the use be importing Excel spreadsheets to pass as parameter values? What type of output do you need and how might it be used? Will your tool be used in conjunction with something else you are developing? The more you understand who will be using your tool and their expectations, the better your result.

PowerShell Tools Look Native

Finally, be very careful with tool and parameter names. Tools should always adopt the standard PowerShell *Verb-Noun* pattern, and should only use the most appropriate verb from the list returned by `Get-Verb`. Microsoft also publishes that list online[12] and the online list includes incorrect variations and explanations that you can use to check yourself.

[12] https://docs.microsoft.com/en-us/powershell/scripting/developer/cmdlet/approved-verbs-for-windows-powershell-commands

When you run `Get-Verb` in PowerShell 7, Microsoft has added a description for each verb.

Don't beat yourself up *too* hard over fine distinctions between approved verbs, like the difference between `Get` and `Read`. If you check out that website, you'll realize that `Get-Content` should probably be `Read-Content`; likely a distinction Microsoft came up with *after* `Get-Content` was already in the wild.

The noun portion of your name should be singular, although we have been known to bend this rule from time to time. What you should consider is the potential for a naming conflict. Yes, there are ways to call the right PowerShell command but it can be a bit cumbersome so why not avoid it altogether. We suggest you get in the habit of using a short prefix on your command's noun. For example, if you work for DonCo, Inc., then you might design commands named `Get-DCISystemStatus` rather than just `Get-SystemStatus`. The prefix helps prevent your command name from conflicting with those written by other people and it will make it easier to discover and identify commands and tools created for your organization.

Parameter names should also follow native PowerShell patterns. Whenever you need a parameter, take a look at native PowerShell commands and see what parameter name they use for similar purposes. For example, if you need to accept computer names, you should use `-ComputerName` (notice it's singular!) and not some variation like `-MachineName`. If you need a filename, that's usually `-FilePath` or `-Path` on most native commands. Later in the book we'll show you how to create a parameter alias. This let's you use the official and proper parameter name, like `-Computername` but still let users of your tool use something like `-Server`.

An Example

Before we even start thinking about our design decisions, we like to review the business requirements for a new tool. We try to translate those business requirements to usage examples, so it's clearer to us how a tool might be used. If there are other stakeholders to involve - such as the people who might consume this tool, once it's done - we get them to sign off on this "functional specification," so that we can go into the design phase knowing with clear, mutual expectations for the new tool. We also try to capture *problem statements* that this new tool is meant to solve, because those sometimes offer a clearer business perspective than a specification that someone else may have written. The discussion might go something like this:

> We've been managing servers in our data center with PowerShell and PowerShell remoting. With the arrival of PowerShell 7, we will be able to use SSH for PowerShell remoting. We won't switch every server to SSH because some servers may rely on Just Enough Administration. We need to inventory servers and identify which servers have PowerShell remoting enabled. And if it is enabled, is it also configured to use SSL. Long term, we need to move servers that must use WSMan remoting to use SSL. We also need to know if a server already has `ssh` installed.

That statement would drive some more detailed questions from us, asking for specifics on what the tool needs to query. Suppose the answer came back as:

- Computer host name
- WSMan remoting enabled
- WSMan ports in use
- WSMan protocols
- Is SSH enabled
- A report date

There are a number of PowerShell commands to use, although `Test-NetConnection` might be a good choice. We can plan on writing a tools called `Get-RemoteListeningConfiguration`. It is of course going to have a `-ComputerName` parameter that accepts one or more computer names as strings. Looking ahead with a PowerShell 7 migration coming, it might also be helpful to know the operating system and current PowerShell version. That information *could* be part of this command, but since the information doesn't really apply to remoting, it might be best to keep that as a separate tool. We might also take into account how computer names will be fed to the command and what we might do with the output. We'll assume that some computers won't respond, so we'll design a way to deal with that situation.

Our design usage examples might be pretty simple:

```
Get-RemoteListeningConfiguration -Computername SRV1
Get-RemoteListeningConfiguration -Computername SRV1,SRV2
Get-Content servers.txt | Get-RemoteListeningConfiguration
Import-CSV servers.csv | Get-RemoteListeningConfiguration
Get-RemoteListeningConfiguration (Get-ADComputer -filter *).Name
```

The output of the command should be structured and predictable. That is, given a specific set of inputs, we should get the same output, which will make this a fairly straightforward design to write unit tests for. Our command is only doing one thing, and has very few parameters, which gives us a good feeling about the design's tight scope.

So we'd take that back to the team and ask what they thought. Almost invariably, that will generate questions.

How will we know if a machine fails? Will the tool keep going? Will it log that information? Does it *need* to log anything?

OK - we might need to evolve the design a bit. We know that we need to keep going in the event of a failure, and give the user the option to log failures to, perhaps, a text file. Provided the team was happy with a text file as the error log, we're good including that in the design. If they wanted something more complicated - the option to log to a database, or to an event log - then we'd design a separate logging tool to do all of that. For the sake of argument, though, let's say they're OK with the text file.

```
Get-Content servers.txt | Get-RemoteListeningConfiguration -LogPath errorlog.txt
```

Let's say that the team is satisfied with these additions, and that we have our desired usage examples locked down. We can now get into the coding. But before we do, why don't you take a stab at *your own* design exercise?

Your Turn

If you're working with a group, this will make a great discussion exercise. You won't need a computer, just a whiteboard or some pen and paper. The idea is to read through the business requirements and come up with some usage examples that meet the requirements. We'll provide *all* of the business requirements in a single statement, so that you don't have to "go back to the team" and gather more information.

Start Here

Your team has come to you and asked you to design a PowerShell tool that will help them automate a repetitive and critical, task. Team members are skilled in *using* PowerShell, so they just need a command, or set of commands, that will help automate this task.

Members of the HelpDesk, as well as the Server Management Team, periodically manually run a set of commands to get performance information that can be used to assess the status or health of system. This information is often used to generate reports and to populate a trend database. The information gathered includes these values:

- Computer name
- Total number of processes
- Total processor load
- % free physical memory
- % free space on drive C:
- The computer's uptime

It would be helpful to have a single command to run that would generate a unified result. It needs to be able to process multiple computers. And it will need to take credentials into account. It would also be helpful to have some sort of optional logging mechanism for failures.

Your Task

Your job is to design the tool that will meet the team's business requirements. You are **not** writing any code at this point. When creating a new tool, you have to consider who will use the tool, how they

might use it and their expectations. And the user might be you! The end-result of your design will be a list of command usage examples (like we have shown you previously), which should illustrate how each of the team's business needs will be solved by the tool. It's fine to include existing PowerShell commands in your examples, if those commands play a role in meeting the requirements.

Stop reading here and complete the task before resuming.

Our Take

We'll design the command name as `Get-TMComputerStatus`. The "TM" stands for "Toolmaking," which is part of this book's name, since we don't have a specific company or organizational name to use. We'll design the following use cases:

```
Get-TMComputerStatus  SRV1,SRV2
Get-Content servers.txt | Get-TMComputerStatus -credential company\administrator
(Get-ADComputer -filter *).name | Get-TMComputerStatus -ErrorLogFilePath err.log
```

Our intent here is that `-Verbose` will generate on-screen warnings about failures, while `-ErrorLogFilePath` will write failed computer names to a file. It might also be nice to append errors to the same log. Notice that, to make this "specification" easier to read, we've put each parameter on its own line. The command won't actually *execute* exactly like that, but that's fine - clarity is the idea at this point.

```
Get-TMComputerStatus -Computername SRV1,SRV2,SRV3 `
                     -Credential "company\administrator" `
                     -ErrorLogFilePath "statuserrors.txt" `
                     -ErrorAppend
```

This example uses `-ErrorLogFilePath` and `-ErrorAppend` to indicate logging errors to a text file and appending.

And because the output might be used to generate reports or be consumed by other applications, we'll want to make sure our code will meet these scenarios:

```
Import-CSV servers.csv | Get-TMComputerStatus -credential $cred |
Export-CSV status.csv

Get-Content group1.txt | Get-TMComputerStatus |
ConvertTo-HTML <parameters>

Get-TMComputerStatus (Get-ADComputer -filter *).name |
Sort-Object Computername |
Format-Table -GroupBy Computername -property Date,Uptime,Pct* |
Out-File report.txt
```

We're illustrating two things here. First is that we can accept an imported CSV file, assuming it has a Computername column. Our output is also consumable by standard PowerShell commands like `ConvertTo-HTML`, which implies that `Format-` commands and `Export-` commands will also work.

Let's Review

Let's quickly review some of the key concepts from this chapter, just to make sure you've got them all firmly in mind. See if you can answer these questions:

1. What's the "prime directive" of PowerShell tool design?
2. What verb should you use instead of "Ping" in a command name?
3. What verb should you use instead of "Delete" in a command name?
4. What's the most important output from a tool design process?
5. What is one downside of having an overly complex tool?

Review Answers

Not all of our questions lend themselves to easy, black-and-white answers, so if you didn't answer exactly the way we did, it doesn't necessarily mean you're wrong. But here's how we'd have answered:

1. Make tools that do one thing, and one thing only.
2. Test.
3. Remove.
4. A comprehensive set of usage examples for the proposed tool.
5. It will be harder to write comprehensive unit tests for it.

Start with a Command

Before we ever open up a script editor, we start in the basic PowerShell command-line window. This is your "lowest common denominator" for testing, and it's a perfect way to make sure that the commands your tool will run are actually correct. It's way easier to debug or troubleshoot a single command in the window than it is to debug an entire script. And by "single command" we mean a PowerShell expression.

If you've already read the previous chapter, then you know that we've been asked to develop a tool that will query the following information:

- Computer name
- Total number of processes
- Total processor load
- % free physical memory
- % free space on drive C:
- The computer's uptime
- a time stamp

We'll plan to use CIM or CIM-related commands for much of this. We also know we're going to need to write a text log file in the event of errors. There's more in terms of the tool itself we'll need to do, like adding a datetime stamp, but these are the basic units of functionality we need to figure out.

We're going to assume that you already know how to run PowerShell commands. If that's not your strong suit, please stop and go read *Learn Windows PowerShell in a Month of Lunches*, because it's all about discovering and running commands. Our point here is that we want to test and make sure we know how to actually accomplish everything our tool needs to accomplish, by manually running commands in the command-line window.

In our specific case, we want to also make sure we know how to reliably retrieve all of the information in our list, which is going to involve more than one WMI/CIM class. We'll need `Win32_OperatingSystem` and `Win32_Processor` at the least. We could use `Win32_LogicalDisk` to get C: drive information. Or we can use CIM-related commands like `Get-Volume`. Again - you should know how to do these things already if you're reading this book, so we're not going to walk through that entire "discovery" process.

We're planning to use `Get-CimInstance` to do the actual querying, and because we'll eventually end up querying multiple classes, we'll use `New-CimSession` and `Remove-CimSession` to create (and then remove) a persistent connection to each computer, so that we can run all the queries over one

connection. We'll need to be able to detect errors in case the connection doesn't work. Review the help for `New-CimSession` if you're not familiar with those tasks.

For now we'll assume that all computers can be reached using the CIM cmdlets. If they can't, that probably means the server is in need of a major upgrade, and we'll want to be able to log those failures.

Since so much of this tool is getting performance related data, you may be wondering about `Get-Counter`. This is a command designed to get performance information and can probably gather most of what we need. For us there are a few potential gotchas. First, `Get-Counter` relies on legacy remoting protocols like RPC and DCOM. It is not firewall friendly. Second, the output from `Get-Counter` is a little convoluted and would take a few extra steps to get the values we're looking for. `Get-CimInstance` is a better (read that as modern) approach. And it will be easier to get the values we need. But this is exactly the type of discussion you need to have. What are the best PowerShell tools to use? What are the implications if we go with one over another?

Your Turn

The previous chapter also included an exercise for you, and this one picks up where it left off. This is where you'll get to practice what we've preached in this chapter: making sure you know how to accomplish everything that your tool will need to do, by starting in the PowerShell command-line window.

Start Here

Remember that you are designing a tool that will get remoting configuration information for a remote server. You don't have to write any code at this point. Instead, mock up a sample output. What PowerShell cmdlets do you think you will need to run? Come up with a list of examples on how the command might be used. List what requirements you might need in order for your code to run.

Your Task

Stop reading here and complete the task before resuming.

Our Take

For *our* exercise, we are creating a tool to get status information about a server. We know that we assume at a minimum that the command can be run passing a single computername and writing an object to the pipeline.

```
PS C:\> Get-TMComputerStats -Computername SRV1

Computername  : SRV1
Processes     : 275
CPULoad       : 13
PctFreeMemory : 50.9874975465509
PctFreeDrive  : 39.1990451499511
Uptime        : 1.20:48:21.2817144
Date          : 7/20/2020 2:09:16 PM
```

We also assume the person running the script will import a list of computernames and want to be able to export to a CSV file. If there are errors they should be written to a log file.

```
PS C:\> Get-Content c:\work\servers.txt |
Get-TMComputerstatus -ErrorLog c:\work\tmstatus-error.log -errorappend |
Export-CSV D:\status.csv -append
```

We'll need commands like these to generate the necessary results:

- `Get-Date`
- `Get-Volume`
- `Get-CimInstance -class win32_OperatingSystem`
- `Get-CimInstance -class win32_processor`
- `Get-CimInstance -class win32_process`
- `Out-File`

Let's Review

Here are a few questions you can answer to make sure you've picked up the key elements of this chapter:

1. What is the value of testing commands in the command-line before you start writing a script?
2. Why is it important to understand how your command might be used?
3. Why is it critical that your command write an object to the pipeline?

Review Answers

Here are our answers to the questions. Note that these aren't necessarily the *only* correct answers, but hopefully they'll help you understand if you grasped the chapter's material or not.

1. Individual commands are easier to test and fix in the command-line window. Using tested commands in your script will help reduce the number of bugs that you introduce while coding. Remember, that if you command fails at the console it won't work in your script any better.
2. If you don't know who your command will be used, you won't be able to include necessary features like input from the pipeline.
3. If your command doesn't write an object to the pipeline, than you can't process the results with other PowerShell commands like `Where-Object`, `Sort-Object` or `Out-Gridview`.

Build a Basic Function and Module

In this chapter, we'll start creating the tool that we designed in "Tool Design," using some of the commands that we figured out and tested in the previous chapter. It's **very important to understand** that this chapter isn't going to attempt to build the entire tool, or attempt to solve the entire business statement from our Design chapter. We'll be taking things one step at a time, because it's really the *process* of toolmaking that we want to demonstrate for you.

Start with a Basic Function

Basic functions have existed in PowerShell since v1, and they are one of the many types of *commands* that PowerShell understands (some of the others being cmdlets, applications, workflows, and so on). Functions make a great unit of work for toolmaking, so long as you follow a basic principle of *keeping your function tightly scoped and self-contained*. We've written already about the need to have tightly scoped functions - that is, functions that do just one thing. *Self-contained* means that the function needs to live in its own little world, and become a kind of little black box. Practically speaking, that means two things:

Information to be used inside the function should come only from declared input parameters. Of course, some functions may "look up" data from elsewhere, like a database or a registry, and that's fine if it's what the function does. But functions shouldn't rely on external variables or other sources. You want them as self-contained as possible.* Output from a function should be to the PowerShell pipeline *only*. Stuff like creating a file on disk, updating a database, and other actions aren't *output*, they're *actions*. Obviously, a function can perform one of those actions *if that's what the function does*.

Design the Input Parameters

Looking back through the design for your function to get computer status details, what information will the function need? Your usage examples should already provide pretty clear guidance on what parameters you'll have to create, which is one reason we recommend creating usage examples as tht primary design deliverable. So now let's create basic versions of those parameters.

```
Function Get-TMComputerStatus {
 Param(
  [string[]]$Computername,
  [string]$ErrorLogFilePath,
  [switch]$ErrorAppend
 )
}
```

Notice how careful we're being with the formatting of this code? In order to conserve space in this book, we're only indenting the code a little bit within the function and within the Param() block, but you'll typically indent four spaces (which, in most code editors, is what the Tab key will insert). **Do not get lazy about your code formatting**. Lazy formatting is a sign of the devil, and is a sure sign of code that probably has bugs - and will be hard to debug.

In the Param() block, we've only had to declare a few parameters. The Computername is pretty straightforward. Although thinking ahead, we might need to take alternate credentials into account. for the computer to query. We also have parameters related to the logging we want to do.

For those of you looking for some nits to pick, yeah, there's probably one here. An argument could be made that logging is a second thing the command is doing and it should not be part of this command. On the other hand, a counter argument might be that the task of getting computer status information by design includes logging. Semantics. But if nothing else, including these parameters offers us some "teachable moments" so we're going to go with it.

Parameter definitions are very simple declarations, and we'll build on these in upcoming chapters. For now, here are some things to notice:

- Data types are enclosed in square brackets. Common ones include `[string]`, `[int]`, and `[datetime]`.
- Parameters become variables inside the function, so their names are preceded with a `$`. And for God's sake, don't try to create a parameter name with spaces!
- Each parameter is separated from the next by a comma. You don't *have* to put them one-per-line as we've done, but when we start building on these, it'll be a lot easier to read if they're broken out one per line.
- The `-ComputerName` parameter will accept one or more values in an *array*, which is what `[string[]]` denotes.

Write the Code

Now let's insert some basic functional code. Again **this will not complete the tool's entire mission** - we're just getting started, and we want to walk you through each step.

```
Function Get-TMComputerStatus {
 Param(
  [string[]]$Computername,
  [string]$ErrorLogFilePath,
  [switch]$ErrorAppend
 )

  foreach ($computer in $Computername) {
    # get data via Get-CimInstance and Get-Volume
    # create output object
  } #foreach #computer

} #Get-TMComputerStatus
```

Notice that we tagged a `#Get-TMComputerStatus` comment on the closing bracket of the function? That's a good habit to get into when you have a closing bracket, as it can help remind you which construct the bracket closes. You'll appreciate this when you have a script file with multiple functions defined.

We know we'll need to run `Get-CimInstance` and maybe `Get-Volume`. At some point all of these outputs will need to be condensed down to a simple, unified object. The basic command also doesn't include any error handling at this point. But this should be enough to give us a feel for how the command will be structured and how the execution will flow internally.

Details: The ForEach Construct

Our code uses a `ForEach` construct. It works a bit like PowerShell's `ForEach-Object` command, but it has a different syntax:

```
ForEach ($item in $collection) {
    # code
}
```

The second variable - $collection here, and $ComputerName in our preliminary function - is expected to contain zero or more items. The `ForEach` *loop* will execute its {script block} one time *for each* item that is contained in the second variable. So, in `Get-RemoteListeningConfiguration`, if we provided three computer names to the `-ComputerName` parameter, the `ForEach` loop would run three times. Each time the loop runs, one item is taken from the second variable and placed into the first. So, within the script block above, $item will contain one thing at a time from $collection. In the defined function, $computer will contain one string at a time, taken from $ComputerName.

You'll often see people use singular and plural words in their `ForEach` loops:

```
$names = Get-Content names.txt
ForEach ($name in $names) {
    # code
}
```

That approach makes it easier to remember that $name contains one thing from $names, but that's purely for human readability. PowerShell doesn't magically know that "name" is the singular of "names," and it doesn't care. The above could easily be rewritten as:

```
$unicorns = Get-Content names.txt
ForEach ($purple in $unicorns) {
    # code
}
```

And PowerShell would be perfectly happy. That code would be a lot harder to read and keep track of, though. In our sample function, you'll notice that our second variable is *not* plural, right?

```
 foreach ($computer in $computername) {
```

That's because $ComputerName is one of our function's input parameters. PowerShell's convention is to use singular words for command and parameter names. You won't see `-ComputerNames`; you only see `-ComputerName` as a parameter. We stuck with the convention, and so our `ForEach` loop doesn't follow a singular/plural pattern. Again, PowerShell itself doesn't care, and we feel it's more important that our "outward-facing" elements - command and parameter names - follow PowerShell naming conventions. And, honestly, using $computer as our one-item-at-a-time variable name saves a little space in this book.

Design the Output

Finally, we need to have the command output something. We'll add that in this version.

```
Function Get-TMComputerStatus {
  Param(
    [string[]]$Computername,
    [string]$ErrorLogFilePath,
    [switch]$ErrorAppend
  )

  foreach ($computer in $Computername) {
    $OS = Get-CimInstance win32_operatingsystem -computername $computer |
    Select-Object -property CSName,TotalVisibleMemorySize,FreePhysicalMemory,
    NumberOfProcesses,
```

```
    @{Name="PctFreeMemory";Expression = {($_.freephysicalmemory/`
    ($_.TotalVisibleMemorySize))*100}},
    @{Name="Uptime";Expression = { (Get-Date) - $_.lastBootUpTime}}

    $cpu = Get-CimInstance win32_processor -ComputerName $computer |
    Select-Object -Property LoadPercentage

    $vol = Get-Volume -CimSession $computer -DriveLetter C |
    Select-Object -property @{Name = "PctFreeC";Expression = `
    {($_.SizeRemaining/$_.size)*100 }}

    $os,$cpu,$vol

  } #foreach $computer

} #Get-TMComputerStatus
```

The output in the book has been formatted to fit the page.Normally you wouldn't need to break lines using the back tick. This code works and writes results to the pipeline. However, this version is breaking the cardinal rule of writing more than 1 type of object to the pipeline. We'll need to refine this code to get a single object type out of it. That's coming up in the next few chapters.

Create a Script Module

Our last step will be to save all of this code as a *script module*. These are supported on PowerShell v2 and later, and should ideally be stored in one of the paths specified in the `PSModulePath` environment variable. On PowerShell v4 and later, the default path includes C:Program FilesWindowsPowerShellModules, so that's where we'll create our module. Specifically, we'll save it as C:Program FilesWindowsPowerShellModulesToolmakingToolmaking.psm1. Notice that the **subfolder name** and the **filename** must match in order for PowerShell to automatically discover our module and load it on-demand.

We're also going to create a *module manifest*. This isn't strictly required, but it's a good idea. A properly designed manifest can speed up PowerShell's module auto-discovery, and as our module becomes more complex will offer us some useful features. To create the manifest, we'll run:

```
Cd "\Program Files\WindowsPowerShell\Modules\Toolmaking"
New-ModuleManifest -Path .\Toolmaking-Prelim.psd1
                   -Author "Don Jones & Jeff Hicks"
                   -RootModule toolmaking.psm1
                   -FunctionsToExport @('Get-TMComputerStatus)
                   -Description "Sample Toolmaking module"
                   -ModuleVersion 1.0.0.0
```

As is our custom in this book, we've put each parameter on its own line for readability. In practice, you'd type all of the above as one long line which you'll see in our final code samples. Notice:

- The manifest name **must match the module subfolder name**.
- The manifest file gets a .psd1 filename extension.
- The root module is the .psm1 file that we already created. It is typically in the same folder as the manifest.
- `-FunctionsToExport` is an array of functions that we want people to be able to "see" within our module. This is optional, but using it speeds up PowerShell's auto-discovery magic. Technically you can use wildcards but the recommendation is to use complete command names.
- The author and description are optional, but it's not a bad idea to include them especially if you intend on publishing or sharing your module with the world.
- The module version is also optional, but it's highly recommended, and becomes required if you're going to publish your module to a repository (which we will eventually be doing).

We've included our module, to this point, in the code samples for this book (which are available by running `Install-Module PowerShell-Toolmaking` in PowerShell). You'll find it in the Chapters folder, under this chapter's title. To load the module, you'll need to manually run `Import-Module` and provide the full path to our .psd1 file on your computer. In the code samples for this chapter, our module name is Toolmaking-Prelim, to avoid conflicting with the "finished" Toolmaking module that's part of the code sample download.

Pre-Req Check

Before we test our command, especially if you're planning to run it yourself and follow along, you need to check a few things.

- Make sure your PowerShell window always says "Administrator" in the title bar. If not, run the shell "as Administrator" by right-clicking the PowerShell Task Bar icon and selecting the appropriate option.
- Run `Get-ExecutionPolicy`; the result should be RemoteSigned, Bypass, or Unrestricted. If not, use `Set-ExecutionPolicy` to change the setting to one of those. We use Bypass because we are working on a non-production system in a secure environment.

Running the Command

Now for the real test. First, **close your PowerShell window**. That will ensure our test is in a "clean" PowerShell environment. Then open a new one (make sure it's "as Administrator"), and run:

```
Get-TMComputerStatus -Computername localhost
```

You should get some output from that. In fact, you should just be able to type `Get-TMCo` and hit Tab, type a space, type `-Comp` and hit Tab, and then type a space and `localhost`. If the Tab completion isn't working, then double-check your script for proper file names, any typos in the code (indicated in the PowerShell ISE by red squiggly underlines), and so on. Also make sure that you've used a path that's in your machine's PSModulePath environment variable:

```
$env:PSModulePath
```

If the command runs without trouble, then you're good to go. Take some time to make sure you understand *why* each line of code is in the command, and that you can explain the reason for each step we've performed to this point.

If you make any changes to your module, it is very important to understand that PowerShell won't "see" those changes. That's because it loaded your module into memory when you first ran your command; afterwards, it runs entirely from memory, and doesn't re-load from disk. So if you make any changes to your code, you need to do one of two things:

- Close the PowerShell console window in which you've been testing, and open a new one. This is a sure-fire way to make sure you get a fresh start every time.
- Unload your module, and then run your command again to re-load your module. In our case, that would mean running `Remove-Module Toolmaking`, since "Toolmaking" is our module name (as defined by the subfolder name and the .psd1 filename).
- Or you can try to manually force PowerShell to re-import the module with the command `Import-Module Toolmaking -force`.

You'll also notice that we tend to test our command in a normal PowerShell console window, even through we're developing in something like the PowerShell ISE, Visual Studio Code, etc. That's because development environments sometimes have a slightly different way of running scrips, and the console window represents the "standard" way that your script will run in production. Since the console represents the "production environment," that's where we test.

Your Turn

Let's return to the tool that we asked you to design in the Design chapter to get remote listening configuration information. It's time to start coding it up.

Start Here

To review, we designed the command name as `Get-TMRemoteListeningConfiguration`. The "TM" stands for "Toolmaking," which is part of this book's name, since we don't have a specific company or organizational name to use. We'll design the following use cases:

```
Get-TMRemoteListeningConfiguration -ComputerName SERVER1,SERVER2
                  -ErrorLog failed.txt
                  -Verbose
```

Our intent here is that `-Verbose` will generate on-screen warnings about failures, while `-ErrorLog` will write failed computer names to a file. Notice that, to make this "specification" easier to read, we've put each parameter on its own line. The command won't actually *execute* exactly like that, but that's fine - clarity is the idea at this point.

```
Get-TMRemoteListeningConfiguration -ComputerName SERVER1,SERVER2
```

This example illustrates that `-ErrorLog` and `-Verbose` were optional. We also want to illustrate some of our flexible execution options:

```
Get-Content servers.txt | Get-TMRemoteListeningConfiguration
```

This illustrates our ability to accept computer names from the pipeline. Finally:

```
Import-CSV computers.csv | Get-TMRemoteListeningConfiguration | ConvertTo-HTML
```

We're illustrating two things here. First is that we can accept an imported CSV file, assuming it has a column named Computername. Our output is also consumable by standard PowerShell commands like `ConvertTo-HTML`, which also implies that `Format-` commands and `Export-` commands will also work.

Your Task

Create a basic function named `Get-TMRemoteListeningConfiguration`. Specify all of the parameters that are listed in the design, although right now we might not actually use all of those parameters. Write enough code so that, given a computer name you can get the desired configuration information.

- Is it listening on port 5985
- Is it listening on port 5986
- Is it listening on port 22

Don't worry about piping computer names into the function at this point. But you will need to use a `ForEach` construct.

Create the function in a script module named TMTools. Test your function against your computer, ie local host. For now, don't worry about the logging or other features specified in the design. Keep in mind what you learned from previous chapters. For now, it's okay to create output using `Select-Object` and custom properties. Later, we'll work on getting the output closer to the design specification. Test your command in the PowerShell console, rather than in the ISE or VS Code, and bear in mind the caveats we pointed out about unloading your module after making changes.

Our Take

Here's our solution for you to compare to your own. Minor variations shouldn't be cause for concern, provided your command works when you run it.

```
Function Get-TMRemoteListeningConfiguration {
    Param(
    [string[]]$Computername,
    [string]$ErrorLog
    )

    $ports = 22,5985,5986
    foreach ($computer in $computername) {
        foreach ($port in $ports) {
            Test-NetConnection -Port $port -ComputerName $Computer |
            Select-Object Computername,RemotePort,TCPTestSucceeded
        }
        #TODO
        #better output
        #error handling and logging
    } #foreach

} #Get-RemoteListeningConfiguration function
```

Note that we are using a nested set of ForEach loops. We're telling PowerShell, "For each computer, run though this collection of ports." The command gives us output like this:

```
ComputerName RemotePort TcpTestSucceeded
------------ ---------- ----------------
WIN10PRO             22             True
WIN10PRO           5985             True
WIN10PRO           5986            False
```

If you get odd results using `localhost` use `$env:computername` or type your actual computername.

We created a manifest for this, too:

```
New-ModuleManifest -Path TMTools.psd1
                   -RootModule .\TMTools.psm1
                   -FunctionsToExport Set-TMServiceLogon
                   -ModuleVersion 1.0.0.0
```

We've included our solution, to this point, in the code samples for this book (which are available by running `Install-Module PowerShell-Toolmaking` in PowerShell). You'll find it in the Chapters folder, under this chapter's title. To load the module, you'll need to manually run `Import-Module` and provide the full path to our .psd1 file on your computer. In the code samples for this chapter, our module name is TMTools-Prelim, to avoid conflicting with the "real" TMTools module that you're building on your own.

Let's Review

Let's make sure you picked up on the key elements of this chapter. See if you can answer the following questions:

1. To create a module named `CorpTools`, what folder and filename structure would you use?
2. What data type would accept whole numbers in a parameter?
3. What would a parameter defined as `[string[]]$Username` indicate?
4. How do you separate defined function parameters?
5. What does a `foreach` construct allow you to do?

Review Answers

Here are the answers to this chapter's review questions:

1. Something like Program FilesWindowsPowerShellModulesCorpToolsCorpTools.psd1 for the module manifest, and CorpTools.psm1, in the same folder, for the main script module file.
2. It is `[int]` or `[int32]`.
3. That the parameter accepts an array or collection of user name strings.
4. Separate each parameter definition by a comma.
5. Enumerate through an array, or collection, of items.

Adding CmdletBinding and Parameterizing

In this chapter, we'll focus entirely on the `Param()` block of our function, and discuss some of the cool things you can do with it.

About CmdletBinding and Common Parameters

Back when PowerShell v2 was being developed, Microsoft toyed with the idea of having a `cmdlet{}` construct that was essentially a superset of `function{}`. The idea was that these "script cmdlets" would exhibit all of the behaviors of a "real" cmdlet (e.g., one written in .NET and compiled into an assembly). By the time v2 released, these had become *advanced functions*, and are differentiated primarily by the `[CmdletBinding()]` attribute. To illustrate the first major difference, let's start with a *basic* function:

```
function test {
    Param(
    [string]$ComputerName
    )
}
```

That's it. No code at all. Now ask PowerShell for help with that function:

```
PS C:\> help test

NAME
    test

SYNTAX
    test [[-ComputerName] <string>]


ALIASES
    None
```

That's what we'd expect - PowerShell is producing the best help it can given the complete nonexistence of anything. Now, let's make just one change to the code:

```
function test {
    [CmdletBinding()]
    Param(
        [string]$ComputerName
    )
}
```

and again ask for help:

```
PS C:\> help test

NAME
    test

SYNTAX
    test [[-ComputerName] <string>]  [<CommonParameters>]


ALIASES
    None
```

PowerShell has added the common parameters. If you read the about_CommonParameters help file[13], you'll discover that *all PowerShell commands* support this set of parameters. The number has grown through the subsequent versions of PowerShell, and now consists of 11 parameters. Cmdlet authors don't need to do anything to make these work - PowerShell takes care of everything. And now, because we added `[CmdletBinding()]`, our function will support all of these common parameters as well. Some of the cooler ones (with availability differing based on your version of PowerShell) include:

- The `-Verbose` parameter enables the output of `Write-Verbose` in your function, overriding the global $VerbosePreference variable.
- The `-Debug` parameter enables the use of `Write-Debug` in your function.
- The `-ErrorAction` parameter modifies your function's behavior in the event of an error, and overrides the global $ErrorActionPreference variable.
- The `-ErrorVariable` parameter lets you specify a variable name, in which PowerShell will capture any errors your function generates.
- The `-InformationAction` parameter overrides the global $InformationPreference variable and enables `Write-Information` output.
- The `-InformationVariable` parameter specifies a variable in which output from `Write-Information` will be captured.

[13] https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.core/about/about_commonparameters

- The `-OutVariable` parameter specifies a variable, in which PowerShell will place copies of your function's output, while also sending copies into the main pipeline. We'll cover this more in our chapter on troubleshooting.
- The `-PipelineVariable` parameter specifies a variable, in which PowerShell will store a copy of the current pipeline element. We'll cover this more in our chapter on troubleshooting.

There are others, and we'll discuss almost all of them in more detail in upcoming chapters.

Accepting Pipeline Input

If you remember our original tool design, we specified a need to capture input from the pipeline. This requires a modification both to our parameters and to the code of the function. As a reminder, here's where we're starting after the previous chapter:

```
Function Get-TMComputerStatus {
  Param(
    [string[]]$Computername,
    [string]$ErrorLogFilePath,
    [switch]$ErrorAppend
  )

  foreach ($computer in $Computername) {
    $OS = Get-CimInstance win32_operatingsystem -computername $computername |
    Select-Object -property CSName,TotalVisibleMemorySize,FreePhysicalMemory,
    NumberOfProcesses,
    @{Name="PctFreeMemory";Expression = {($_.freephysicalmemory/`
    ($_.TotalVisibleMemorySize))*100}},
    @{Name="Uptime";Expression = { (Get-Date) - $_.lastBootUpTime}}

    $cpu = Get-CimInstance win32_processor -ComputerName $computername |
    Select-Object -Property LoadPercentage

    $vol = Get-Volume -CimSession $computername -DriveLetter C |
    Select-Object -property @{Name = "PctFreeC";Expression = `
    {($_.SizeRemaining/$_.size)*100 }}

    $os,$cpu,$vol

  } #foreach $computer

} #Get-TMComputerStatus
```

And here's our modified function:

```
Function Get-TMComputerStatus {
 [cmdletbinding()]
 Param(
  [Parameter(ValueFromPipeline=$True)]
  [string[]]$Computername,
  [string]$ErrorLogFilePath,
  [switch]$ErrorAppend
)

 BEGIN {}

 PROCESS {
  foreach ($computer in $Computername) {
    $OS = Get-CimInstance win32_operatingsystem -computername $computername |
    Select-Object -property CSName,TotalVisibleMemorySize,FreePhysicalMemory,
    NumberOfProcesses,
    @{Name="PctFreeMemory";Expression = {($_.freephysicalmemory/`
    ($_.TotalVisibleMemorySize))*100}},
    @{Name="Uptime";Expression = { (Get-Date) - $_.lastBootUpTime}}

    $cpu = Get-CimInstance win32_processor -ComputerName $computername |
    Select-Object -Property LoadPercentage

    $vol = Get-Volume -CimSession $computername -DriveLetter C |
    Select-Object -property @{Name = "PctFreeC";Expression = `
    {($_.SizeRemaining/$_.size)*100 }}

    #TODO: Clean up output
    $os,$cpu,$vol

    } #foreach $computer
}
 END {}
} #Get-TMComputerStatus
```

Here's what we did:

- We added `[CmdletBinding{}]` to the `Param()` block.
- We used blank lines to visually separate our parameters in the `Param()` block.
- We added a `[Parameter()]` *decorator*, or *attribute*, to the $ComputerName parameter. Although we physically placed it on the preceding line, PowerShell will read those two lines as one.

- In the decorator, we specified that the `$ComputerName` parameter is capable of accepting values (`[string]` values, to be specific, since that's what the parameter is) from the pipeline.
- We added `BEGIN{}`, `PROCESS{}`, and `END{}` scriptblocks.

Understanding how all this fits together requires you to remember that we want our function to run in two distinct modes, and that each mode has slightly different requirements from PowerShell.

Running Commands in Non-Pipeline Mode

Imagine running our command like this:

```
Get-TMComputerStatus -ComputerName ONE,TWO,THREE
```

In this mode, PowerShell will ignore the `BEGIN{}`, `PROCESS{}`, and `END{}` *labels*, but it won't ignore the *code within those labels*. In other words, it's simply like the labels never existed. $ComputerName will contain an array, or collection, of three `[string]` objects, "ONE", "TWO", and "THREE". Our entire command will run one time, from the first line of code to the last. Our `ForEach` loop will execute three times.

Running Commands in Pipeline Mode

Imagine running our command like this:

```
"ONE","TWO","THREE" | Get-TMComputerStatus
```

First, PowerShell will construct a three-element array, because that's what comma-separated lists do in PowerShell. It will then "scan ahead" in the pipeline, and execute the `BEGIN{}` block ONCE for each command in the pipeline. That's true both for advanced functions and for compiled cmdlets. The Begin block (which does not *have* to be in all-uppercase, and which can be omitted if you don't have any code to stick in there) is a good place to do "set up" tasks, such as opening database connections, setting up log files, or initializing arrays. Any variables you create in the Begin block will continue to exist elsewhere in your function.

Next, PowerShell will start feeding the elements from that three-element array down the pipeline, *one at a time*. So it will insert "ONE" into $ComputerName, and then run the `PROCESS{}` block. Our `ForEach` loop will execute, but only once - it's actually kind of redundant in this mode, but we need it for the non-pipeline mode. PowerShell will then feed "TWO" into $ComputerName, and run `PROCESS{}` again. It'll then put "THREE" into $ComputerName, and run `PROCESS{}` one last time.

Finally, after all the objects have been sent through the pipeline, PowerShell will re-scan the pipeline and ask everyone to run their `END{}` blocks once. Again, you can omit this if you don't have anything to put in there, but for visual purposes we like to include it even if it's empty. One suggestion is to insert a comment into empty `Begin` or `End` blocks so you don't think something is missing.

```
End {
 # intentionally empty
}
```

We also like including comments at the closing } (e.g. `} #close begin`) so you can tell where scriptblocks begin and end.

Values and PropertyNames

Notice that our example is using this decorator:

```
[Parameter(ValueFromPipeline=$True)]
```

This enables `ByValue` binding of pipeline input. You can enable this only for one parameter per data type. Since `$ComputerName` is a `[string]`, it's therefore the only `[string]` parameter we can mark as accepting pipeline input `ByValue`.

We can also enable input `ByPropertyName`:

```
[Parameter(ValueFromPipeline = $True,ValueFromPipelineByPropertyName=$True)]
```

If you're not deeply familiar with pipeline parameter input ByValue and ByPropertyName, we urge you to read *Learn Windows PowerShell in a Month of Lunches* and learn all about it. It's a crucial PowerShell feature.

Now, if the object in the pipeline isn't a System.String, but it has a ComputerName *property*, our `$ComputerName` variable will pick that up as well.

In the parameter attribute you will see that we assigned a value of `$True` to `ValueFromPipeline`. In earlier versions of PowerShell this was required and you'll find plenty of examples in the wild. Along with other settings like `Mandatory` and `ValueFromPipelineByPropertyName`. However, specifying $True is redundant. If you decorate a parameter with `ValueFromPipeline` it is automatically true. You do not have to explicitly make it so. We will probably be explicitly redundant for a bit to make sure you understand what we're doing but eventually, we'll drop the explicit assignment. This also applies to other parameter settings like `Mandatory`.

Mandatory-ness

Because our function can't really run correctly without a computer name, we want to ensure at least one is always provided. Here's our revised set of parameters:

```
Param(
  [Parameter(ValueFromPipeline=$True,Mandatory=$True)]
  [string[]]$Computername,
  [string]$ErrorLogFilePath,
  [switch]$ErrorAppend
)
```

Some notes on our decision-making process, here:

- Making `$ComputerName` mandatory makes sense. If a value isn't provided, PowerShell will prompt for it, and then fail with an error if one still isn't given.
- Making `$ErrorLogFilePath` mandatory doesn't make sense, because we don't want to force people to log errors. We'll check to see if this is provided, and enable logging accordingly.

If one of your parameters is a `[Switch]` avoid making it mandatory, because you're essentially forcing it to be $True (or forcing someone to run `-Enable:$false` to turn it off, which is awkward).

Parameter Validation

Our Computername parameter has a potential weakness, in that it'll accept any string whatsoever. We have no way of knowing what data might be fed into this parameter or even how.

```
Param(
  [Parameter(ValueFromPipeline=$True,Mandatory=$True)]
  [ValidateNotNullorEmpty()]
  [string[]]$Computername,
  [string]$ErrorLogFilePath,
  [switch]$ErrorAppend
)
```

Here, we've added a `[ValidateNotNullOrEmpty()]` attribute to our `$Computername` parameter. PowerShell will complain if the value is empty or null. Again, we have no way of knowing how the user of our function is getting a value for Computername. We might even take this a step further and add a second validation.

```
Param(
  [Parameter(ValueFromPipeline=$True,Mandatory=$True)]
  [ValidateNotNullorEmpty()]
  [ValidatePattern("^\w+$")]
  [string[]]$Computername,
  [string]$ErrorLogFilePath,
  [switch]$ErrorAppend
)
```

It is possible that the user could specify a Computername value that was a variable with nothing but spaces. This would technically pass the not null or empty test. The second test using a regular expression pattern to ensure that the value is starts and ends with a word character. Parameter values need to pass all the validation tests. Also be aware that parameter validation doesn't negate the need for error handling. All we've done so far is validate that the parameter *might* be a computername. But is it up? Do you have proper credentials? Eventually we will need to handle those situations.

There are other validation methods available, and we'll cover those in an upcoming chapter. You can also read about_functions_advanced_parameters[14] for a full list.

Parameter Aliases

Finally, although we've followed native PowerShell patterns in using `-ComputerName` as our parameter name, we might also find value in this addition:

```
Param(
  [Parameter(ValueFromPipeline=$True,Mandatory=$True)]
  [ValidateNotNullorEmpty()]
  [ValidatePattern("^\w+$")]
  [Alias("CN","Machine","Name")]
  [string[]]$Computername,
  [string]$ErrorLogFilePath,
  [switch]$ErrorAppend
)
```

Here, we've defined three aliases for our parameter, making `-CN`, `-Machine`, and `-Name` valid alternatives. The user can run the command using these alternate parameters instead of `-Computername`. Note that code within the function will use the defined parameter name internally, that is, $Computername.

[14] https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.core/about/about_functions_advanced_parameters

Your Turn

Okay, let's return to the command you started building in the previous chapter, and start making some improvements.

Start Here

Here's where *we* finished up after last chapter; you can either use this as a starting point, or use your own lab result.

```
Function Get-TMRemoteListeningConfiguration {
    Param(
    [string[]]$Computername,
    [string]$ErrorLog
    )

    $ports = 22,5985,5986
    foreach ($computer in $computername) {
        foreach ($port in $ports) {
            Test-NetConnection -Port $port -ComputerName $Computer |
            Select-Object Computername,RemotePort,TCPTestSucceeded
        }
        #TODO
        #better output
        #error handling and logging
    } #foreach

} #Get-RemoteListeningConfiguration function
```

Your Task

Go ahead and make this an advanced function, and accomplish the following:

- Ensure that the ComputerName parameter is mandatory.
- Ensure that ComputerName can accept pipeline input `ByValue`.
- Ensure that the Computername value isn't null or empty.
- Add a Computername alias of `CN`.

Our Take

Here's what we came up with. Notice especially the `PROCESS{}` label addition in the body of the code.

```
Function Get-TMRemoteListeningConfiguration {
    [cmdletbinding()]
    Param(
    [Parameter(ValueFromPipeline = $True, Mandatory = $True)]
    [ValidateNotNullorEmpty()]
    [Alias("CN")]
    [string[]]$Computername,
    [string]$ErrorLog
    )

    Begin {
      #not used
    }
    Process {
    $ports = 22,5985,5986
    foreach ($computer in $computername) {
        foreach ($port in $ports) {
            Test-NetConnection -Port $port -ComputerName $Computer |
            Select-Object Computername,RemotePort,TCPTestSucceeded
        }
        #TODO
        #better output
        #error handling and logging
    } #foreach
   }
   End {
    #not used
   }
} #Get-RemoteListeningConfiguration function
```

We've included our solution, to this point, in the code samples for this book (which are available by running `Install-Module PowerShell-Toolmaking` in PowerShell). You'll find it in the Chapters folder, under this chapter's title. To load the module, you'll need to manually run `Import-Module` and provide the full path to our .psd1 file on your computer. In the code samples for this chapter, our module name is TMTools-Prelim, to avoid conflicting with the "real" TMTools module that you're building on your own.

Let's Review

See if you can answer these questions, to help ensure that you've picked up the key facts from this chapter:

1. What does `[CmdletBinding()]` do?
2. What does PowerShell do with mandatory parameters?
3. What is the reason for parameter validation?
4. What might happen in an `END{}` block?

Review Answers

Here's what we came up with:

1. Designates a function as an *advanced function* and activates the common PowerShell parameters.
2. If a value isn't provided, then the shell will prompt for values.
3. To verify that parameter values meet at least an initial level of verification. If the validation fails the command will fail to run which is preferable to getting half-way through and then erroring out.
4. Closing a database connection, closing a log file, or other "clean up" tasks.

Emitting Objects as Output

So far, the tool we've been building isn't querying all of the information we originally specified in our design. That was a deliberate decision so that we could get some structure around the tool first. We've also held off because once we start querying a bunch of information, we need to take a specific approach to combining it, and we wanted to tackle that approach in a single chapter.

Right now, the "functional" part of our tool looks like this:

```
# Query data
$os = Get-CimInstance -ClassName Win32_OperatingSystem -ComputerName $computername
$cpu = Get-CimInstance win32_processor -ComputerName $computername
$vol = Get-Volume -CimSession $computername -DriveLetter C
```

The output is currently the collection of all these results. That's not good design.

The code we have thus far is using `Select-Object` to create some of the custom properties that we want.

```
$vol = Get-Volume -CimSession $computername -DriveLetter C |
Select-Object -property @{Name = "PctFreeC";
Expression = {($_.SizeRemaining/$_.size)*100 }}
```

The problem is that all of the properties or data is spread across multiple results. We need to assemble them into a unified result object.

Assembling the Information

We're going to move away from using backticks in some places, to keep our code's column width under the 80 character count that fits well in this book. Instead, we're going to start using a technique called *splatting*. With this technique, we construct a hash table whose keys are parameter names, and whose values are the corresponding parameter values. This is also a smart step in our particular case because we're calling the same command multiple times with only slight parameter variations. Here's what splatting looks like.

```
$params = @{ClassName = 'Win32_OperatingSystem'
            ComputerName = 'CLIENT1'
            }
```

We are defining a hashtable. You can call it anything you want. Put each parameter on a new line. For switch parameters, assign a value of `$True`.

```
$params = @{ClassName = 'Win32_OperatingSystem'
            ComputerName = 'CLIENT1'
            Verbose = $True
            }
```

You then feed those to the command by prefixing the variable name with `@` instead of `$`:

```
Get-CimInstance @params
```

There, now you can tell your family you splatted today ;). So here's our revised chunk of code that queries the information we need into variables:

```
foreach ($computer in $Computername) {
    $params = @{
            Classname = "Win32_OperatingSystem"
            Computername = $computer
            }
    $OS = Get-CimInstance @params

    $params.ClassName = "Win32_Processor"
    $cpu = Get-CimInstance @params

    $params.className = "Win32_logicalDisk"
    $vol = Get-CimInstance @params -filter "DeviceID='c:'"

    #TODO: Clean up output

} #foreach $computer
```

A couple of notes on this snippet. One of the benefits of splatting is that because you are using a hashtable you can modify it on the fly. In this case, all of the calls to `Get-CimInstance` are using the same computer name. The only thing that is changing is the class name. You can assign a new value by using the dotted notation. And using splatting doesn't mean you can't use other parameters. We decided to switch out the `Get-Volume` command to also use `Get-CimInstance`. We are splatting the core parameters but also adding a filter parameter.

Constructing and Emitting Output

What we **absolutely do not want to do** at this point is output *text*. PowerShell should never use `Write-Host` for tool output, because that output would be drawn directly on the screen as text. We couldn't re-use, re-direct, or re-anything that output, which isn't the point of a reusable tool. Instead, our tools should *always* output structured data in the form of objects, just like "real" PowerShell commands do. In this case, there is data from 3 different objects. What we're going to do is create a new, single object and assign property values from the collected data.

You probably noticed the snippet above doesn't have all of the `Select-Object` statements to define all the custom properties. That can get a bit clunky especially when you compare it to this.

```
#Output data
$props = @{
    Computername = $os.CSName
    TotalMem     = $os.TotalVisibleMemorySize
    FreeMem      = $os.FreePhysicalMemory
    Processes    = $os.NumberOfProcesses
    PctFreeMem   = ($os.FreePhysicalMemory/$os.TotalVisibleMemorySize)*100
    Uptime       = (Get-Date) - $os.lastBootUpTime
    CPULoad      = $cpu.LoadPercentage
    PctFreeC     = ($vol.FreeSpace/$vol.size)*100
}
$obj = New-Object -TypeName PSObject -Property $props
Write-Output $obj
```

Again, some notes:

- Were constructing a hash table - not unlike when splatting - that holds our output. Each key in the hash table is a property name we want to output, and each value is the corresponding data for that property.
- You can make the property names anything you want. You don't have to use the object's original property name.
- We're constructing values here instead of using `Select-Object`.
- We use `New-Object` to construct a blank object and attach our properties and values.
- We don't *need* to save our object in $obj at this point, but we tend to do that because later we'll be modifying the object a bit, so it's useful to have it in a variable.
- We explicitly output the object *immediately* to the pipeline, using `Write-Output`, rather than accumulating it in an array or something to output later. The whole point of the pipeline is to accumulate objects for us, and pass them on to whatever's next in the pipeline. Technically, you don't need to use `Write-Output`. If you had simply used `$obj`, the result would have been the same.

A Quick Test

Running the code snippet for the local machine we get:

```
FreeMem      : 15050452
Uptime       : 4.20:30:00.2480806
CPULoad      : 8
Processes    : 304
PctFreeMem   : 45.0038011371357
TotalMem     : 33442624
Computername : BOVINE320
PctFreeC     : 36.5086482029627
```

Notice that *these properties aren't in the right order!* That's because we used a normal hash table to construct our property list, and .NET memory-optimizes that storage, which can result in a re-ordering. *That's fine.* At this level of a tool, we shouldn't be worried about what the output looks like - we could always use a `Format` command, or `Select-Object`, to specify an order. It *is* possible to construct an `[ordered]` hash table instead, but we rarely do so. You'll see another trick later in the book.

By the way, we deliberately left the memory values in the native format, because it'll be useful for showing you another trick later.

Here's our revised code:

Revised Get-TMComputerStatus

```
Function Get-TMComputerStatus {
    [cmdletbinding()]
    Param(
        [Parameter(ValueFromPipeline = $True, Mandatory = $True)]
        [ValidateNotNullorEmpty()]
        [ValidatePattern("^\w+$")]
        [Alias("CN", "Machine", "Name")]
        [string[]]$Computername,
        [string]$ErrorLogFilePath,
        [switch]$ErrorAppend
    )

    BEGIN {}

    PROCESS {
        foreach ($computer in $Computername) {
            $params = @{
```

```
                Classname    = "Win32_OperatingSystem"
                Computername = $computer
            }
            $OS = Get-CimInstance @params

            $params.ClassName = "Win32_Processor"
            $cpu = Get-CimInstance @params

            $params.className = "Win32_logicalDisk"
            $vol = Get-CimInstance @params -filter "DeviceID='c:'"

            $props = @{
                Computername = $os.CSName
                TotalMem     = $os.TotalVisibleMemorySize
                FreeMem      = $os.FreePhysicalMemory
                Processes    = $os.NumberOfProcesses
                PctFreeMem   = ($os.FreePhysicalMemory/$os.TotalVisibleMemorySize)*1\
00
                Uptime       = (Get-Date) - $os.lastBootUpTime
                CPULoad      = $cpu.LoadPercentage
                PctFreeC     = ($vol.FreeSpace/$vol.size)*100
            }
            $obj = New-Object -TypeName PSObject -Property $props
            Write-Output $obj

        } #foreach $computer
    }
    END {}
} #Get-TMComputerStatus
```

Keep in mind that this is also in the code samples that we've mentioned previously.

Your Turn

Let's turn back to the task at creating getting remote listening configuration. The current output is limited to `Select-Object`. But that's about to change!

Start Here

Here's where we left off with our version of this function:

```
Function Get-TMRemoteListeningConfiguration {
    [cmdletbinding()]
    Param(
    [Parameter(ValueFromPipeline = $True, Mandatory = $True)]
    [ValidateNotNullorEmpty()]
    [Alias("CN")]
    [string[]]$Computername,
    [string]$ErrorLog
    )

    Begin {
      #not used
    }
    Process {
    $ports = 22,5985,5986
    foreach ($computer in $computername) {
        foreach ($port in $ports) {
            Test-NetConnection -Port $port -ComputerName $Computer |
            Select-Object Computername,RemotePort,TCPTestSucceeded
        }
        #TODO
        #better output
        #error handling and logging
    } #foreach
   }
   End {
    #not used
   }
} #Get-RemoteListeningConfiguration function
```

Use that, or your own work from the previous chapter, as a starting point.

Your Task

Modify your function so that it outputs an object for each computer it operates against. The output should include the computer name, the remote IP address that responded, and the date of the test. The primary purpose for this function is to determine what WSMan and SSH ports and/or protocols are enabled. You can assume the standard ports.

Don't be afraid to restructure or revise the code. Often we start going down one path only to realize we really need to go down a different one.

Our Take

Here's our version (remember, you can get the actual code file in the downloadable samples).

Get-TMRemoteListeningConfiguration

```
Function Get-TMRemoteListeningConfiguration {
  [cmdletbinding()]
  Param(
    [Parameter(ValueFromPipeline = $True, Mandatory = $True)]
    [ValidateNotNullorEmpty()]
    [Alias("CN")]
    [string[]]$Computername,
    [string]$ErrorLog
  )

  Begin {
    #define a hashtable of ports
    $ports = @{
      WSManHTTP  = 5985
      WSManHTTPS = 5986
      SSH        = 22
    }

    #initialize an splatting hashtable
    $testParams = @{
      Port         = ""
      Computername = ""
    }
  } #begin
  Process {

    foreach ($computer in $computername) {
      $testParams.Computername = $computer

      #define the hashtable of properties for
      #the custom object
      $props = @{
        Computername = $computer
        Date         = Get-Date
      }

      #enumerate the hashtable
      $ports.GetEnumerator() | ForEach-Object {
```

```
        $testParams.Port = $_.Value
        $test = Test-NetConnection @testParams

        #add results
        $props.Add($_.name, $test.TCPTestSucceeded)

        #assume the same remote address will respond to all
        #requests
        if (-NOT $props.ContainsKey("RemoteAddress")) {
          $props.Add("RemoteAddress", $test.RemoteAddress)
        }
      }

      #create the custom object
      $obj = New-Object -TypeName PSObject -Property $props
      Write-Output $obj

      #TODO
      #error handling and logging
    } #foreach
  } #process
  End {
    #not used
  } #end
} #Get-RemoteListeningConfiguration function
```

Notice a couple of things:

- We are now using the `Begin‘` block to initialize a few hashtables. This only needs to be done once before any computernames are processed. The names on the ports hashtable will eventually be used as property names.
- We setup a hashtable of parameters to splat to `Test-NetConnection`. There are a few ways to handle this hashtable and purists might take offense. But it works and is pretty easy to understand.
- We’re assuming the same IP address will respond to all requests so that value only needs to be added once to the property hashtable.

Let’s Review

Here are a few questions you can use to make sure you’ve picked up the key points of this chapter:

1. What is splatting?

2. Why is it important to output objects from a tool?
3. Why might you save a new object in a variable instead of outputting it right away?
4. Why should you not accumulate objects in an array before outputting them all at once?

Review Answers

Here are our answers:

1. Splatting is a way of assembling command parameters into a hash table and feeding them to the command in one unit.
2. Objects are structured data, and can be consumed by many other commands. Text is not structured, and cannot easily be consumed by anything other than human eyeballs.
3. You will often want to perform additional actions on, or modifications to, the object, and having the object in a variable facilitates that.
4. Accumulating will make your command "block" the pipeline; outputting objects one at a time allows the pipeline to run multiple commands in parallel.

An Interlude: Changing Your Approach

Let's take a quick break from the narrative. In the preceding chapters, we've focused a lot on building tools that conform to PowerShell's native patterns and practices. That's all well and good, but sometimes you can make a point truly hit home by showing its opposite.

Consider this forums post from PowerShell.org[15], which we've referenced with permission from its original author. He posted this code which we've reformatted slightly to fit the page:

```
$UserNames = Get-ADUser -Filter * -SearchBase `
"OU=NAME_OF_OU_WITH_USERS3,OU=NAME_OF_OU_WITH_USERS2,
OU=NAME_OF_OU_WITH_USERS1,DC=DOMAIN_NAME,DC=COUNTRY_CODE"
Select -ExpandProperty samaccountname

$UserRegex = ($UserNames | ForEach{[RegEx]::Escape($_)}) -join "|"

$myArray = (Get-ChildItem -Path "\\file2\Felles\Home\*" -Directory |
Where {$_.Name -notmatch $UserRegex})

foreach ($mapper in $myArray) {
    #Param ($mapper = $(Throw "no folder name specified"))

    # calculate folder size and recurse as needed
    $size = 0
    Foreach ($file in $(ls $mapper -recurse)) {
    If (-not ($file.psiscontainer)) {
        $size += $file.length
        }
    }

    # return the value and go back to caller
    echo $size
}
```

The goal is to list the sizes of each user's home folder, and to show any "orphan" folders - that is, folders which no longer correspond to an Active Directory user.

[15] https://powershell.org/forums/topic/compare-home-folders-with-user-names-and-fetch-folder-size/#post-61401

The Critique

Now, this isn't in any way meant as a beat-up on the original author. People learn different things at different times, and arrive at their code through a variety of paths. Let's just take the code for what it is.

- If we were asked to solve this problem, we'd write this as two functions, not as one script. One function would sum-up folder sizes, which is a totally useful function in a lot of scenarios. Another would figure out which folders were orphans.
- We'd also take a more PowerShell-native approach, avoiding things like `echo`. Instead, we'd have a goal of outputting objects, since those could be piped to commands that made them into CSV files, HTML reports, and lots more. Although on most systems `echo` should be an alias for `Write-Output` which means objects will be written to the pipeline. But using this alias doesn't make that clear, and someone could have used `echo` as an alias for `Write-Host` and then you would be back to not having objects in the pipeline.
- We'd probably make more use of native PowerShell commands, since they tend to run a smidge faster than a script.
- Finally, we'd try to keep our functions as generic and non-context-specific as possible, to maximize re-use. This means no hard-coded names or paths.

One thing to remember is that, in Windows, *folders don't have a size*. We have to instead get all the files within that folder, and add up *their* sizes.

Our Take

Here's our first function. Notice that, if a folder doesn't exist, we're explicitly outputting an "empty" object.

Get-FolderSize

```
function Get-FolderSize {
[CmdletBinding()]
Param(
    [Parameter(
        Mandatory,
        ValueFromPipeline,
        ValueFromPipelineByPropertyName
    )]
    [string[]]$Path
)
BEGIN {}
PROCESS {
```

```
    ForEach ($folder in $path) {
        Write-Verbose "Checking $folder"
        if (Test-Path -Path $folder) {
            Write-Verbose " + Path exists"

            #turn the folder into a true FileSystem path
            $cPath = Convert-Path $Folder

            $params = @{
                Path    = $cPath
                Recurse = $true
                File    = $true
            }
            $measure = Get-ChildItem @params |
            Measure-Object -Property Length -Sum
            [pscustomobject]@{
                Path  = $cPath
                Files = $measure.count
                Bytes = $measure.sum
            }
        }
        else {
            Write-Verbose " - Path does not exist"
            [pscustomobject]@{
                Path  = $folder
                Files = 0
                Bytes = 0
            }
        } #if folder exists
    } #foreach
} #PROCESS
END {}
} #function
```

That's our first function, and the results end up looking like this:

```
Path                                          Files    Bytes
----                                          -----    -----
Z:\Documents\GitHub\ToolmakingBook\code          35    44101
Z:\Documents\GitHub\ToolmakingBook\manuscript    55 63679159
Z:\nope                                           0        0
```

Obviously, we could pipe that to `Select-Object` to turn the Byte count into another unit, like

megabytes, but we feel it's important for our tool to output the lowest-level of information possible, to maximize its utility. Notice that we didn't test this against "home folders" per se; we want this to be a generic folder-size-adding-up function. Later, we'll write a controller script to put this function to a more specific business use, like summing up user home folder sizes.

Now we're going to write a second function to deal with orphan folders. This will actually incorporate our `Get-FolderSize` function. This tool is a bit more task-specific, because it needs to understand our specific need to identify orphaned home folders.

Get-UserHomeFolderInfo

```
function Get-UserHomeFolderInfo {
[CmdletBinding()]
Param(
    [Parameter(Mandatory)]
    [string]$HomeRootPath
)
BEGIN {}
PROCESS {
    Write-Verbose "Enumerating $HomeRootPath"
    $params = @{
        Path      = $HomeRootPath
        Directory = $True
    }
    ForEach ($folder in (Get-ChildItem @params)) {

        Write-Verbose "Checking $($folder.name)"
        $params = @{
            Identity    = $folder.name
            ErrorAction = 'SilentlyContinue'
        }
        $user = Get-ADUser @params

        if ($user) {
            Write-Verbose " + User exists"
            $result = Get-FolderSize -Path $folder.fullname
            [pscustomobject]@{
                User   = $folder.name
                Path   = $folder.fullname
                Files  = $result.files
                Bytes  = $result.bytes
                Status = 'OK'
            }
        }
        else {
```

```
            Write-Verbose " - User does not exist"
            [pscustomobject]@{
                User   = $folder.name
                Path   = $folder.fullname
                Files  = 0
                Bytes  = 0
                Status = 'Orphan'
            }
            } #if user exists

        } #foreach
    } #PROCESS
    END {}
}
```

Here, we're taking a root location which contains home folders. We go through them one at a time, and check to see that a corresponding Active Directory user exists. If one doesn't, we output a "blank" object with an "Orphan" status property. We could easily use `Where-Object` to filter for just the orphans, so that someone can deal with those. If the user does exist, we use `Get-FolderSize` to get the size info, and output the same kind of object. This time, the object is fully populated, with an "OK" status.

The idea of writing out the same kind of object, either way, ensures consistent output, and maximizes the reusability of the information. You'll find this code in the downloadable samples, under this chapter's folder.

Our functions are far from a complete and production worthy pieces of code. There's no error handling or parameter validation among other things. They were written to illustrate concepts and patterns. You are welcome to finish these functions off to meet your own production requirements.

Summary

The idea here is to take the task and really break it down. In the original forums post, the source data was "all users in AD," which created some challenges in finding orphan folders. In our approach, we've used the actual list of folders as the source data, and checked each one against Active Directory. That won't tell us if we have users *without* home folders, but that wasn't a stated problem. In reality, we'd expect the users themselves would bring it up to the help desk if they didn't have a home folder.

We took the one "generic" portion of our task and wrote it out as its own tool - `Get-FolderSize`. We made sure it was useful on its own, accepting pipeline input and such, even though that's not how

`Get-UserHomeFolderInfo` actually uses it. We incorporated Verbose output that will make each one a bit easier to follow and debug, if necessary. And, because we've used functions, each task is tightly scoped and does just one thing, making each function less complex, easier to debug, and easier to understand and maintain.

Using Verbose, Warning, and Informational Output

A couple of chapters ago, we pointed out that adding `[CmdletBinding()]` to our `Param` block would "enable" the output of certain commands for verbose, warning, and informational. Well, it's time to put that to use.

Knowing the Six Channels

It's useful to understand that PowerShell has six "channels," or pipelines, rather than the one that we normally think of.

First up is the *Success* pipeline, which is the one you're used to thinking of as just "the pipeline." This gets some special treatment from the PowerShell engine. For example, it's the pipeline used to pass objects from command to command. Additionally, at the end of the pipeline, PowerShell sort of invisibly adds the `Out-Default` cmdlet, which has the effect of running any objects in the pipeline through PowerShell's formatting system. Whatever hosting application you're using - the PowerShell console, ISE,VS Code, etc. - is responsible for dealing with that output by placing it onto the screen or something.

But there are five other pipelines:

1. Success - which we discussed.
2. Errors
3. Warnings
4. Verbose
5. Debug
6. Informational

Those numbers actually correspond with how PowerShell references each pipeline for redirection purposes[16].

Each pipeline represents a discrete and independent way of passing information. Each hosting application decided how to deal with each pipeline. For example, the console host displays items from pipeline `#4` (Verbose) in yellow text, prefixed by "VERBOSE: ". Other hosts might log that output to an event log or ignore it completely.

[16] https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.core/about/about_redirection

Additionally, the shell defines several "preference" variables that control the output of each pipeline. `$VerbosePreference` controls pipeline #4, `$WarningPreference` controls #3, and so on. Setting a preference to "SilentlyContinue" will suppress that pipeline's output; setting it to "Continue" will display the output in whatever way the host application defines. The common parameters we described in a previous chapter override the preference variables on a per-command basis. For example, adding `-Verbose` to your command, when you run it, will enable `Write-Verbose` output in your command.

Adding Verbose and Warning Output

Verbose output is disabled by default; Warning output is enabled. With that in mind, we tend to do something like the following with those two forms of output:

Get-TMComputerStatus

```
Function Get-TMComputerStatus {
[cmdletbinding()]
Param(
    [Parameter(ValueFromPipeline = $True, Mandatory = $True)]
    [ValidateNotNullorEmpty()]
    [ValidatePattern("^\w+$")]
    [Alias("CN", "Machine", "Name")]
    [string[]]$Computername,
    [string]$ErrorLogFilePath,
    [switch]$ErrorAppend
)

BEGIN {
    Write-Verbose "Starting $($myinvocation.mycommand)"
}

PROCESS {
    foreach ($computer in $Computername) {
     Write-Verbose "Querying $computer"
     $params = @{
        Classname    = "Win32_OperatingSystem"
        Computername = $computer
     }

     $OS = Get-CimInstance @params

     $params.ClassName = "Win32_Processor"
     $cpu = Get-CimInstance @params
```

```
        $params.className = "Win32_logicalDisk"
        $vol = Get-CimInstance @params -filter "DeviceID='c:'"

        [pscustomobject]@{
           Computername = $os.CSName
           TotalMem     = $os.TotalVisibleMemorySize
           FreeMem      = $os.FreePhysicalMemory
           Processes    = $os.NumberOfProcesses
           PctFreeMem   = ($os.FreePhysicalMemory/$os.TotalVisibleMemorySize)*100
           Uptime       = (Get-Date) - $os.lastBootUpTime
           CPULoad      = $cpu.LoadPercentage
           PctFreeC     = ($vol.FreeSpace/$vol.size)*100
        }
      } #foreach $computer
}
END {
        Write-Verbose "Ending $($myinvocation.mycommand)"
}
} #Get-TMComputerStatus
```

Sharp-eyed readers will notice two things:

- We snuck in a change to the `New-Object` creation. This is mainly just to show you a new technique that you may run across. Rather than defining a hash table of properties and passing it to `New-Object`, we've used the `[pscustomobject]` type accelerator to do the same job in a bit less space.
- We've replaced a lot of our inline comments with verbose output. This lets the same message be seen by someone *running* the code, provided they add `-Verbose` when doing so.

When you run your command with `-Verbose` any cmdlets that you call which support `-Verbose` will also use it.

```
PS C:\> Get-TMComputerStatus -Computername thinkp1 -Verbose
VERBOSE: Starting Get-TMComputerStatus
VERBOSE: Querying thinkp1
VERBOSE: Perform operation 'Enumerate CimInstances' with following parameters,
''namespaceName' = root\cimv2,'className' = Win32_OperatingSystem'.
VERBOSE: Operation 'Enumerate CimInstances' complete.
VERBOSE: Perform operation 'Enumerate CimInstances' with following parameters,
''namespaceName' = root\cimv2,'className' = Win32_Processor'.
VERBOSE: Operation 'Enumerate CimInstances' complete.
VERBOSE: Perform operation 'Query CimInstances' with following parameters,
''queryExpression' = SELECT * FROM Win32_logicalDisk WHERE DeviceID='c:',
'queryDialect' = WQL,'namespaceName' = root\cimv2'.
VERBOSE: Operation 'Query CimInstances' complete.


Processes    : 220
Computername : THINKP1
FreeMem      : 27192868
PctFreeMem   : 82.2032500565296
PctFreeC     : 54.8739635724224
Uptime       : 1.03:13:05.0375303
TotalMem     : 33080040
CPULoad      : 1

VERBOSE: Ending Get-TMComputerStatus
```

We also added verbose output in the `Begin` and `End` scriptblocks to reflect starting and ending a command. Instead of hard-coding your command name, let PowerShell detect it from the automatic variable `$myinvocation`. This is a very handy technique when you have functions calling other functions and you are trying to follow the flow of execution.

We haven't added any warning output yet, because we haven't a need for it. But we will, eventually - so keep `Write-Warning` in the back of your brain. Eventually we'll add statements like this:

```
Write-Warning "Danger, Will Robinson!"
```

Doing More With Verbose

If you take a moment to think about it, you'll realize that incorporating `Write-Verbose` statements into your tool makes a lot of sense. In fact, we recommend you include the statements from the very beginning. Don't wait to add them until after you have finished scripting. Add them first! Insert

verbose messages throughout your script that highlight what action your command is performing, or the value of key variables. This will help you troubleshoot and debug during the development process because you can run your command with `-Verbose`. The verbose messages can also double as internal documentation. Finally, if someone is trying to run your tool and encountering problems, you can have them start a transcript, run your command with `-Verbose`, then close the transcript and send it to you. If you've written good verbose messages you'll be able to track what is happening and hopefully identify the problem.

In fact, you might consider adding Verbose messages like this at the beginning of your command.

```
Write-Verbose "Execution Metadata:"
Write-Verbose "User = $($env:userdomain)\$($env:USERNAME)"
$id = [System.Security.Principal.WindowsIdentity]::GetCurrent()
$IsAdmin = [System.Security.Principal.WindowsPrincipal]::new($id).IsInRole(
'administrators')
Write-Verbose "Is Admin = $IsAdmin"
Write-Verbose "Computername = $env:COMPUTERNAME"
Write-Verbose "OS = $((Get-CimInstance Win32_Operatingsystem).Caption)"
Write-Verbose "Host = $($host.Name)"
Write-Verbose "PSVersion = $($PSVersionTable.PSVersion)"
Write-Verbose "Runtime = $(Get-Date)"
```

When executed you'll get potentially useful information like this:

```
VERBOSE: Execution Metadata:
VERBOSE: User = BOVINE320\Jeff
VERBOSE: Is Admin = False
VERBOSE: Computername = BOVINE320
VERBOSE: Perform operation 'Enumerate CimInstances' with following
parameters, ''namespaceName' = root\cimv2,'className' =
Win32_Operatingsystem'.
VERBOSE: Operation 'Enumerate CimInstances' complete.
VERBOSE: OS = Microsoft Windows 10 Pro
VERBOSE: Host = Windows PowerShell ISE Host
VERBOSE: PSVersion = 5.1.18362.752
VERBOSE: Runtime = 07/03/2020 15:05:50
```

Another tip is to add a prefix to each verbose message that indicates what scriptblock is being called.

Get-Foo

```
Function Get-Foo {
    [cmdletbinding()]
    Param(
        [string]$Computername
    )

    Begin {
        Write-Verbose "[BEGIN  ] Starting: $($MyInvocation.Mycommand)"
        Write-Verbose "[BEGIN  ] Initializing array"
        $a = @()

    } #begin

    Process {
        Write-Verbose "[PROCESS] Processing: $Computername"
        # code goes here
    } #process

    End {
        Write-Verbose "[END    ] Ending: $($MyInvocation.Mycommand)"
    } #end

} #end Get-Foo
```

See how there is sort of a “block comment” effect? This makes it easier to know exactly where your command is. Note the use of padded spaces. This is to make the verbose output easy to read.

```
PS C:\> Get-Foo -Computername FOO -Verbose
VERBOSE: [BEGIN  ] Starting: Get-Foo
VERBOSE: [BEGIN  ] Initializing array
VERBOSE: [PROCESS] Processing: FOO
VERBOSE: [END    ] Ending: Get-Foo
```

A variation you might consider is including a timestamp. This is especially useful for long running commands.

Get-Bar

```
Function Get-Bar {
    [cmdletbinding()]
    Param(
        [string]$Computername
    )

    Begin {
        Write-Verbose "[$((Get-Date).TimeOfDay) BEGIN  ] Starting: $($MyInvocation.M\
ycommand)"
        Write-Verbose "[$((Get-Date).TimeOfDay) BEGIN  ] Initializing array"
        $a = @()

    } #begin

    Process {
        Write-Verbose "[$((Get-Date).TimeOfDay) PROCESS] Processing: $Computername"
        # code goes here
    } #process

    End {
        Write-Verbose "[$((Get-Date).TimeOfDay) END    ] Ending: $($MyInvocation.Myc\
ommand)"

    } #end

} #end Get-Bar
```

Which will give you verbose output like this:

```
PS C:\> Get-Bar -Computername foo -Verbose
VERBOSE: [14:01:51.8020007 BEGIN  ] Starting: Get-Bar
VERBOSE: [14:01:51.8140051 BEGIN  ] Initializing array
VERBOSE: [14:01:51.8170015 PROCESS] Processing: foo
VERBOSE: [14:01:51.8190054 END    ] Ending: Get-Bar
```

There's really no limit to how you can use Verbose messages. It is up to you to decide what would be useful information.

Informational Output

This new, sixth channel was introduced in PowerShell v5. In fact, PowerShell v5 more or less did away with its original `Write-Host` cmdlet, and turned `Write-Host` into a wrapper around `Write-Information`. The Information stream is a bit different from other pipelines that can carry messages, because it's designed to carry *structured* messages. It requires a bit of pre-planning to use well. However, there's still an `$InformationPreference` variable which can suppress or allow the output of this stream, and it's set to SilentlyContinue, or "off," by default. When you run a command, you can specify `-InformationAction Continue` to enable that command's Informational output.

$InformationPreference and `-InformationAction` are automatically set to "Continue" when you use `Write-Host`, so that `Write-Host` behaves as it did in previous versions of PowerShell.

It's worth noting that Informational output works in PowerShell jobs, scheduled jobs, and workflows, which isn't the case with most of the other forms of messaging - Verbose, Warning, etc.

On a basic level, using `Write-Information` isn't any different than using `Write-Verbose`. The `-MessageData` parameter is in the first position, so you'll often skip using the parameter name and just add whatever message you want to include - same as we just did with `Write-Verbose`. But messages can also be *tagged*, usually with a keyword like "information," "instructions," or whatever you decide. The information stream can then be *searched* based on those tags. You can also run commands using the `-InformationVariable` parameter to have informational messages added to a variable that you designate. This can help keep the information messages from "cluttering up" your normal output.

So, for example:

Get-Example

```
Function Get-Example {

    [CmdletBinding()]
    Param()

    Write-Information "First message" -tag status
    Write-Information "Note that this had no parameters" -tag notice
    Write-Information "Second message" -tag status

}

Get-Example -InformationAction Continue -InformationVariable x
```

Using "Continue" the way we did there makes it apply to all `Write-Information` commands *inside* the `Example` function. And if you run that (in PowerShell 5 or later), you'll see that the informational

messages do indeed appear. Were you to examine $x, you'd find the messages in it, as well. Contrast the above with this:

Get-Example with variable output

```
function Get-Example {
    [CmdletBinding()]
    Param()

    Write-Information "First message" -tag status
    Write-Information "Note that this had no parameters" -tag notice
    Write-Information "Second message" -tag status
}

Example -InformationAction SilentlyContinue -IV x
$x
```

This time, our messages don't appear, because we used "SilentlyContinue." However, *the commands still run and still work,* and if you were to examine $x you'd find all three messages in there. Notice that we shortened `-InformationVariable` to its `-IV` alias to save some room. Let's now go one step further:

Get-Example with filtered output

```
function Get-Example {
    [CmdletBinding()]
    Param()
    Write-Information "First message" -tag status
    Write-Information "Note that this had no parameters" -tag notice
    Write-Information "Second message" -tag status

}

Get-Example -InformationAction SilentlyContinue -IV x

$x | Where-Object tags -in @('notice')
```

In this example, only our second message, "Note that this had no parameters", will display, because we filtered that out of $x by using the Tags property of the messages.

A Detailed Information Example

Like Verbose output, effectively using the Information channel requires some planning on your part. **You** have to figure out what needs to be logged and how it might be used and you need to implement

your `Write-Information` commands when creating your tool. Here's a very simple function we can use to illustrate how you might use `Write-Information`. You can find a file with these test functions in the code folder for this chapter.

Test-Me

```
Function Test-Me {
[cmdletbinding()]
Param()

Write-Information "Starting $($MyInvocation.MyCommand) " -Tags Process
Write-Information "PSVersion = $($PSVersionTable.PSVersion)" -Tags Meta
Write-Information "OS = $((Get-CimInstance Win32_operatingsystem).Caption)" `
 -Tags Meta

Write-Verbose "Getting top 5 processes by WorkingSet"
Get-Process | Sort-Object WS -Descending |
Select-Object -first 5 -OutVariable s

Write-Information ($s[0] | Out-String) -Tags Data

Write-Information "Ending $($MyInvocation.MyCommand) " -Tags Process
}
```

Running the command normally will give you the top 5 processes by working set. Now run it like this:

```
PS C:\> test-me -InformationAction Continue
Starting Test-Me
PSVersion = 5.1.18362.752
OS = Microsoft Windows 10 Pro

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
      0       0     3860    1597352     382.23    3272   0 Memory Compression
    273      45  1790192    1307884   1,376.14    4208   1 TabNine
   1694     187   739776     618188     438.19   24200   1 firefox
   2111     182   539468     516704   1,946.97     976   1 firefox
   1906      67   844212     362408   7,494.30    1460   1 dwm

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
      0       0     3860    1597352     382.23    3272   0 Memory Compression
```

```
Ending Test-Me
```

By setting the common parameter, `-InformationAction` to `Continue` this "turned on" the information channel which also displays the information. This can be useful when building your messages and you want to see how what they will do.

Next, run the command using the `-InformationVariable` parameter.

```
PS C:\> test-me -InformationVariable inf
```

This time you won't get the information messages, because the command is running with the default "SilentlyContinue" setting for information messages, suppressing them. Instead, they will be directed to the variable `inf`.

```
PS C:\> $inf
Starting Test-Me
PSVersion = 5.1.18362.752
OS = Microsoft Windows 10 Pro

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)     Id  SI ProcessName
-------  ------    -----      -----     ------     --  -- -----------
      0       0     3860    1596064     382.27   3272   0 Memory Compression


Ending Test-Me
```

What you get back is a very rich object:

```
PS C:\> $inf | Get-Member


   TypeName: System.Management.Automation.InformationRecord

Name            MemberType Definition
----            ---------- ----------
Equals          Method     bool Equals(System.Object obj)
GetHashCode     Method     int GetHashCode()
GetType         Method     type GetType()
ToString        Method     string ToString()
Computer        Property   string Computer {get;set;}
ManagedThreadId Property   uint32 ManagedThreadId {get;set;}
```

```
MessageData     Property   System.Object MessageData {get;}
NativeThreadId  Property   uint32 NativeThreadId {get;set;}
ProcessId       Property   uint32 ProcessId {get;set;}
Source          Property   string Source {get;set;}
Tags            Property   System.Collections.Generic.List[string] Tags...
TimeGenerated   Property   datetime TimeGenerated {get;set;}
User            Property   string User {get;set;}
```

Which means you can work with the data however you'd like.

```
PS C:\> $inf.where({$_.tags -contains 'meta'}) | select Computer,MessageData

Computer     MessageData
--------     -----------
BOVINE320      PSVersion =  5.1.18362.752
BOVINE320      OS = Microsoft Windows 10 Pro
```

The key takeaway is that if your command doesn't have any `Write-Information` commands, then the information parameters are irrelevant.

However, as we mentioned earlier, beginning with PowerShell 5.05, `Write-Host` was re-factored to be a conduit for `Write-Information`. Check this revised version of the function.

Test-Me2

```
Function Test-Me2 {
[cmdletbinding()]
Param()

Write-Host "Starting $($MyInvocation.MyCommand) " -fore green
Write-Host "PSVersion = $($PSVersionTable.PSVersion)" -fore green
Write-Host "OS = $((Get-CimInstance Win32_operatingsystem).Caption)" `
-fore green

Write-Verbose "Getting top 5 processes by WorkingSet"
Get-Process | Sort-Object -property WS -Descending |
Select-Object -first 5 -OutVariable s

Write-Host ($s[0] | Out-String) -fore green

Write-Host "Ending $($MyInvocation.MyCommand) " -fore green
}
```

One benefit of using `Write-Host` is the ability to colorize the output. Unfortunately even if you run the command like this:

```
test-me2 -InformationVariable inf2
```

You will get the information output saved to $inf2. But you'll also get the informational messages written to the host in green. This may not be desirable. This technique also loses the ability to add tags.

Here's one final version that is more a proof of concept than anything.

Test-Me3

```
Function Test-Me3 {
[cmdletbinding()]
Param()

if ($PSBoundParameters.ContainsKey("InformationVariable")) {
  $Info = $True
  $infVar = $PSBoundParameters["InformationVariable"]
}

if ($info) {
  Write-Host "Starting $($MyInvocation.MyCommand) " -fore green

  (Get-Variable $infVar).value[-1].Tags.Add("Process")
  Write-Host "PSVersion = $($PSVersionTable.PSVersion)" -fore green
  (Get-Variable $infVar).value[-1].Tags.Add("Meta")
  Write-Host "OS = $((Get-CimInstance Win32_operatingsystem).Caption)" `
  -fore green
  (Get-Variable $infVar).value[-1].Tags.Add("Meta")
}
Write-Verbose "Getting top 5 processes by WorkingSet"
Get-Process | Sort-Object WS -Descending |
Select-Object -first 5 -OutVariable s

if ($info) {
  Write-Host ($s[0] | Out-String) -fore green
  (Get-Variable $infVar).value[-1].Tags.Add("Data")
  Write-Host "Ending $($MyInvocation.MyCommand) " -fore green
  (Get-Variable $infVar).value[-1].Tags.Add("Process")
  }
}
```

This function tests to see if `-InformationVariable` was specified and if so a variable ($Info) is switch on. When information is needed via `Write-Host`, if $Info is true, then the `Write-Host` lines are called. Immediately after each line a tag is added to the information variable.

```
test-me3 -InformationVariable inf3
```

This will display the information messages in green and generate the information variable.

```
PS C:\> $inf3 | Group-object {$_.tags -join "-"}

Count Name                      Group
----- ----                      -----
    2 PSHOST-Process            {Starting Test-Me3 , Ending Test-Me3 }
    2 PSHOST-Meta               {PSVersion = 5.1.18362.752, OS = Mi...}
    1 PSHOST-Data               {...
```

Before we move on, don't forget that the information variables are just another type of object. You could export the variable using `Export-Clixml`, store the results in a database, or create a custom text log file from the different properties.

Verbose output is still a good choice when you're using PowerShell versions prior to 5. Once you're using 5, however, it may make sense to start migrating to Information messages instead, given their flexibility, tags, and searchability. For now, because we're aiming for greater compatibility, we're sticking with verbose output in our example.

Your Turn

As you might imagine, you're going to add some verbose output to the tool designed to get remote listening configurations.

Start Here

Here's where we left off after the last round of revisions. You can start here (or use our code sample from the download), or start with your result from the earlier chapter.

```
Function Get-TMRemoteListeningConfiguration {
  [cmdletbinding()]
  Param(
    [Parameter(ValueFromPipeline = $True, Mandatory = $True)]
    [ValidateNotNullorEmpty()]
    [Alias("CN")]
    [string[]]$Computername,
    [string]$ErrorLog
  )
```

```
Begin {
  #define a hashtable of ports
  $ports = @{
    WSManHTTP  = 5985
    WSManHTTPS = 5986
    SSH        = 22
  }

  #initialize an splatting hashtable
  $testParams = @{
    Port         = ""
    Computername = ""
  }
} #begin
Process {

  foreach ($computer in $computername) {
    $testParams.Computername = $computer

    #define the hashtable of properties for
    #the custom object
    $props = @{
      Computername = $computer
      Date         = Get-Date
    }

    #enumerate the hashtable
    $ports.GetEnumerator() | ForEach-Object {
      $testParams.Port = $_.Value
      $test = Test-NetConnection @testParams

      #add results
      $props.Add($_.name, $test.TCPTestSucceeded)

      #assume the same remote address will respond to all
      #requests
      if (-NOT $props.ContainsKey("RemoteAddress")) {
        $props.Add("RemoteAddress", $test.RemoteAddress)
      }
    }

    #create the custom object
    $obj = New-Object -TypeName PSObject -Property $props
```

```
      Write-Output $obj

      #TODO
      #error handling and logging
    } #foreach
  } #process
  End {
    #not used
  } #end
} #Get-RemoteListeningConfiguration function
```

Your Task

Add some meaningful verbose output to your tool. If you see an opportunity to add warning output, feel free to add that as well.

Our Take

Here's what we came up with:

Get-TMRemoteListeningConfiguration

```
Function Get-TMRemoteListeningConfiguration {
  [cmdletbinding()]
  Param(
    [Parameter(ValueFromPipeline,Mandatory)]
    [ValidateNotNullorEmpty()]
    [Alias("CN")]
    [string[]]$Computername,
    [string]$ErrorLog
  )

  Begin {
   Write-Information "Command = $($myinvocation.mycommand)" -Tags Meta
   Write-Information "PSVersion = $($PSVersionTable.PSVersion)" -Tags Meta
   Write-Information "User = $env:userdomain\$env:username" -tags Meta
   Write-Information "Computer = $env:computername" -tags Meta
   Write-Information "PSHost = $($host.name)" -Tags Meta
   Write-Information "Test Date = $(Get-Date)" -tags Meta

    #define a private function to write the verbose messages
    Function WV {
    Param($prefix,$message)
```

```
        $time = Get-Date -f HH:mm:ss.ffff
        Write-Verbose "$time [$($prefix.padright(7,' '))] $message"
    }

    WV -prefix BEGIN -message "Starting $($myinvocation.MyCommand)"

    #define a ordered hashtable of ports so that the testing
    #goes in the same order
    $ports = [ordered]@{
      WSManHTTP  = 5985
      WSManHTTPS = 5986
      SSH        = 22
    }

    #initialize an splatting hashtable
    $testParams = @{
      Port         = ""
      Computername = ""
      WarningAction = "SilentlyContinue"
      WarningVariable = "wv"
    }
    #keep track of total computers tested
    $total=0
    #keep track of how long testing takes
    $begin = Get-Date
  } #begin
  Process {
    foreach ($computer in $computername) {
      $total++
      #make the computername all upper case
      $testParams.Computername = $computer.ToUpper()

      WV PROCESS "Testing $($testParams.Computername)"

      #define the hashtable of properties for the custom object
      $props = [ordered]@{
        Computername = $testparams.Computername
        Date         = Get-Date
      }

      #this array will be used to store passed ports
      #It is used by Write-Information
      $passed = @()
```

```
      #enumerate the hashtable
      $ports.GetEnumerator() | ForEach-Object {
        $testParams.Port = $_.Value

        WV "PROCESS" "Testing port $($testparams.port)"
        $test = Test-NetConnection @testParams

        WV PROCESS "Adding results"
        $props.Add($_.name, $test.TCPTestSucceeded)
        if ($test.TCPTestSucceeded) {
            $passed+=$testParams.Port
        }

        if (-NOT $props.Contains("RemoteAddress")) {
           wv "PROCESS" "Adding RemoteAddress $($test.remoteAddress)"
          $props.Add("RemoteAddress", $test.RemoteAddress)
        }
      }

      Write-Information "$($testParams.Computername) = $($passed -join ',')" `
      -Tags data

      $obj = New-Object -TypeName PSObject -Property $props
      Write-Output $obj

      #TODO: error handling and logging
    } #foreach
  } #process
  End {
    $runtime = New-TimeSpan -Start $begin -End (Get-Date)
    WV END "Processed $total computer(s) in $runtime"
    WV END "Ending $($myinvocation.mycommand)"
  } #end

} #Get-TMRemoteListeningConfiguration function
```

Here are a few highlights of the changes we made.

- We wanted to use `Write-Verbose` throughout. But we also wanted to standardize on the verbose message and reduce the amount of typing. In the `Begin` block we created a private "helper" function called `wv`. Because the command is never exposed we are ok using a non-standard

name. The function handles writing the verbose message. If we decide later we want a different time stamp format or other information, all we have to do is modify the function.
- We are using `Write-Information` to capture additional information.
- If you have really been paying attention, some of you may also noticed that we changed the $Ports hashtable into an `[ordered]` hashtable. We wanted the ports to always be tested in the same order and this is an easy way to accomplish that.
- We removed comments where a verbose message would provide the same information.

Let's Review

See if you can answer these questions:

1. In PowerShell v5 and later, what command does `Write-Host` wrap around?
2. How does the person running your function get Verbose output from it?
3. What are the common parameters related to the information stream?
4. What are some of the benefits of incorporating verbose output into your commands?

Review Answers

Here are our answers:

1. `Write-Information`.
2. Running the command with the -Verbose parameter, or manually setting $VerbosePreference to "Continue."
3. The `-InformationAction` and `-InformationV$ivariable` parameters.
4. You can get detailed information about what your script is doing which can be logged as well as provide useful debugging or troubleshooting information.

Comment-Based Help

One of the things we all love about PowerShell is its help system. Like Linux' "MAN" pages, PowerShell's help files can provide a wealth of information, examples, instructions, and more. So we definitely want to provide help with the tools we create - and you should, too. You've got two ways of doing so. First, write "full" PowerShell help. That consists of external, XML-formatted "MAML" (Microsoft Assistance Markup Language) files, which can even include versions for different languages. We'll show you how to do that in the next Part of this book. For now, we're instead going to use the simpler, single-language comment-based help[17] that lives right inside your function.

Where to Put Your Help

There are three "legal" places where PowerShell will look for your specially formatted comments, in order to turn them into help displays:

1. Just before your function's opening `function` keyword, with no blank lines between the last comment line and your function. We don't like this spot, because we prefer...
2. Just inside your function, after the opening `function` declaration and before your `[CmdletBinding()]` or `Param` parts. We prefer this spot, because it's easier to move the help with your function if you're copying and pasting your code someplace else. Your comments will also "collapse" into the function if you use an editor that has "code-folding" features.
3. Or stick it as the last thing in your function before the closing `}`. We're not fans of this spot, either, because having your comments at the top of the function helps better document the function for someone reading the code.

Getting Started

Let's just dive in with a completed example using the `Get-TMComputerStatus` example.

[17] https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.core/about/about_comment_based_help

Get-MachineInfo

```
Function Get-TMComputerStatus {
 <#
.SYNOPSIS
Get computer status information.
.DESCRIPTION
This command retrieves system information from one or more remote computers
using Get-CimInstance. It will write a summary object to the pipeline for
each computer. You also have the option to log errors to a text file.
.PARAMETER Computername
The name of the computer to query.
.PARAMETER ErrorLog
The path to the error text file. This is not implemented yet.
.PARAMETER ErrorAppend
Append errors to the error file. This is not implemented yet.
.EXAMPLE
PS C:\> Get-TMComputerstatus -computername SRV1
#>
[cmdletbinding()]
Param(
    [Parameter(ValueFromPipeline, Mandatory)]
    [ValidateNotNullorEmpty()]
    [ValidatePattern("^\w+$")]
    [Alias("CN", "Machine", "Name")]
    [string[]]$Computername,
    [string]$ErrorLog,
    [switch]$ErrorAppend
)

BEGIN {
    Write-Information "Command = $($myinvocation.mycommand)" -Tags Meta
    Write-Information "PSVersion = $($PSVersionTable.PSVersion)" -Tags Meta
    Write-Information "User = $env:userdomain\$env:username" -tags Meta
    Write-Information "Computer = $env:computername" -tags Meta
    Write-Information "PSHost = $($host.name)" -Tags Meta
    Write-Information "Test Date = $(Get-Date)" -tags Meta
    Write-Verbose "Starting $($myinvocation.mycommand)"
}

PROCESS {
    foreach ($computer in $Computername) {
        Write-Verbose "Querying $($computer.toUpper())"
        $params = @{
```

```
            Classname    = "Win32_OperatingSystem"
            Computername = $computer
        }
        $OS = Get-CimInstance @params
        $params.ClassName = "Win32_Processor"
        $cpu = Get-CimInstance @params

        $params.className = "Win32_logicalDisk"
        $vol = Get-CimInstance @params -filter "DeviceID='c:'"

        [pscustomobject]@{
          Computername = $os.CSName
          TotalMem     = $os.TotalVisibleMemorySize
          FreeMem      = $os.FreePhysicalMemory
          Processes    = $os.NumberOfProcesses
          PctFreeMem   = ($os.FreePhysicalMemory/$os.TotalVisibleMemorySize)*100
          Uptime       = (Get-Date) - $os.lastBootUpTime
          CPULoad      = $cpu.LoadPercentage
          PctFreeC     = ($vol.FreeSpace/$vol.size)*100
        }
    } #foreach $computer
}
END {
    Write-Verbose "Starting $($myinvocation.mycommand)"
}
} #Get-TMComputerStatus
```

The help here reflects what we believe is the bare minimum for inclusion in the race of Upright Human Beings. Some notes:

- You do not have to use all-uppercase letters, but the period preceding each help keyword (.SYNOPSIS, .DESCRIPTION) must be in the first column.
- We used a block comment; you could also use line-by-line comments. The block comment looks nicer and is considered a collapsible region.
- .SYNOPSIS is meant to be a very short description of what your command does.
- .DESCRIPTION is a longer description, which can be full of details, instructions, and insights.
- .PARAMETER *is followed by the parameter name*, and then followed by a description of the parameter's use. You do not need to provide a listing for every single parameter.
- .EXAMPLE should be followed immediately *by the example itself*; PowerShell will add a PowerShell prompt in front of this line when the help is displayed. Although if your tool takes advantage of different providers such as the registry, you can certainly insert an appropriate prompt to illustrate your example. Subsequent lines can explain the example.

- You can have blank comment lines between each of these settings make it all easier to read in code.
- You normally don't need to worry about line length. PowerShell will wrap lines as necessary depending on the console size of the current host. However if you want to manually break lines, a width of 80 characters is your best bet.

Here's a slightly revised and "prettier" help block.

```
 <#
.SYNOPSIS
Get computer status information.

.DESCRIPTION
This command retrieves system information from one or more remote computers
using Get-CimInstance. It will write a summary object to the pipeline for
each computer. You also have the option to log errors to a text file.

.PARAMETER Computername
The name of the computer to query. This parameter has aliases of
CN, Machine and Name.

.PARAMETER ErrorLog
The path to the error text file. This is not implemented yet.

.PARAMETER ErrorAppend
Append errors to the error file. This is not implemented yet.

.EXAMPLE
PS C:\> Get-TMComputerStatus -computername SRV1

Computername : SRV1
TotalMem     : 33080040
FreeMem      : 27384236
Processes    : 218
PctFreeMem   : 82.7817499616083
Uptime       : 11.06:23:52.7176115
CPULoad      : 2
PctFreeC     : 54.8730920184876

Get the status of a single computer

.EXAMPLE
PS C:\> Get-Content c:\work\computers.txt | Get-TMComputerStatus |
```

```
Export-CliXML c:\work\data.xml

Pipe each computer name from the computers.txt text file to this
command. Results are immediately exported to an XML file using
Export-CliXML.
#>
```

As we wrote, these elements are the *bare minimum*. You can do more. A lot more.

Going Further with Comment-Based Help

You can use an `.INPUTS` section to list .NET class types, one per line, that your command accepts as input from the pipeline.

```
.INPUTS
System.String
```

Similarly, `.OUTPUTS` lists the type names that your script outputs. Because ours presently only outputs a generic PSObject, there's not much point in listing anything.

A `.NOTES` section can list additional information, which is only displayed when the "full" help is requested by the user.

```
.NOTES
version     : 1.0.0
last updated: 1 June, 2020
```

A `.LINK`, followed by a topic name or a URL, will show up as a "Related Topic" in the help. Use one `.LINK` keyword for each related topic; don't put multiples under a single .LINK.

```
.LINK
https://powershell.org/forums/
.LINK
Get-CimInstance
```

There's more, too - read the `about_comment_based_help` topic in PowerShell for the full list. A few of them, you'll see us include in upcoming chapters, as we add functionality that pertains to those help keywords. So be on the lookout.

Broken Help

PowerShell's a little bit picky - okay, a lot bit picky - about help formatting and syntax. Get just one thing wrong, and none of the help will work AND you'll get no error message or explanation. So if you're not getting the help display you expect, go review your help keyword spelling, period locations, and other details *very carefully*.

Your Turn

Time to add some comment-based help to your function.

Start Here

Here's where we left off after the most recent chapter. You can use this as a starting point, or use your own result from that chapter:

```
Function Get-TMRemoteListeningConfiguration {
  [cmdletbinding()]
  Param(
    [Parameter(ValueFromPipeline,Mandatory)]
    [ValidateNotNullorEmpty()]
    [Alias("CN")]
    [string[]]$Computername,
    [string]$ErrorLog
  )

  Begin {
   Write-Information "Command = $($myinvocation.mycommand)" -Tags Meta
   Write-Information "PSVersion = $($PSVersionTable.PSVersion)" -Tags Meta
   Write-Information "User = $env:userdomain\$env:username" -tags Meta
   Write-Information "Computer = $env:computername" -tags Meta
   Write-Information "PSHost = $($host.name)" -Tags Meta
   Write-Information "Test Date = $(Get-Date)" -tags Meta

    #define a private function to write the verbose messages
    Function WV {
    Param($prefix,$message)
        $time = Get-Date -f HH:mm:ss.ffff
        Write-Verbose "$time [$($prefix.padright(7,' '))] $message"
    }
```

```
    WV -prefix BEGIN -message "Starting $($myinvocation.MyCommand)"

    #define a ordered hashtable of ports so that the testing
    #goes in the same order
    $ports = [ordered]@{
      WSManHTTP  = 5985
      WSManHTTPS = 5986
      SSH        = 22
    }

    #initialize an splatting hashtable
    $testParams = @{
      Port         = ""
      Computername = ""
      WarningAction = "SilentlyContinue"
      WarningVariable = "wv"
    }
    #keep track of total computers tested
    $total=0
    #keep track of how long testing takes
    $begin = Get-Date
  } #begin
  Process {
    foreach ($computer in $computername) {
      $total++
      #make the computername all upper case
      $testParams.Computername = $computer.ToUpper()

      WV PROCESS "Testing $($testParams.Computername)"

      #define the hashtable of properties for the custom object
      $props = [ordered]@{
        Computername = $testparams.Computername
        Date         = Get-Date
      }

      #this array will be used to store passed ports
      #It is used by Write-Information
      $passed = @()

      #enumerate the hashtable
      $ports.GetEnumerator() | ForEach-Object {
        $testParams.Port = $_.Value
```

```
        WV "PROCESS" "Testing port $($testparams.port)"
        $test = Test-NetConnection @testParams

        WV PROCESS "Adding results"
        $props.Add($_.name, $test.TCPTestSucceeded)
        if ($test.TCPTestSucceeded) {
            $passed+=$testParams.Port
        }

        if (-NOT $props.Contains("RemoteAddress")) {
           wv "PROCESS" "Adding RemoteAddress $($test.remoteAddress)"
          $props.Add("RemoteAddress", $test.RemoteAddress)
        }
      }

      Write-Information "$($testParams.Computername) = $($passed -join ',')" `
      -Tags data

      $obj = New-Object -TypeName PSObject -Property $props
      Write-Output $obj

      #TODO: error handling and logging
    } #foreach
  } #process
  End {
    $runtime = New-TimeSpan -Start $begin -End (Get-Date)
    WV END "Processed $total computer(s) in $runtime"
    WV END "Ending $($myinvocation.mycommand)"
  } #end

} #Get-TMRemoteListeningConfiguration function
```

Your Task

Add, at a minimum, the following to your tool:

- Synopsis
- Description
- Parameter descriptions
- Two examples, including descriptions

Import your module and test your help (`Help Get-TMRemoteListeningConfiguration -ShowWindow`, for example) to make sure it works.

Our Take

Here's the help we came up with. As always, you'll find this in the code downloads, under this chapter's folder.

Get-TMRemoteListeningConfiguration Help

```
Function Get-TMRemoteListeningConfiguration {

<#
.Synopsis
Test remote listening ports.

.Description
This command will be used to test the network listening configuration on one
or more remote computers. It will test if standard PowerShell remoting is
enabled on both ports 5985 and 5986. It will also test if SSH is enabled on
port 22.

Because the command is testing at the network layer, you do not need to use
any credentials.

.Parameter Computername
Specify the name of IP address of a remote computer.

.Parameter ErrorLog
Specify the path to a text file to log errors. This feature is still in
development.

.Example
PS C:\> Get-TMRemoteListeningConfiguration -Computername srv1

Computername  : SRV1
Date          : 7/28/2020 5:13:57 PM
WSManHTTP     : True
RemoteAddress : 192.168.3.50
WSManHTTPS    : False
SSH           : False

Get configuration information for a single computer.
```

```
.Example
PS C:\> Get-Content c:\work\computers.txt | Get-TMRemoteListeningConfiguration | Exp\
ort-CSV c:\work\results.csv -NoTypeinformation

Get remote listening information for every computer in the computers.txt file and
export results to a CSV file.

.Inputs
System.String

.Link
Test-NetConnection

#>
```

Let's Review

Let's make sure you picked up on a couple of key points from this chapter:

1. What must follow the .PARAMETER keyword?
2. What must follow the .EXAMPLE keyword?
3. What is our recommended location for comment based help?
4. What happens if you misspell one of the help headings?

Review Answers

Here are our answers to the review questions:

1. On the same line, the parameter name. On the following lines, a description of the parameter's purpose.
2. On the same line, nothing. On the next line, the command example. On subsequent lines, a description or explanation of the example.
3. Between the Function title and `[Cmdletbinding()]`.
4. You will get PowerShell's auto-generated help.

Handling Errors

We have a lot of functionality yet to write in the tool we've been building, and we've been deferring a lot of it to this point. In this chapter, we'll focus on how to capture, deal with, log, and otherwise "handle" errors that our tool may encounter. As a note, PowerShell.org offers a free eBook, *The Big Book of PowerShell Error Handling*, which dives into this topic from a more technical reference perspective. We recommend checking it out, once you've completed this tutorial-focused chapter.

Understanding Errors and Exceptions

PowerShell defines two broad types of "bad situation:" an *error* and an *exception*. Because most PowerShell commands are designed to deal with multiple things at once, and because in many cases a problem with one thing doesn't mean you want to stop dealing with all the other things, PowerShell tries to err on the side of, "just keep going." So, often times, when something goes wrong in a command, PowerShell will emit an *error* and keep going. For example:

```
Get-Service -Name BITS,Foo,WinRM
```

There's no service named "Foo," and so PowerShell will emit an *error* on that second item. But by default, PowerShell will *keep going* and process the third item in the list. When PowerShell is in this "keep going" mode, **you cannot have your code respond to the problem condition**. So, if you want to do something about the problem, you have to change PowerShell's behavior to this kind of *non-terminating error*.

At a global level, PowerShell defines a `$ErrorActionPreference` variable, which tells PowerShell what to do in the event of a non-terminating error. This variable tells PowerShell what to do when a problem comes up, but PowerShell is able to keep going. The default value for this variable is "Continue". The options are:

- "Continue" means emit an error message and keep going. Your code can't detect that a problem occurred, so you can't do anything else.
- "SilentlyContinue" means emit no error message, and keep going. Again, you can't detect the problem or respond to it yourself.
- "Inquire" tells PowerShell to display a prompt, and ask the user whether to continue or stop.
- "Stop" tells PowerShell to turn the non-terminating *error* into a *terminating exception* and stop running the command. *This* is something your code can detect and respond to.

Check out the help topic `about_preference_variables`.

Rather than changing `$ErrorActionPreference` at the scope level, you'll typically want to specify a behavior on a per-command basis. You can do this using the `-ErrorActionPreference` common parameter, or its alias `-EA`, that exists on each and every PowerShell command - even the ones you write yourself when you include `[CmdletBinding[]`.

For example, try running these commands and note the different behaviors:

```
Get-Service -Name BITS,Foo,WinRM -EA Continue
Get-Service -Name BITS,Foo,WinRM -EA SilentlyContinue
Get-Service -Name BITS,Foo,WinRM -EA Inquire
Get-Service -Name BITS,Foo,WinRM -EA Stop
```

The thing to remember is that *you can't handle exceptions in your code* unless PowerShell *actually generates an exception*. Most commands *won't* generate an exception unless you run them with the "Stop" error action. One of the biggest mistakes people make is forgetting to add `-EA Stop` to a command that they want to properly handle when something goes wrong.

Bad Handling

There are two fundamentally bad practices that we see people engaging in. Now, these aren't *always always always* bad, but they're *usually* bad, so we want to bring them to your attention.

First up is globally setting `$ErrorActionPreference='SilentlyContinue'` right at the top of a script. In the days of VBScript people used `On Error Resume Next`. This is essentially saying, "I don't want to know if anything's wrong with my code." People will do this in a misguided attempt to suppress possible errors that they know won't matter. For example, attempting to delete a file which does not exist will cause an error - but you probably don't care, because "mission accomplished" either way, right? But to suppress that unwanted error, you should be using `-EA SilentlyContinue` on the `Remove-Item` command, not globally suppressing *all* errors in your script.

The other bad practice is a bit more subtle, and can come up in the same situation. Suppose you *do* run `Remove-Item` with `-EA SilentlyContinue`, and then suppose you try to delete a file that does exist, but that you simply don't have permission to. You'll be suppressing the error, and wondering while the file still exists. So before you just start suppressing errors, make sure you've really thought it through. Nothing is more vexing than spending hours debugging a script, simply because you suppressed an error message that would have told you right where the problem was.

Two Reasons for Exception Handling

There are two very broad reasons to handle exceptions in your code. Oh, notice that we're using their official name, *exceptions*, to differentiate from the non-handle-able *errors* that we wrote of previously.

Reason one is that you plan to run your tool out of your view. Perhaps it's a scheduled task, or perhaps you're writing tools that will be used by remote customers. In other case, you want to make sure that you have some evidence for any problems that occur, to help you with your debugging. In this scenario, you might globally set `$ErrorActionPreference` to "Stop" at the top of your script, and wrap your entire script in an error-handling construct. That way, any errors, even unanticipated ones, can be trapped and logged for diagnostic purposes. While this is a very valid scenario, it isn't the one we're going to focus on in this book.

We'll focus on reason two, which is that you're running a command *where you can anticipate a certain kind of problem occurring*, and you want to actively deal with that problem. This might be a failure to connect to a computer, a failure to log on to something, or some other scenario along those lines. So let's dig into that with the tool we've been building.

Handling Exceptions in Our Tool

In the tool we've been building to get computer status information, we can anticipate the `Get-CimInstance` command running into problems, either should a computer simply be offline or nonexistent. We want to handle that condition and, depending on the parameters we were run with, log the failed computer name to a text file So we'll start by focusing on the command that could cause the problem, and make sure that it'll generate a *terminating exception* if it runs into trouble. We'll change this:

```
foreach ($computer in $Computername) {
    Write-Verbose "Querying $($computer.toUpper())"
    $params = @{
        Classname    = "Win32_OperatingSystem"
        Computername = $computer
    }
    $OS = Get-CimInstance @params
```

Into this:

```
foreach ($computer in $Computername) {
    Write-Verbose "Querying $($computer.toUpper())"
    $params = @{
        Classname    = "Win32_OperatingSystem"
        Computername = $computer
        ErrorAction = "Stop"
    }
    $OS = Get-CimInstance @params
```

Now, it's hugely important to notice that we've already constructed our command so that it's only ever attempting to connect to one computer at a time. That happens by means of our `ForEach` loop. Any time you're going to be handling errors, it's crucial that you construct things so that only *one thing can fail at a time*. That's because we're telling PowerShell *to not continue*. Were we to attempt five computers at once, a failure in any of them would result in the rest of them never being attempted. Make sure you understand why this design principle is so important!

Simply changing the error action to "Stop" isn't enough, though. We also need to wrap our code in a `Try/Catch` construct. If an exception occurs in the `Try` block, then the entire rest of the `Try` block will be skipped, and the `Catch` block will execute instead. So the `PROCESS{}` block of our function begins looks like this:

```
PROCESS {
  foreach ($computer in $Computername) {
     Write-Verbose "Querying $($computer.toUpper())"
     $params = @{
         Classname    = "Win32_OperatingSystem"
         Computername = $computer
         ErrorAction  = "Stop"
     }
     Try {
         $OS = Get-CimInstance @params
         $OK = $True
     }
     Catch {
         $OK = $False
         #exception handling code...
     }
...
```

The idea here is that, should a problem happen with `Get-CimInstance`, then *everything else gets abandoned*. That should make sense. If the first time `Get-CimInstance` is run it fails, there's no reason to attempt to get the other data. If that one thing goes wrong, we need to quit.

Now let's focus on what we'll do if an error - sorry, an *exception* - does occur:

```
foreach ($computer in $Computername) {
     Write-Verbose "Querying $($computer.toUpper())"
     $params = @{
         Classname    = "Win32_OperatingSystem"
         Computername = $computer
         ErrorAction  = "Stop"
     }
     Try {
         $OS = Get-CimInstance @params
         $OK = $True
     }
     Catch {
         $OK = $False
         $msg = "Failed to get system information from $computer. `
         $($_.Exception.Message)"
         Write-Warning $msg
         if ($ErrorLog) {
             Write-Verbose "Logging errors to $ErrorLog. Append = $ErrorAppend"
            "[$(Get-Date)] $msg" | Out-File -FilePath $ErrorLog -Append:$ErrorAppend
         }
     }
     if ($OK) {
        #only continue if successful
        $params.ClassName = "Win32_Processor"
        $cpu = Get-CimInstance @params
        ...
```

First, we apologize if some of that wraps - remember, if you see a backslash in the last column is an indicator that the formatter for the book couldn't fit everything into the printed page width.. As always, the code's available in the downloads - so grab that for an un-wrapped version!

Anyway, here's what's happening.

- **First** We're using error handling on the first `Get-CimInstance` command. If it fails, PowerShell will create a terminating exception. That's the purpose of setting the common ErrorAction parameter to Stop.
- **Second** If an error occurs, the exception object will be *caught* by the Catch block. `$_` in this context means the exception object. This is a rich object. What we're doing is creating a message indicating which computer failed and the exception message. We're using `Write-Warning` to display this message.
- **Third** If the user also specified the path to a log file, we will write it to the file. Notice our trick to pass the `-ErrorAppend` parameter value to `Out-File`. This works because both parameters are switches. It's worth mentioning that we could, and probably should have error handling for

`Out-File` in the event the user specified a non-existent location or one they don't have access to. But we didn't want this example to be any more complicated.

- **Fourth** After the exception is caught and handled, or if there was no problem, PowerShell jumps to the next line of code. In our case it is the If statement. Back in the `Try` block, if there are no errors we set a variable, `$OK`, to True. We are using this to decide if the code should continue or not.

Technically, the other `Get-CimInstance` commands could also be wrapped in `Try/Catch` but that felt like overkill. Assuming we didn't misspell anything, if we have access and the computer is online, they should work. The other option is to move all of `Get-Ciminstance` commands into the`Try` block and use the same `Catch` block to handle all errors. This is a bit of complex logic - so go through it a few times and make sure you understand it!

Handling Exceptions for Non-Commands

What if you're running something - like a .NET Framework method - that doesn't have an `-ErrorAction` parameter? Well, in *most* cases, you can just run them in a `Try` block as-is, because *most* of them will throw trappable, terminating exceptions if something goes wrong. The "non-terminating exception" thing is kind of unique to PowerShell commands, like functions and cmdlets.

However, you *still* may have instances when you need to do this:

```
Try {
    $ErrorActionPreference = "Stop"
    # run something that doesn't have -ErrorAction
    $ErrorActionPreference = "Continue"
} Catch {
    # ...
}
```

This is your error handling of last resort. Basically, we're temporarily modifying `$ErrorActionPreference` for the duration of the one command (or whatever) we want to trap an exception for. This isn't at all a common situation in our experience, but we figured we'd point it out.

Going Further with Exception Handling

It's possible to have multiple `Catch` blocks after a given `Try` block, with each `Catch` dealing with a specific type of exception. For example, if a file deletion failed, you could react differently for a "File Not Found" or an "Access Denied" situation. To do this, you'll need to know the .NET Framework type name of each exception you want to call out separately. *The Big Book of PowerShell Error Handling* has a list of common ones, and advice for figuring these out (e.g., generating the error on your own in an experiment, and then figuring out what the exception type name was). Broadly, the syntax looks like this:

```
Try {
    # something here generates an exception
} Catch [Exception.Type.One] {
    # deal with that exception here
} Catch [Exception.Type.Two] {
    # deal with the other exception here
} Catch {
    # deal with anything else here
} Finally {
    # run something else
}
```

Also shown in that example is the optional `Finally` block, which will always run after the `Try` or the `Catch`, whether an exception occurs or not.

Deprecated Exception Handling

You may, in your Internet travels, run across a `Trap` construct in PowerShell. This dates back to v1, when the PowerShell team frankly didn't have time to get `Try/Catch` working, and `Trap` was the best short-term fix they could come up with. `Trap` is *deprecated*, meaning that it's left in the product for backward compatibility, but you're not intended to use it in newly written code. For that reason, we're not covering it here. It *does* have some uses for global, "I want to catch and log any possible error" situations, but `Try/Catch` is considered a more structured, professional approach to exception handling, and we recommend you stick with it.

Your Turn

It's time to deal with errors in your code.

Start Here

This is where we left off at the end of the previous chapter; you can use this as a starting point, or use your own results from that chapter.

```
Function Get-TMRemoteListeningConfiguration {

  [cmdletbinding()]
  Param(
    [Parameter(ValueFromPipeline,Mandatory)]
    [ValidateNotNullorEmpty()]
    [Alias("CN")]
    [string[]]$Computername,
    [string]$ErrorLog
  )

  Begin {
   Write-Information "Command = $($myinvocation.mycommand)" -Tags Meta
   Write-Information "PSVersion = $($PSVersionTable.PSVersion)" -Tags Meta
   Write-Information "User = $env:userdomain\$env:username" -tags Meta
   Write-Information "Computer = $env:computername" -tags Meta
   Write-Information "PSHost = $($host.name)" -Tags Meta
   Write-Information "Test Date = $(Get-Date)" -tags Meta

    #define a private function to write the verbose messages
    Function WV {
    Param($prefix,$message)
        $time = Get-Date -f HH:mm:ss.ffff
        Write-Verbose "$time [$($prefix.padright(7,' '))] $message"
    }

    WV -prefix BEGIN -message "Starting $($myinvocation.MyCommand)"

    #define a ordered hashtable of ports so that the testing
    #goes in the same order
    $ports = [ordered]@{
      WSManHTTP  = 5985
      WSManHTTPS = 5986
      SSH        = 22
    }

    #initialize an splatting hashtable
    $testParams = @{
      Port         = ""
      Computername = ""
      WarningAction = "SilentlyContinue"
      WarningVariable = "wv"
    }
```

```
    #keep track of total computers tested
    $total=0
    #keep track of how long testing takes
    $begin = Get-Date
  } #begin
  Process {
    foreach ($computer in $computername) {
      $total++
      #make the computername all upper case
      $testParams.Computername = $computer.ToUpper()

      WV PROCESS "Testing $($testParams.Computername)"

      #define the hashtable of properties for the custom object
      $props = [ordered]@{
        Computername = $testparams.Computername
        Date         = Get-Date
      }

      #this array will be used to store passed ports
      #It is used by Write-Information
      $passed = @()

      #enumerate the hashtable
      $ports.GetEnumerator() | ForEach-Object {
        $testParams.Port = $_.Value

        WV "PROCESS" "Testing port $($testparams.port)"
        $test = Test-NetConnection @testParams

        WV PROCESS "Adding results"
        $props.Add($_.name, $test.TCPTestSucceeded)
        if ($test.TCPTestSucceeded) {
            $passed+=$testParams.Port
        }

        if (-NOT $props.Contains("RemoteAddress")) {
           wv "PROCESS" "Adding RemoteAddress $($test.remoteAddress)"
          $props.Add("RemoteAddress", $test.RemoteAddress)
        }
      }

      Write-Information "$($testParams.Computername) = $($passed -join ',')" `
```

```
      -Tags data

      $obj = New-Object -TypeName PSObject -Property $props
      Write-Output $obj

      #TODO: error handling and logging
    } #foreach
  } #process
  End {
    $runtime = New-TimeSpan -Start $begin -End (Get-Date)
    WV END "Processed $total computer(s) in $runtime"
    WV END "Ending $($myinvocation.mycommand)"
  } #end

} #Get-TMRemoteListeningConfiguration
```

Your Task

Your job is to add error handling to your tool. This might be a bit trickier than you realize. How errors are handled is also part of the cmdlet design. `Get-Service` behaves one way if you give it a bad name. But if you give `Test-NetConnection` a bad name, that doesn't mean there is an error as far as the cmdlet is concerned. Rather, the test simply fails. You may want to capture the warning messages and generate our own "exceptions". Or modify the function to use additional commands.

Our Take

Here's what we came up with.

Get-TMRemoteListeningConfiguration

```
Function Get-TMRemoteListeningConfiguration {

<#
.Synopsis
Test remote listening ports.

.Description
This command will be used to test the network listening configuration on one
or more remote computers. It will test if standard PowerShell remoting is
enabled on both ports 5985 and 5986. It will also test if SSH is enabled on
port 22.

Because the command is testing at the network layer, you do not need to use
```

```
any credentials.

.Parameter Computername
Specify the name of IP address of a remote computer.

.Parameter ErrorLog
Specify the path to a text file to log errors. Entries will be automatically
appended. Make sure the folder location exists.

.Example
PS C:\> Get-TMRemoteListeningConfiguration -Computername srv1

Computername  : SRV1
Date          : 7/28/2020 5:13:57 PM
WSManHTTP     : True
RemoteAddress : 192.168.3.50
WSManHTTPS    : False
SSH           : False

Get configuration information for a single computer.

.Example
PS C:\> Get-Content c:\work\computers.txt | Get-TMRemoteListeningConfiguration -erro\
rlog c:\work\tmerrors.txt | Export-CSV c:\work\results.csv -NoTypeinformation

Get remote listening information for every computer in the computers.txt file and
export results to a CSV file. Any errors will be logged to c:\work\tmerrors.txt.

.Inputs
System.String

.Link
Test-NetConnection

#>
[cmdletbinding()]
Param(
    [Parameter(ValueFromPipeline, Mandatory)]
    [ValidateNotNullorEmpty()]
    [Alias("CN")]
    [string[]]$Computername,
    [string]$ErrorLog
)
```

```
Begin {
    Write-Information "Command = $($myinvocation.mycommand)" -Tags Meta
    Write-Information "PSVersion = $($PSVersionTable.PSVersion)" -Tags Meta
    Write-Information "User = $env:userdomain\$env:username" -tags Meta
    Write-Information "Computer = $env:computername" -tags Meta
    Write-Information "PSHost = $($host.name)" -Tags Meta
    Write-Information "Test Date = $(Get-Date)" -tags Meta

    #define a private function to write the verbose messages
    Function WV {
        Param($prefix, $message)
        $time = Get-Date -f HH:mm:ss.ffff
        Write-Verbose "$time [$($prefix.padright(7,' '))] $message"
    }

    WV -prefix BEGIN -message "Starting $($myinvocation.MyCommand)"

    if ($errorlog) {
        WV BEGIN "Errors will be logged to $ErrorLog"
        $outParams = @{
            FilePath    = $ErrorLog
            Encoding    = "ascii"
            Append      = $True
            ErrorAction = "stop"
        }
    }
    #define a ordered hashtable of ports so that the testing
    #goes in the same order
    $ports = [ordered]@{
        WSManHTTP  = 5985
        WSManHTTPS = 5986
        SSH        = 22
    }

    #initialize an splatting hashtable
    $testParams = @{
        Port            = ""
        Computername    = ""
        WarningAction   = "SilentlyContinue"
        #changed variable to not be confusing with helper function
        WarningVariable = "warn"
    }
```

```
    #keep track of total computers tested
    $total = 0
    #keep track of how long testing takes
    $begin = Get-Date
} #begin
Process {
    foreach ($computer in $computername) {
        #assume the computer can be reached
        $ok = $True
        $total++
        #make the computername all upper case
        $testParams.Computername = $computer.ToUpper()

        WV PROCESS "Testing $($testParams.Computername)"

        #define the hashtable of properties for the custom object
        $props = [ordered]@{
            Computername = $testparams.Computername
            Date         = Get-Date
        }

        #this array will be used to store passed ports
        #It is used by Write-Information
        $passed = @()

        #enumerate the hashtable
        foreach ($item in $ports.GetEnumerator()) {

            $testParams.Port = $item.Value

            WV "PROCESS" "Testing port $($testparams.port)"
            $test = Test-NetConnection @testParams

            if ($warn -match "Name resolution of $($testParams.computername) failed"\
) {
                $msg = "[$(Get-Date)] $warn"
                if ($ErrorLog) {
                    Try {
                        $msg | Out-File @outParams
                    }
                    Catch {
                        Write-Warning "Failed to log error. $($_.exception.message)"
                    }
```

```
                } #if errorlog
                $ok = $False
                #break out of the ForEach loop
                break
            }

            WV PROCESS "Adding results"
            $props.Add($item.name, $test.TCPTestSucceeded)
            if ($test.TCPTestSucceeded) {
                $passed += $testParams.Port
            }

            if (-NOT $props.Contains("RemoteAddress")) {
                wv "PROCESS" "Adding RemoteAddress $($test.remoteAddress)"
                $props.Add("RemoteAddress", $test.RemoteAddress)
            }

        } #foreach port

        if ($ok) {
            WV PROCESS "Generating an object for $($testparams.computername)"
            Write-Information "$($testParams.Computername) = $($passed -join ',')" -\
Tags data

            $obj = New-Object -TypeName PSObject -Property $props
            Write-Output $obj
        }
    } #foreach computer

} #process
End {
    $runtime = New-TimeSpan -Start $begin -End (Get-Date)
    WV END "Processed $total computer(s) in $runtime"

    #display a warning if errors would captured
    if (Test-Path -Path $ErrorLog) {
        Write-Warning "Errors were detected. See $ErrorLog."
    }

    WV END "Ending $($myinvocation.mycommand)"
} #end

} #Get-TMRemoteListeningConfiguration
```

Again, apologies for any word-wrapping in there (indicated by backslashes); consult the downloadable code samples for a well-formatted version.

In our solution we're using the warning variable to test if something went wrong. In this function, the only thing that can really go wrong is if `Test-NetConnection` can't resolve the computername. We're expecting warnings for closed ports. We didn't use a cmdlet like this `Resolve-DNSName` because that doesn't indicate the computer is online. Nor is pinging reliable as firewall rules might prevent ICMP traffic.

One of the major changes we made was to switch from using `ForEach-Object` when processing each port to using a `ForEach` construct. We did this so that if the warning variable matches the name resolution error, we can log it and then break out of the loop. It makes no sense to test any other ports. If there is a matching warning this is where we can insert the logging mechanism. In the `End` block we've even added a bit of code to let the user know if errors were logged. For the sake of demonstration, we used a `Try/Catch` block for `Out-File`. Another option would be to use a `[ValidateScript()]` attribute on the `ErrorLog` parameter and test if the location exists and can be written to.

Let's Review

Try answering these questions to see if you picked up the main points of this chapter:

1. What does `$ErrorActionPreference` do?
2. What is the purpose of the `-ErrorAction` parameter?
3. What is PowerShell capable of handling in a `Try` construct?
4. When writing code that will handle errors, why is it important to only attempt one operation, against one target, at a time?
5. How might you handle an exception generated by a .NET Framework class method?
6. How many `Catch` blocks can go with a single `Try` block?

Review Answers

Here are our answers:

1. Sets a global behavior for non-terminating errors.
2. Overrides $ErrorActionPreference on a per-command basis.
3. Terminating exceptions, as opposed to non-terminating errors.
4. Because you have to turn errors into terminating exceptions, which could result in some targets not being processed.
5. Set $ErrorActionPreference to Stop, if the method isn't already returning a terminating exception. Most will.
6. As many as you want, provided all but the last one are targeting a specific exception type.

Basic Debugging

Debugging is pretty much the bane of everyone's existence. This chapter is designed to get you started with the core debugging concepts and tools in PowerShell. We'll revisit this topic in an upcoming chapter and expand into nittier, grittier techniques and tools.

Two Kinds of Bugs

There are two kinds of software bugs on this big blue marble we all live on. The first kind of bug is the easiest: *syntax*. This is literally where you mistype something, your script won't run, and you get a helpful error message telling you where the bug is. Funny thing is, *so many people don't read the helpful error messages!* It's probably because they're red and scary, so in the PowerShell console you can do this:

```
$host.privatedata.errorforegroundcolor = 'green'
```

Now your errors will "feel" better and you will read them!. The PowerShell ISE and VS Code, with their red-squiggly underlines and color-coding, also make it easier to spot syntax errors. Tab completion can make it easier to avoid syntax errors, and frankly you should be doing this all the time anywayt. Generally, these bugs are the simplest to squash, and so we will speak no more of them.

Bug Species Two is a lot harder: the *logic* bug. This is where your script *runs*, probably without error, but it doesn't do what you tell it to. Or, you get errors that make no darn sense, like a "illegal computer name" when you know perfectly well you've provided a legal computer name. That kind of thing. This is the kind of bug on which we will focus this chapter.

Logic bugs come from one basic source: bad assumptions.

- A variable contained a value other than the one you assumed.
- A command produced a result other than the one you assumed.
- An object property contained a value other than the one you assumed.

Basically, you assumed one thing was happening, but it wasn't, so bug.

The Ultimate Goal of Debugging

The ultimate goal of debugging, then, is to find out where your assumptions were wrong. Where does code execution diverge from your expectations. That means two things are needed:

1. You need to have assumptions about what your script is doing. This seems like an obvious statement, but we see a *lot* of people trying to debug scripts who have no clue what it's supposed to be doing. They just kind of fling themselves at the script and hope to beat it into submission through pure force of will. This does not work. If you can't look at a script and *make some assumptions* about what variables contain, what commands are producing, and what properties contain, then you cannot debug the script. Stop reading here and brush up on your PowerShell fundamentals.
2. You need to have tools that let you *validate your assumptions*. Believe it or not, this is the easy part in PowerShell. The first part, stating what you think the script is going to do, is the hard part.

We're going to spend a lot of time in this chapter, and again in a more advanced debugging chapter, covering that second bit - tools. But before we do that, we'd like to run you through an exercise to really demonstrate what we're talking about with that first bit - assumptions.

Developing Assumptions

Take a look at the following script. *Read through it.* In detail. At each and every line of code, ask yourself *what is this going to do? What will be the result?* and actually write that down on a piece of paper. Seriously, this is a good exercise - get some paper and do this. Write down all the variables, and what you think they contain. As that changes, lightly cross out your old value and write down the new one. When the script uses a property of some kind, from an object, write down what you think that contains, too. *Don't run any of this code.* If you don't know what a command does, that's fine - look up its help in PowerShell or online (use a search engine to look up the command name). Read the help, and make a guess at what the command does or is producing.

If you don't do this little exercise, we can't teach you debugging. Don't skip this. And no, we did not include this in the downloadable code because we want you looking at it here, not cheating and running it.

Here's your code:

Get-DriveInfo

```
function Get-DriveInfo {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory,ValueFromPipelineByPropertyName)]
        [string[]]$ComputerName
    )
    PROCESS {
        ForEach ($comp in $ComputerName) {

            $session = New-CimSession -ComputerName $comp
            $params = @{
                CimSession = $session
                ClassName  = 'Win32_LogicalDisk'
            }
            $drives = Get-CimInstance @params

            if ($drives.DriveType -notlike '*optical*') {
                [pscustomobject]@{
                    ComputerName = $comp
                    Letter       = $drives.deviceid
                    Size         = $drives.size
                    Free         = $drives.freespace
                }
            }
        } #foreach
    } #process
} #function
```

This command is *intended* to produce a list of drives, including size and free space, for all non-optical drives (e.g., no DVD drives) in the system. It doesn't work. With your set of assumptions in hand, let's start looking at some debugging tools.

Debugging Tool 1: Write-Debug

This is the first and perhaps easiest tool to use, especially in a very short script like this one. You just drop some `Write-Debug` commands in key places, much as you would use `Write-Verbose`. When you run your function, which must include `[CmdletBinding()]` as this one does, each `Write-Debug` statement will give you a chance to stop and examine your script. We like to include them at each point where something changes, or where the script is about to take a logical branching. So:

Step1.ps1

```
function Get-DriveInfo {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory, ValueFromPipelineByPropertyName)]
        [string[]]$ComputerName
    )
    PROCESS {
        Write-Debug "[PROCESS] Beginning"
        ForEach ($comp in $ComputerName) {

            Write-Debug "[PROCESS] on $comp"
            $session = New-CimSession -ComputerName $comp
            $params = @{
                CimSession = $session
                ClassName  = 'Win32_LogicalDisk'
            }
            $drives = Get-CimInstance @params

            Write-Debug "[PROCESS] CIM query complete"
            if ($drives.DriveType -notlike '*optical*') {
                [pscustomobject]@{
                    ComputerName = $comp
                    Letter       = $drives.deviceid
                    Size         = $drives.size
                    Free         = $drives.freespace
                }
            }
        } #foreach
    } #process
} #function

"localhost", "localhost" | Get-DriveInfo -Debug
```

That's in the downloadable code examples as Step1.ps1 in this chapter's folder. Run this from the console - the ISE is a little irritating to use with `Write-Debug`.

Our first assumption, based on the last line in the script, is that two strings, "localhost", will be passed into the `$ComputerName` parameter.

```
Confirm
The input object cannot be bound to any parameters for the command
either because the command does not take pipeline input or the input
and its properties do not match any of the parameters that take
pipeline input.
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help
(default is "Y"):
```

Apparently not. "...the command does not take pipeline input..." say what? It most certainly does... oh, wait. Our command takes pipeline input *by property name*, and we didn't pass in an object having a "ComputerName" property. We passed in String objects. We need to modify our parameter attribute. Here's Step2.ps1:

Step2.ps1

```
function Get-DriveInfo {
    [CmdletBinding()]
    Param(
        [Parameter(
        Mandatory,
            ValueFromPipelineByPropertyName,
            ValueFromPipeline)]
        [string[]]$ComputerName
    )
    PROCESS {
        Write-Debug "[PROCESS] Beginning"
        ForEach ($comp in $ComputerName) {

            Write-Debug "[PROCESS] on $comp"
            $session = New-CimSession -ComputerName $comp
            $params = @{
                CimSession = $session
                ClassName  = 'Win32_LogicalDisk'
            }
            $drives = Get-CimInstance @params

            Write-Debug "[PROCESS] CIM query complete"
            if ($drives.DriveType -notlike '*optical*') {
                [pscustomobject]@{
                    ComputerName = $comp
                    Letter       = $drives.deviceid
                    Size         = $drives.size
                    Free         = $drives.freespace
                }
```

```
            }
        } #foreach
    } #process
} #function

"localhost", "localhost" | Get-DriveInfo -Debug
```

Let's run it again.

```
DEBUG: [PROCESS] Beginning

Confirm
Continue with this operation?
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help
(default is "Y"):
```

This is `Write-Debug` in action. You can see our "Beginning" message, and the debugger is now active. We're going to hit `S` to Suspend the script. It'll be like we're in the script's head, and we just want to check that `$ComputerName` has something in it.

```
DEBUG: [PROCESS] Beginning

Confirm
Continue with this operation?
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help
(default is "Y"):s
PS z:\Documents\GitHub\ToolmakingBook\code\PowerShell-Toolmaking\
Chapters\basic-debugging>> $computername
localhost
PS z:\Documents\GitHub\ToolmakingBook\code\PowerShell-Toolmaking\
Chapters\basic-debugging>>
```

OK, it looks like `$ComputerName` has "localhost" in it, which was exactly what we expected. Perfect. So we'll run `Exit` and let the script continue by choosing `Y` for Yes.

```
PS z:\Documents\GitHub\ToolmakingBook\code\PowerShell-Toolmaking\
Chapters\basic-debugging>> exit

Confirm
Continue with this operation?
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help
(default is "Y"):y
DEBUG: [PROCESS] on localhost

Confirm
Continue with this operation?
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help
(default is "Y"):s
PS z:\Documents\GitHub\ToolmakingBook\code\PowerShell-Toolmaking\
Chapters\basic-debugging>> $comp
localhost
PS z:\Documents\GitHub\ToolmakingBook\code\PowerShell-Toolmaking\
Chapters\basic-debugging>> exit

Confirm
Continue with this operation?
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help
(default is "Y"):y
DEBUG: [PROCESS] CIM query complete

Confirm
Continue with this operation?
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help
(default is "Y"):
```

Here, we've suspended again and checked the contents of $comp. "Localhost", just as expected. We exited and allowed the script to continue. Now we're at our "query complete" prompt, so we'll suspend again and see what we got back.

```
Confirm
Continue with this operation?
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help
(default is "Y"):s
PS z:\Documents\GitHub\ToolmakingBook\code\PowerShell-Toolmaking\
Chapters\basic-debugging›› $drives

DeviceID DriveType ProviderName VolumeName            Size        Fre
                                                                  eSp
                                                                  ace
-------- --------- ------------ ----------            ----        ---
C:       3                                           63898120192 470
D:       5                     CCSA_X64FRE_EN-US_DV5 4380387328  0
```

We have a problem here. *Look at what $drives contains* and then look at what our script is going to do next. Look at that `If` statement. See where it's looking at the DriveType property? See how our script is expecting "*optical*" for the DriveType? See in the output above how it isn't "optical" at all, but is rather a digit? Dang.

```
PS z:\Documents\GitHub\ToolmakingBook\code\PowerShell-Toolmaking\
Chapters\basic-debugging›› exit

Confirm
Continue with this operation?
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help
(default is "Y"):h
Write-Debug : The running command stopped because the user selected
the Stop option.
At \\vmware-host\shared folders\Documents\GitHub\ToolmakingBook\code\
PowerShell-Toolmaking\Chapters\basic-debugging\Step2.ps1:21 char:13
+             Write-Debug "[PROCESS] CIM query complete"
+             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : OperationStopped: (:) [Write-Debug], P
   arentContainsErrorRecordException
    + FullyQualifiedErrorId : ActionPreferenceStop,Microsoft.PowerSh
   ell.Commands.WriteDebugCommand
```

Hey, no point continuing when we know it's broken. We need to fix our script. You see, we *assumed* DriveType would be something like "Fixed" or "Optical" or whatever, but it isn't. Apparently 5 means an optical drive. So we've found a logic bug in our script, because we *made ourselves aware of our assumptions* and then took the time to *verify* them.

Debugging Tool 2: Set-PSBreakpoint

We've fixed that line of our code, so here's Step3.ps1:

Step3.ps1

```
function Get-DriveInfo {
    [CmdletBinding()]
    Param(
        [Parameter(
            Mandatory,
            ValueFromPipelineByPropertyName,
            ValueFromPipeline)]
        [string[]]$ComputerName
    )
    PROCESS {
        Write-Debug "[PROCESS] Beginning"
        ForEach ($comp in $ComputerName) {

            Write-Debug "[PROCESS] on $comp"
            $session = New-CimSession -ComputerName $comp
            $params = @{
                CimSession = $session
                ClassName  = 'Win32_LogicalDisk'
            }
            $drives = Get-CimInstance @params

            Write-Debug "[PROCESS] CIM query complete"
            if ($drives.DriveType -ne 5) {
                [pscustomobject]@{
                    ComputerName = $comp
                    Letter       = $drives.deviceid
                    Size         = $drives.size
                    Free         = $drives.freespace
                }
            }
        } #foreach
    } #process
} #function

"localhost", "localhost" | Get-DriveInfo
```

Let's run it:

```
PS z:\Documents\GitHub\ToolmakingBook\code\PowerShell-Toolmaking\
Chapters\basic-debugging> .\Step3.ps1

ComputerName Letter   Size                       Free
------------ ------   ----                       ----
localhost    {C:, D:} {63898120192, 4380387328} {47037890560, 0}
localhost    {C:, D:} {63898120192, 4380387328} {47037890560, 0}
```

That... that is weird. We expected two lines of results, because we have it two computer names, but... those are not the droids we were looking for. What's with all the curly brackets? Time to do some more debugging. This time, let's play with PowerShell's built-in *breakpoint* feature.

Breakpoints work a lot like `Write-Debug` in that, when you "hit" a breakpoint, you get dumped into "suspend mode" and can "inspect" what's happening inside your script. Unlike `Write-Debug`, breakpoints don't display a prompt first - they just go into suspend mode. And, also unlike `Write-Debug`, breakpoints can be easily set, removed, enabled, and disabled. For long scripts, that means breakpoints can be a lot more convenient, because you don't have to keep hitting "Yes" through a bunch of early ones to get to the one you're actually after at the moment.

Our problem right now is that our output isn't what we expected, and so we want to "break" the script right before that output is produced. Breakpoints are tied to a specific script by path and filename, and only exist for the duration of the PowerShell session where you created the breakpoint. We'll add some lines at the bottom of our script file, just for convenience, so that you can see the breakpoints in the downloadable code samples. Normally, we'd just set these breakpoints right in the console window.

This is Step4.ps1.

Step4.ps1

```
function Get-DriveInfo {
    [CmdletBinding()]
    Param(
        [Parameter(
            Mandatory,
            ValueFromPipelineByPropertyName,
            ValueFromPipeline)]
        [string[]]$ComputerName
    )
    PROCESS {
        Write-Debug "[PROCESS] Beginning"
        ForEach ($comp in $ComputerName) {

            Write-Debug "[PROCESS] on $comp"
            $session = New-CimSession -ComputerName $comp
            $params = @{
```

```
                CimSession = $session
                ClassName  = 'Win32_LogicalDisk'
            }
            $drives = Get-CimInstance @params

            Write-Debug "[PROCESS] CIM query complete"
            if ($drives.DriveType -ne 5) {
                [pscustomobject]@{
                    ComputerName = $comp
                    Letter       = $drives.deviceid
                    Size         = $drives.size
                    Free         = $drives.freespace
                }
            }
        } #foreach
    } #process
} #function

Set-PSBreakpoint -Line 23 -Script ($MyInvocation.MyCommand.Source)
"localhost", "localhost" | Get-DriveInfo
```

Notice that `$MyInvocation.MyCommand.Source` trick? $MyInvocation is a built-in variable (we mentioned it before in the chapter on Verbose, Warning, and Informational output) that contains a bunch of stuff about the current script or command. In this case, we're using it to nab our script's source file location. This way, no matter where the file lives, we'll be attaching the breakpoint to the right file.

Here's our output:

```
PS X:\basic-debugging> .\Step4.ps1

  ID Script         Line Command        Variable       Action
  -- ------         ---- -------        --------       ------
   0 Step4.ps1        23
Entering debug mode. Use h or ? for help.

Hit Line breakpoint on '\\vmware-host\shared folders\Documents\GitHub\
ToolmakingBook\code\PowerShell-Toolmaking\Chapters\
basic-debugging\Step4.ps1:23'

At \\vmware-host\shared folders\Documents\GitHub\ToolmakingBook\code\
PowerShell-Toolmaking\Chapters\basic-debugging\Step4.ps1:23 char:17
+                 [pscustomobject]@{'ComputerName'=$comp
```

```
+                  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[DBG]: PS X:\basic-debugging>>
```

We've dropped into suspend mode at this point.

```
[DBG]: PS X:\basic-debugging>> $drives

DeviceID DriveType ProviderName VolumeName            Size        Fre
                                                                  eSp
                                                                  ace
-------- --------- ------------ ----------            ----        ---
C:       3                                           63898120192 470
D:       5                     CCSA_X64FRE_EN-US_DV5 4380387328  0
```

Ah. See, we knew this - `$drives` contains multiple objects, but we've been behaving, in our code, as if it contained one object. Look at line 22:

```
   if ($drives.DriveType -ne 5) {
```

This is like saying, "So, I have a bunch of cars. If they is red, then do this." Well, what if one car is red and another isn't? This is a nonsensical statement - just as our grammar, there, was nonsensical. We need to be looking at one drive at a time, and outputting an object for one drive at a time, rather than outputting one object for all the drives.

Here's Step5.ps1.

Step5.ps1

```
function Get-DriveInfo {
    [CmdletBinding()]
    Param(
        [Parameter(
            Mandatory,
            ValueFromPipelineByPropertyName,
            ValueFromPipeline)]
        [string[]]$ComputerName
    )
    PROCESS {
        Write-Debug "[PROCESS] Beginning"
        ForEach ($comp in $ComputerName) {

            Write-Debug "[PROCESS] on $comp"
            $session = New-CimSession -ComputerName $comp
            $params = @{
```

```
                CimSession = $session
                ClassName  = 'Win32_LogicalDisk'
            }

            $drives = Get-CimInstance @params
            Write-Debug "[PROCESS] CIM query complete"

            ForEach ($drive in $drives) {
                if ($drive.DriveType -ne 5) {
                    [pscustomobject]@{
                        ComputerName = $comp
                        Letter       = $drive.deviceid
                        Size         = $drive.size
                        Free         = $drive.freespace
                    }
                }
            } #foreach drive
        } #foreach computer
    } #process
} #function

Set-PSBreakpoint -Line 24 -Script ($MyInvocation.MyCommand.Source)
"localhost", "localhost" | Get-DriveInfo
```

Notice that we updated our breakpoint line number. Because we're using the same console session, *the old breakpoint will still be there*, so we'll set an additional one. Now, to run it:

```
PS X:\basic-debugging> .\Step5.ps1

  ID Script         Line Command        Variable       Action
  -- ------         ---- -------        --------       ------
   3 Step5.ps1        24
Hit Line breakpoint on '\\vmware-host\shared folders\Documents\GitHub\
ToolmakingBook\code\PowerShell-Toolmaking\Chapters\
basic-debugging\Step5.ps1:24'
Hit Line breakpoint on '\\vmware-host\shared folders\Documents\GitHub\
ToolmakingBook\code\PowerShell-Toolmaking\Chapters\
basic-debugging\Step5.ps1:24'

At \\vmware-host\shared folders\Documents\GitHub\ToolmakingBook\code\
PowerShell-Toolmaking\Chapters\basic-debugging\Step5.ps1:24 char:21
+                 if ($drive.DriveType -ne 5) {
```

```
+                     ~~~~~~~~~~~~~~~~~~~~~
[DBG]: PS X:\basic-debugging>> $drive

DeviceID DriveType ProviderName VolumeName Size        FreeSpace   PS
                                                                   Co
                                                                   mp
                                                                   ut
                                                                   er
                                                                   Na
                                                                   me
-------- --------- ------------ ---------- ----        ---------   --
C:       3                                 63898120192 47044632576 lo
```

OK - much more what we expected, and `$drive` contains one object, with a DriveType of `3`. We would expect this to pass our `If` construct and produce a line of output, so we'll continue:

```
[DBG]: PS X:\basic-debugging>> exit

ComputerName : localhost
Letter       : C:
Size         : 63898120192
Free         : 47044632576

Hit Line breakpoint on '\\vmware-host\shared folders\Documents\GitHub\
ToolmakingBook\code\PowerShell-Toolmaking\Chapters\basic-debugging\
Step5.ps1:24'
Hit Line breakpoint on '\\vmware-host\shared folders\Documents\GitHub\
ToolmakingBook\code\PowerShell-Toolmaking\Chapters\basic-debugging\
Step5.ps1:24'

At \\vmware-host\shared folders\Documents\GitHub\ToolmakingBook\code\
PowerShell-Toolmaking\Chapters\basic-debugging\Step5.ps1:24 char:21
+                 if ($drive.DriveType -ne 5) {
+                     ~~~~~~~~~~~~~~~~~~~~~
```

Got it! Now, let's clear out all the breakpoints we've created - we could just close the console window we've been using, but this works as well:

```
Get-PSBreakpoint | Remove-PSBreakpoint
```

Debugging Tool 3: The PowerShell ISE

The ISE provides a visual way of setting breakpoints. Just move to whatever line you want and press F9; the line will turn a dark red and you'll know where the breakpoint is. Press F9 again to clear it. It's really just running `Set-PSBreakpoint` and `Remove-PSBreakpoint` in the background; were you to manually run those in the ISE's console pane, you'd see the dark red highlights come and go.

The ISE also provides quick access to some of PowerShell's cooler debugging features. For example, go ahead and open Step6.ps1 in the ISE, and follow along:

Step6.ps1

```
function Get-DriveInfo {
    [CmdletBinding()]
    Param(
        [Parameter(
            Mandatory,
            ValueFromPipelineByPropertyName,
            ValueFromPipeline)]
        [string[]]$ComputerName
    )
    PROCESS {
        Write-Debug "[PROCESS] Beginning"
        ForEach ($comp in $ComputerName) {

            Write-Debug "[PROCESS] on $comp"
            $session = New-CimSession -ComputerName $comp
            $params = @{
                CimSession=$session
                ClassName='Win32_LogicalDisk'
            }

            $drives = Get-CimInstance @params
            Write-Debug "[PROCESS] CIM query complete"

            ForEach ($drive in $drives) {
                if ($drive.DriveType -ne 5) {
                    [pscustomobject]@{
                    ComputerName=$comp
                    Letter=$drive.deviceid
                    Size=$drive.size
                    Free=$drive.freespace
                }
                }
```

```
            } #foreach drive
        } #foreach computer
    } #process
} #function

"localhost","localhost" | Get-DriveInfo
```

1. Place your cursor anywhere on line 20 and press `F9`. For reference, line 20 is where `Get-CimInstance` runs.
2. Press `F5` to run the script.
3. The script will run down to line 20, which will turn a kind of baby-poop brown. The ISE console, if you're in split-screen mode, will show the `[DBG]` prompt, indicating you're in suspend mode.
4. In the console pane, type `$params` and hit `Enter` to verify the contents of the variable.
5. Press `F11`. Line 20 will turn back to breakpoint-red, while line 21 will now be a light yellow highlight. This indicates that line 20 has executed, and line 21 is about to.
6. Hit `F11` again. In the console pane, type `$drives` and hit `Enter` to verify the contents of the variable.
7. Hover your cursor over the highlighted $drives. You'll see the same thing - the contents of the variable. This is a cool and quick way to verify variables' contents.
8. Hit `Shift+F5` to stop the debugger.
9. Move to line 20 and press `F9` to remove the breakpoint.

Debugging Tool 4: VS Code

If your editor of choice is VS Code, it has the same type of debugging features. Open the Step6.ps1 in "VS Code. The keyboard shortcuts are the same is in the PowerShell ISE. Aside from possible color changes you should be able to follow the same steps as above, including hovering your mouse over variables to see their content.

Another nice feature of VS Code is the PSScriptAnalyzer is baked in and is always looking for potential issues. If you click on the Problems link (Ctr+Shift+M), you can see all of the things that might lead to bugs. This won't help get rid of logic bugs, but it will make your code less prone to errors and mistakes.

We'll get into a lot more detail and depth with breakpoints in an upcoming "advanced" chapter on debugging, but until then this should give you a great start.

Your Turn

You guessed it - it's time to do some debugging.

Start Here

You'll find this script as Exercise.ps1 in this chapter's folder, if you've downloaded the code samples. You may recognize it - it's the script you've been building, or at least our version of it. But this one is broken.

Exercise.ps1

```
#this version is broken in several ways
Function Get-TMRemoteListeningConfiguration {

[cmdletbinding()]
Param(
    [Parameter(ValueFromPipelineByValue, Mandatory)]
    [ValidateNotNullorEmpty()]
    [Alias("CN")]
    [string]$Computername,
    [string]$ErrorLog
)

Begin {
    Write-Information "Command = $($myinvocation.mycommand)" -Tags Meta
    Write-Information "PSVersion = $($PSVersionTable.PSVersion)" -Tags Meta
    Write-Information "User = $env:userdomain\$env:username" -tags Meta
    Write-Information "Computer = $env:computername" -tags Meta
    Write-Information "PSHost = $($host.name)" -Tags Meta
    Write-Information "Test Date = $(Get-Date)" -tags Meta

    #define a private function to write the verbose messages
    Function WV {
        Param($prefix, $message)
        $time = Get-Date -f HH:mm:ss.ffff
        Write-Verbose "$time [$($prefix.padright(7,' '))] $message"
    }

    WV -prefix BEGIN -message "Starting $($myinvocation.MyCommand)"

    if ($errorlog) {
        WV BEGIN "Errors will be logged to $ErrorLog"
        $outParams = @{
            FilePath    = $ErrorLog
            Encoding    = "ascii"
            Append      = $True
            ErrorAction = "stop"
```

```
        }
    }
    #define a ordered hashtable of ports so that the testing
    #goes in the same order
    $ports = [ordered]@{
        WSManHTTP  = 5985
        WSManHTTPS = 5986
        SSH        = 22
    }

    #initialize an splatting hashtable
    $testParams = @{
        Port            = ""
        Computername    = ""
        WarningAction   = "SilentlyContinue"
        #changed variable to not be confusing with helper function
        WarningVariable = "warn"
    }
    #keep track of total computers tested
    $total = 0
    #keep track of how long testing takes
    $begin = Get-Date
} #begin
Process {
    foreach ($computerr in $computername) {
        #assume the computer can be reached
        $ok = $True
        $total+1
        #make the computername all upper case
        $testParams.Computername = $computer.ToUpper()

        WV PROCESS "Testing $($testParams.Computername)"

        #define the hashtable of properties for the custom object
        $props = [ordered]@{
            Computername = $testparams.Computername
            Date         = Get-Date
        }

        #this array will be used to store passed ports
        #It is used by Write-Information
        $passed = @()
```

```
        #enumerate the hashtable
        foreach ($item in $ports.GetEnumerator()) {

            $testParams.Port = $item.Value

            WV "PROCESS" "Testing port $($testparams.port)"
            $test = Test-NetConnection @testParams

            if ($warn -match "Name resolution of $($testParams.computername) failed"\
) {
                $msg = "[$(Get-Date)] $warn"
                if ($ErrorLog) {
                    Try {
                        $msg | Out-File @outParams
                    }
                    Catch {
                        Write-Warning "Failed to log error. $($_.exception.message)"
                    }
                } #if errorlog
                $ok = $False
                #break out of the ForEach loop
                break
            }

            WV PROCESS "Adding results"
            $props.Add($item.name, $test.TCPTestSucceeded)
            if ($test.TCPTestSucceeded) {
                $passed += $testParams.Port
            }

            if (-NOT $props.Contains("RemoteAddress")) {
                wv "PROCESS" "Adding RemoteAddress $($test.remoteAddress)"
                $props.Add("RemoteAddress", $test.RemoteAddress)
            }
        } #foreach port

        if ($ok) {
            WV PROCESS "Generating an object for $($testparams.computername)"
            Write-Information "$($testParams.Computername) = $($passed -join ',')" -\
Tags data

            $obj = New-Object -TypeName PSObject -Property $props
            Write-Output $obj
```

```
        }
    } #foreach computer

} #process
End {
    $runtime = New-TimeSpan -Start (Get-Date) -End $end
    WV END "Processed $total computer(s) in $runtime"

    #display a warning if errors would captured
    if (Test-Path -Path $ErrorLog) {
        Write-Warning "Errors were detected. See $ErrorLog."
    }

    WV END "Ending $($myinvocation.mycommand)"
} #end

} #Get-TMRemoteListeningConfiguration
```

Your Task

Using whichever debugging techniques you prefer, fix the script. Be sure to test different ways of using the command, including different parameter combinations.

Our Take

Our sample solution is the same as the previous chapter's, so we won't repeat it here. Briefly:

1. On line 62, '$computer' is spelled wrong.
2. On lines 127, the start and end values for the timespan are reversed. This is an easy and common mistake.
3. On line 7, it should be `ValueFromPipeline`.
4. On line 10, the parameter should accept an array of strings, not just a single one. This too is a common error.
5. On line 65, the total number of computers isn't being incremented properly. The line should be `$total++`.

Let's Review

Let's see if you picked up some of the key points of this chapter:

1. How do breakpoints differ from `Write-Debug`?
2. Where do most logic bugs come from?

Review Answers

Here are our answers:

1. Breakpoints can be set and removed programmatically, in the ISE, and VS Code. They offer a more targeted way of suspending the script, and can be more convenient in longer scripts. They don't prompt before suspending.
2. A variable's contents, a property's contents, or a command's output, that wasn't what you thought it was.

Verify Yourself

We want to give you an opportunity to see if you're ready for the rest of this book. Here's what we're going to do: we'll give you a transcript from a PowerShell console session (the same one is included in the downloadable code samples, because the line-wrapping here in the book is gonna be pretty horrific). The transcript shows a custom PowerShell tool being used. Your job is to observe that usage, and then recreate that tool. We'll provide the original function in the downloadable code samples, but *do not peek* - you're only cheating yourself. At the end of this chapter, we'll do a blow-by-blow walk-through of what your brain should have been thinking as you read the transcript.

Here's a tip: *Read* the transcript first. As you go, make notes about the things you see, and what you'll need to do in order to duplicate those things. Then, start coding, checking off each thing you noted as you incorporate it into your code. The transcript file is too wide to properly fit the page of a printed book so the slashes you see at the end of lines are really line continuation characters.

The Transcript

Here you go:

```
**********************
Windows PowerShell transcript start
Start time: 20170623144152
Username: DESKTOP-7NKT52T\User
RunAs User: DESKTOP-7NKT52T\User
Machine: DESKTOP-7NKT52T (Microsoft Windows NT 10.0.14393.0)
Host Application: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
Process ID: 1412
PSVersion: 5.1.14393.1358
PSEdition: Desktop
PSCompatibleVersions: 1.0, 2.0, 3.0, 4.0, 5.0, 5.1.14393.1358
BuildVersion: 10.0.14393.1358
CLRVersion: 4.0.30319.42000
WSManStackVersion: 3.0
PSRemotingProtocolVersion: 2.3
SerializationVersion: 1.1.0.1
**********************
```

```
Transcript started, output file is .\transcript.txt
PS C:\> Get-XXSystemInfo -Computername localhost

BIOSSerial                                              ComputerName OSVersion
----------                                              ------------ ---------
VMware-56 4d 03 1c 3a c5 5f a3-d6 3c 01 92 aa e7 1d 45 localhost    10.0.14393


PS C:\> Get-XXSystemInfo -Computername localhost -verbose
VERBOSE: Attempting localhost on Wsman
VERBOSE: Operation '' complete.
VERBOSE:   [+] Connected
VERBOSE: Perform operation 'Enumerate CimInstances' with following parameters,
''namespaceName' = root\cimv2,'className' = Win32_OperatingSystem'.
VERBOSE: Operation 'Enumerate CimInstances' complete.
VERBOSE: Perform operation 'Enumerate CimInstances' with following parameters,
''namespaceName' = root\cimv2,'className' = Win32_BIOS'.
VERBOSE: Operation 'Enumerate CimInstances' complete.

BIOSSerial                                              ComputerName OSVersion
----------                                              ------------ ---------
VMware-56 4d 03 1c 3a c5 5f a3-d6 3c 01 92 aa e7 1d 45 localhost    10.0.14393


PS C:\> Get-XXSystemInfo -Computername localhost -verbose -Protocol dcom
VERBOSE: Attempting localhost on dcom
VERBOSE: Operation '' complete.
VERBOSE:   [+] Connected
VERBOSE: Perform operation 'Enumerate CimInstances' with following parameters,
''namespaceName' = root\cimv2,'className' = Win32_OperatingSystem'.
VERBOSE: Operation 'Enumerate CimInstances' complete.
VERBOSE: Perform operation 'Enumerate CimInstances' with following parameters,
''namespaceName' = root\cimv2,'className' = Win32_BIOS'.
VERBOSE: Operation 'Enumerate CimInstances' complete.

BIOSSerial                                              ComputerName OSVersion
----------                                              ------------ ---------
VMware-56 4d 03 1c 3a c5 5f a3-d6 3c 01 92 aa e7 1d 45 localhost    10.0.14393


PS C:\> Get-XXSystemInfo -Computername localhost -Protocol x
Get-XXSystemInfo : Cannot validate argument on parameter 'Protocol'. The argument
"x" does not belong to the set "Dcom,Wsman" specified by the ValidateSet attribute.
```

```
Supply an argument that is in the set and then try the command again.
At line:1 char:52
+ Get-XXSystemInfo -Computername localhost -Protocol x
+                                                    ~
    + CategoryInfo          : InvalidData: (:) [Get-XXSystemInfo], ParameterBindingV
   alidationException
    + FullyQualifiedErrorId : ParameterArgumentValidationError,Get-XXSystemInfo
PS C:\> Get-XXSystemInfo -Computername nope -verbose -Protocol dcom -TryOtherProtocol
VERBOSE: Attempting nope on dcom
PS C:\> TerminatingError(New-CimSession): "The running command stopped because the
preference variable "ErrorActionPreference" or common parameter is set to Stop:
The RPC server is unavailable. "
WARNING: Skipping nope due to failure to connect
VERBOSE: Attempting nope on wsman
PS C:\> TerminatingError(New-CimSession): "The running command stopped because the
preference variable "ErrorActionPreference" or common parameter is set to Stop:
The RPC server is unavailable. "
WARNING: Skipping nope due to failure to connect

PS C:\> Stop-Transcript
**********************
Windows PowerShell transcript end
End time: 20170623144314
**********************
```

Our Read-Through

Let's go through that transcript, and we'll tell you what should have been coming to mind for you at each step.

```
PS C:\> Get-XXSystemInfo -Computername localhost

BIOSSerial                                          ComputerName OSVersion
----------                                          ------------ ---------
VMware-56 4d 03 1c 3a c5 5f a3-d6 3c 01 92 aa e7 1d 45 localhost    10.0.14393
```

OK, this tells us that the command name is `Get-XXSysteminfo`, and it has a -Computername parameter. We don't know if it accepts just one value, or many, at this point. We can see what it produces, so we know we're going to have to query two CIM/WMI classes. We don't know what the module name is, but we could make one up if we needed to.

```
PS C:\> Get-XXSystemInfo -Computername localhost -verbose
VERBOSE: Attempting localhost on Wsman
VERBOSE: Operation '' complete.
VERBOSE:   [+] Connected
VERBOSE: Perform operation 'Enumerate CimInstances' with following parameters,
 ''namespaceName' = root\cimv2,'className' = Win32_OperatingSystem'.
VERBOSE: Operation 'Enumerate CimInstances' complete.
VERBOSE: Perform operation 'Enumerate CimInstances' with following parameters,
''namespaceName' = root\cimv2,'className' = Win32_BIOS'.
VERBOSE: Operation 'Enumerate CimInstances' complete.

BIOSSerial                                              ComputerName OSVersion
----------                                              ------------ ---------
VMware-56 4d 03 1c 3a c5 5f a3-d6 3c 01 92 aa e7 1d 45 localhost    10.0.14393
```

The above tells is that `[CmdletBinding()]` is in use, and that `Write-Verbose` is used.

```
PS C:\> Get-XXSystemInfo -Computername localhost -verbose -Protocol dcom
VERBOSE: Attempting localhost on dcom
VERBOSE: Operation '' complete.
VERBOSE:   [+] Connected
VERBOSE: Perform operation 'Enumerate CimInstances' with following parameters,
''namespaceName' = root\cimv2,'className' = Win32_OperatingSystem'.
VERBOSE: Operation 'Enumerate CimInstances' complete.
VERBOSE: Perform operation 'Enumerate CimInstances' with following parameters,
''namespaceName' = root\cimv2,'className' = Win32_BIOS'.
VERBOSE: Operation 'Enumerate CimInstances' complete.

BIOSSerial                                              ComputerName OSVersion
----------                                              ------------ ---------
VMware-56 4d 03 1c 3a c5 5f a3-d6 3c 01 92 aa e7 1d 45 localhost    10.0.14393
```

We now know that there are multiple protocols. Based on the verbose output above, at least Wsman and Dcom are supported. We can anticipate adding a `ValidateSet()` to only allow those two values, unless we encounter some more.

```
PS C:\> Get-XXSystemInfo -Computername localhost -Protocol x
Get-XXSystemInfo : Cannot validate argument on parameter 'Protocol'. The argument
"x" does not belong to the set "Dcom,Wsman" specified by the ValidateSet attribute.
Supply an argument that is in the set and then try the command again.
At line:1 char:52
+ Get-XXSystemInfo -Computername localhost -Protocol x
+                                                    ~
    + CategoryInfo          : InvalidData: (:) [Get-XXSystemInfo], ParameterBindingV
   alidationException
    + FullyQualifiedErrorId : ParameterArgumentValidationError,Get-XXSystemInfo
```

The above confirms that a `ValidateSet()` is going to be needed.

```
PS C:\> Get-XXSystemInfo -Computername nope -verbose -Protocol dcom -TryOtherProtocol
VERBOSE: Attempting nope on dcom
PS C:\> TerminatingError(New-CimSession): "The running command stopped because
the preference variable "ErrorActionPreference" or common parameter is set to
Stop: The RPC server is unavailable. "
WARNING: Skipping nope due to failure to connect
VERBOSE: Attempting nope on wsman
PS C:\> TerminatingError(New-CimSession): "The running command stopped because
the preference variable "ErrorActionPreference" or common parameter is set to
Stop: The RPC server is unavailable. "
WARNING: Skipping nope due to failure to connect
```

The forgoing suggests that we have the ability to recursively call our own function to try the other protocol. We'll need to build that into the error-handling routine.

Our Answer

As noted earlier, our code is in the downloadable samples, but here's a print version for your convenience:

Answer.ps1

```
Function Get-XXSystemInfo {
[CmdletBinding()]
param(
    [Parameter(
        Mandatory,
        ValueFromPipeline
    )]
    [string[]]$Computername,

    [Parameter()]
    [ValidateSet('Dcom', 'Wsman')]
    [string]$Protocol = 'Wsman',

    [Parameter()]
    [switch]$TryOtherProtocol
)
BEGIN {
    If ($Protocol -eq 'Dcom') {
        $cso = New-CimSessionOption -Protocol Dcom
    }
    else {
        $cso = New-CimSessionOption -Protocol Wsman
    }
}
PROCESS {
    ForEach ($comp in $computername) {
        Try {
        Write-Verbose "Attempting $comp on $protocol"
        $s = New-CimSession -ComputerName $comp -SessionOption $cso -EA Stop

        Write-Verbose "  [+] Connected"
        $os = Get-CimInstance -CimSession $s -ClassName Win32_OperatingSystem
        $bios = Get-CimInstance -CimSession $s -ClassName Win32_BIOS
        $props = @{
            ComputerName = $comp
            BIOSSerial   = $bios.serialnumber
            OSVersion    = $os.version
        }
        New-Object -TypeName PSObject -Property $props
        }
        Catch {
            Write-Warning "Skipping $comp due to failure to connect"
```

```
            if ($TryOtherProtocol) {
                If ($Protocol -eq 'Dcom') {
                    Get-XXSystemInfo -Protocol Wsman -Computername $comp
                }
                else {
                    Get-XXSystemInfo -Protocol Dcom -Computername $comp
                }
            }
        } #Catch
    } #ForEach
} #PROCESS
END {}
}
```

How'd You Do

If you were able to spot all of the major elements, and construct something at least vaguely like our solution, then we think you're probably "good to go" in terms of this book. If not, check out *PowerShell Scripting in a Month of Lunches* from Manning.com, and thoroughly re-read Part 1 of this book, to bring yourself up to speed.

We can't stress that enough: if you're not up to speed at this point, then you're not ready to proceed further in this book.

Part 2: Professional-Grade Toolmaking

In this Part, we're going to try and take your toolmaking skills a bit further. This is the stuff that sets the beginners apart from the real pros. We've constructed these chapters into a kind of storyline, so each one builds on what the previous ones taught. That said, the storyline here isn't tightly coupled, so feel free to dive in to whatever chapter seems of most interest or use to you. Because we're moving into Toolmaking areas that are more optional and as-you-need, you won't see "Your Turn" lab elements in every chapter - but that doesn't mean you shouldn't try and play along! Just follow along with your *own* code. However, when we include a "Your Turn" section, we obviously strongly suggest you follow along with that "lab."

Going Deeper with Parameters

You should already have a strong understanding of parameters inside Advanced Functions. But there's more to cover, and this is the time to do it. It turns out, you can do a *lot* with `Param()` blocks. And this isn't even "it;" we've got a whole chapter on *dynamic* parameters coming up, as well.

Parameter Position

PowerShell has always been okay with you using parameters positionally, rather than providing their name. For example, these two commands are equivalent:

```
Get-Service -Name BITS
Get-Service BITS
```

It is *hugely* important to understand why this works, so let's pull up the help for `Get-Service`:

```
Get-Service [-ComputerName <String[]>] [-DependentServices] -DisplayName
<String[]> [-Exclude <String[]>] [-Include <String[]>] [-RequiredServices]
[<CommonParameters>]

Get-Service [-ComputerName <String[]>] [-DependentServices] [-Exclude
<String[]>] [-Include <String[]>] [-InputObject <ServiceController[]>]
[-RequiredServices] [<CommonParameters>]

Get-Service [[-Name] <String[]>] [-ComputerName <String[]>]
[-DependentServices] [-Exclude <String[]>] [-Include <String[]>]
[-RequiredServices] [<CommonParameters>]
```

It's worth looking at the full help for this, which we don't want to reproduce here - hit up the online help page[18] if you don't have access to PowerShell, so you can follow along.

First, help files will usually list parameters *in positional order*. So in the above, the first parameter set - where `-Name` is defined - the `-Name` parameter is listed first. We can confirm that by scrolling down in the full help and looking at the details of the parameter:

[18] https://docs.microsoft.com/en-us/powershell/module/Microsoft.PowerShell.Management/Get-Service?view=powershell-5.1

```
PS C:\> Help Get-Service parameter Name

-Name <String[]>
    Specifies the service names of services to be retrieved. Wildcards are
    permitted. By default, this cmdlet gets all of the services on the
    computer.

    Required?                    false
    Position?                    0
    Default value                None
    Accept pipeline input?       True (ByPropertyName, ByValue)
    Accept wildcard characters?  false
```

Position 0 (zero) is first. It's worth noting that `-ComputerName`, which is listed first in the other two parameter sets, is *not positional*. That is, you *must* specify `-ComputerName` to use it; you can't just chuck computer names in someplace and expect PowerShell to figure it out. Its own details confirm this:

```
help get-service -Parameter Computername

-ComputerName <String[]>
    Gets the services running on the specified computers. The default is the
    local computer.

    Type the NetBIOS name, an IP address, or a fully qualified domain name
    (FQDN) of a remote computer. To specify the local computer, type the
    computer name, a dot (.), or localhost.

    This parameter does not rely on Windows PowerShell remoting. You can use
    the ComputerName parameter of Get-Service even if your computer is not
    configured to run remote commands.

    Required?                    false
    Position?                    named
    Default value                None
    Accept pipeline input?       True (ByPropertyName)
    Accept wildcard characters?  false
```

This is how PowerShell can tell that `Get-Service BITS` is meant to use the first parameter set. The only parameter with position 0, which can accept a string, is `-Name`, and it only exists in one parameter set, so that must be the one we meant to use. If you define multiple parameter sets, you can in theory have more than one parameter in position 0, but only if (a) each one accepts a different value type and (b) each one is unique to a separate parameter set.

If you don't specify a position for your parameters, PowerShell automatically numbers them all in whatever order they're listed. So consider this short function (we're not including these examples in the downloadable code, because they're not really intended to be executable):

```
function test {
    param(
        [string[]]$one,
        [int]$two,
        [switch]$three
    )
}

help test -Full
```

This yields the following parameter details:

```
PARAMETERS
    -one <string[]>

        Required?                   false
        Position?                   0
        Accept pipeline input?      false
        Parameter set name          (All)
        Aliases                     None
        Dynamic?                    false

    -three

        Required?                   false
        Position?                   Named
        Accept pipeline input?      false
        Parameter set name          (All)
        Aliases                     None
        Dynamic?                    false

    -two <int>

        Required?                   false
        Position?                   1
        Accept pipeline input?      false
        Parameter set name          (All)
        Aliases                     None
        Dynamic?                    false
```

The `$one` parameter is first, in position `0`; `$two` is second in position `1`, and `$three` is not positional because it's a switch. Switches can't be positional when you're relying on auto-generated position numbers. Also notice that the auto-generated help isn't very picky about the order in which those parameters are documented! You can disable this automatic behavior by adding `[CmdletBinding(PositionalBinding=$false)]` in front of your `Param()` block.

Now, let's specify a position for each:

```
function test {
    param(
        [Parameter(Position=1)]
        [string[]]$one,

        [Parameter(Position=2)]
        [int]$two,

        [Parameter(Position=3)]
        [switch]$three
    )
}

help test -Full
```

The results:

```
PARAMETERS
    -one <string[]>

        Required?                    false
        Position?                    1
        Accept pipeline input?       false
        Parameter set name           (All)
        Aliases                      None
        Dynamic?                     false

    -three

        Required?                    false
        Position?                    3
        Accept pipeline input?       false
        Parameter set name           (All)
        Aliases                      None
        Dynamic?                     false
```

```
    -two <int>

        Required?                    false
        Position?                    2
        Accept pipeline input?       false
        Parameter set name           (All)
        Aliases                      None
        Dynamic?                     false
```

We've now specified position numbers for each, letting us put them in whatever position they want, regardless of the order they're listed. And, we can *explicitly* assign a position to the switch parameter. In practice, this would be a bit awkward-looking to use:

```
test a b $true
```

With (in your mind) that `$true` being taken for the switch parameter (which won't actually work, by the way). You can't make switches positional - so there's no point assigning them a position.

Now, let's share some opinions. We *generally* don't declare positions for our parameters, because we tend to use our commands in scripts, and in our scripts we like to spell out all of our parameter names. Doing so makes them easier to follow in the future. However, *sometimes* we'll have a command where it just makes for easier reading to not have parameter names for certain parameters. In those cases *only*, we will declare a position number for those parameters, so that we don't have to rely on PowerShell making something up. That way, if we later expand the function and accidentally change the order in which our parameters are declared, we still get our *declared* positions, rather than PowerShell re-ordering them and messing us up. We *do not* tend to declare a position for *every* parameter, which we see some people do almost reflexively. We feel that doing so makes our parameter block unnecessarily cluttered, and it encourages positional parameter use - which in a script, is not the best possible practice most times.

It's worth noting that there *are* times when positional parameters make a ton of sense. Pester, the testing framework for PowerShell, is one such instance. It's `Should` keyword, for example, is just a PowerShell command. To make the end result more English-readable, Pester's creators chose to use positional parameters, and to use nonstandard command naming ("Should," versus "Should-Object" or some other verb-noun scheme). So there are definitely times when it's the right thing to do, but those tend to be edge cases.

Validation

Let's run through the whole series of validation attributes. We're just going to highlight these; the full documentation[19] has more details and examples, and we're not trying to recreate that here.

[19] https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.core/about/about_functions_advanced_parameters

The first three apply mainly to parameters already marked as Mandatory, allowing them to accept empty values of some kind:

- AllowNull() - allows the parameter to accept a null value
- AllowEmptyString() - allows the parameter to accept an empty string (“”)
- AllowEmptyCollection() - allows an array parameter to accept an empty collection

The remainder are more general-purpose:

- `ValidateCount(min,max)` - specifies a minimum and maximum number of values, in an array, that the parameter will accept
- `ValidateLength(min,max)` - specifies the maximum string length the parameter will accept. You can specify a minimum and maximum value, and if the parameter accepts a collection then this is applied to all members of the collection
- `ValidatePattern(pattern)` - specifies a regular expression that any string input must match in order to be accepted
- `ValidateRange(min,max)` - specifies a range of numeric values that any input must fall between, inclusive of the minimum and maximum specified
- `ValidateScript({script block})` - specifies a script block; within the script, uses `$_` to refer to the proposed value for the parameter, and return $true to accept it or $false to reject it
- `ValidateSet(val,val,val...)` - covered earlier, this specifies a set of legal values for the parameter
- `ValidateNotNull()` - the parameter will not accept null values
- `ValidateNotNullOrEmpty()` - the parameter will not accept null values or empty strings (“”)

Multiple Parameter Sets

This is one of the neatest, most effective, and most often screwed-up elements of PowerShell parameters. Have a look at the help for the old Get-WmiObject command[20] as an example. In it, you’ll see a default parameter set that uses the `-Class` parameter. As soon as you use that parameter, you’re locked into that parameter set. You can’t use `-Query`, because it appears in a different parameter set. Many parameters appear in all of the available parameter sets, but some are unique to a given set.

Here’s what people often do wrong: they’ll define some switch parameter, like “-UseAlternateCredential,” which exists only in a given parameter set. That set will also contain a mandatory “-Credential” parameter. The idea is, you specify the first parameter to push you into the parameter set, and then the parameter set forces you to also provide a credential. This isn’t a *great* design approach, and it certainly doesn’t fit in with PowerShell’s native patterns. Natively, you’d simply specify a “-Credential” parameter, and if someone ran the command without using it, then you just didn’t use it. You don’t, in other words, typically see PowerShell using a switch *simply to push the user into a*

[20] https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.management/get-wmiobject

parameter set. Instead, the parameters that are unique to a parameter set are typically related to one another in some way. For example, in `Get-WmiObject`, the `-Filter` parameter exists with `-Class` to help reduce the results you get back. But `-Filter` does not exist with `-Query`, because your query *could already contain a filter preposition*, making a filtering parameter redundant.

You assign a parameter to a parameter set by specifying a name for the set:

```
[Parameter(ParameterSetName="query")]
[string]$query
```

Any parameters given the same set name will also belong to that set; any parameters *given no set name at all* will belong to *all* sets. Each parameter can have only one ParameterSetName assigned *per Parameter attribute*. The following, however, is legal:

```
[Parameter(ParameterSetName="one")]
[Parameter(ParameterSetName="two")]
[string]$something
```

The `$something` parameter, here, would belong to both set "one" and set "two," but not any other sets which were defined. Again, any parameter assigned to *no* set will be implicitly included in *all* sets. And, it should go without saying but let's say it, each parameter set must have at least one unique parameter, which tells the shell you're using that set.

If you are using multiple parameter sets, use `Show-Command` to verify your parameters are grouped as expected. When you run the cmdlet, the graphical display will show a tab for each parameter set with the related parameters.

PowerShell dynamically tries to figure out which parameter set you're "in" by looking at the parameters you've used. This can sometimes be hard for it if you're using positional parameters whose values might legitimately line up with more than one parameter set (which is another reason we personally like to avoid positional parameters). In most cases, you'll also want to define a *default* parameter set, which is what PowerShell will try to use until it sees a parameter that forces it to consider a different set. You specify this in your `[CmdletBinding()]` attribute:

```
[CmdletBinding(DefaultParameterSetName="whatever")]
```

Parameter set names appear to be case-sensitive starting in Windows PowerShell 5.0. And as we move to a world of cross-platform scripting, watching your casing will be even more important.

In your code you can check for parameter values of use code like this to determine what parameter set is in use.

```
if ($PSCmdlet.ParameterSetName -eq "computer") {
`#do this
}
```

Value From Remaining Arguments

This is a bit of an odd duck that, honestly, you don't see much and we're not sure we've ever used. It basically says, "for this parameter, take all the values that haven't been assigned to other parameters, and dump 'em in."

```
[Parameter(ValueFromRemainingArguments=$True)]
[string]$Extras
```

Frankly, we think you are better off carefully planning all of your parameters and if you do so you should never need to use this attribute. It is definitely for rare, edge cases.

Help Message

The help message is intended to be a very short description of what the parameter wants. This is *mainly* available from the prompt that PowerShell creates when a mandatory parameter is omitted.

```
[Parameter(HelpMessage="Enter a computer name or IP")]
[string[]]$ComputerName
```

This is the message that will be displayed at the prompt if the user types `!?`. If you don't create comment based or external help, PowerShell will use this message when displaying help.

If you are marking a parameter as mandatory, we recommend you include a meaningful help message.

Alias

You can define a parameter alias using syntax like this:

```
[Alias("cn")]
[string[]]$Computername
```

Using a parameter alias is a handy technique if you have company jargon you expect people to use but need to follow PowerShell standards with a proper parameter name. Aliases also tend to be shorter but with improvements in tab-completion this doesn't seem as compelling as it once might have been. Be aware that aliases aren't easily discovered and may not show up in auto-generated help, especially for older PowerShell versions. You can specify them in comment-based help, or if you use the Platyps module, which we cover in the chapter on writing help, it will discover and document them accordingly.

More CmdletBinding

You should already know that `[CmdletBinding()]`, when added before your `Param()` block, enables common parameters. You should also know how it can enable the `-WhatIf` and `-Confirm` parameter by adding `SupportsShouldProcess` and `ConfirmImpact`; and earlier in this chapter we showed you how it can disable automatic positional parameter numbering and specify a default parameter set. It can do a bit more, as described in the official docs[21]:

- Specify a `HelpURI`, which must begin with http:// or https://, where the command's online documentation can be found
- Specify `SupportsPaging`, enabling the `-First-`, `-Skip`, and `-IncludeTotalCount` parameters. You must implement support for these; an example is included in the docs.

A Demonstration

We thought it might be useful for you to see a sample function that uses many of these concepts and techniques. The reason for including them in *your* work is not to show off but to make your tool easier for someone to use, perhaps even yourself, and to catch and potential problems at the beginning before your command starts doing anything.

The complete function, `Get-DiskCheck`, is in the chapter download file. Here is the relevant parameter section.

[21] https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_functions_cmdletbindingattribute?view=powershell-5.1

Get-DiskCheck

```
Function Get-DiskCheck {
[cmdletbinding(DefaultParameterSetName = "name")]
Param(
    [Parameter(
        Position = 0,
        Mandatory,
        HelpMessage = "Enter a computer name to check",
        ParameterSetName = "name",
        ValueFromPipeline
    )]
    [Alias("cn")]
    [ValidateNotNullorEmpty()]
    [string[]]$Computername,

    [Parameter(
        Mandatory,
        HelpMessage = "Enter the path to a text file of computer names",
        ParameterSetName = "file"
    )]
    [ValidateScript( {
            if (Test-Path $_) {
                $True
            }
            else {
                Throw "Cannot validate path $_"
            }
        })]
    [ValidatePattern("\.txt$")]
    [string]$Path,

    [ValidateRange(10, 50)]
    [int]$Threshhold = 25,

    [ValidateSet("C:", "D:", "E:", "F:")]
    [string]$Drive = "C:",

    [switch]$Test
)
```

We'll admit that we are fudging a bit on best practices with this function but that is only for the sake of demonstration.

The function has two primary parameter sets. One for computer names and one for the path to a text file with computer names. The mandatory `-computername` parameter also has an alias of CN and a validation to make sure the value is not null or an empty string.

The `-Path` parameter has multiple validation tests. The path value must end in .txt and it must exist. This is what ValidateScript is testing. Normally, you can just use the validation attribute but you can control the output if there is an error. In this scenario if the filename doesn't pass `Test-Path` the scriptblock throws an exception with our text. We could have written it like this:

```
[ValidateScript({Test-Path $_)}
```

But if it failed the user would see the default exception message which may not be helpful or perhaps overly verbose. With our approach if the validation fails the error message is a bit more succinct.

The remaining parameters belong to both parameter sets and we're using a few validation attributes. One nice benefit of using `ValidateSet` is that you can cycle through the possible values with tab-completion.

When you ask for help on the function you can see the two different parameter sets.

```
SYNTAX
    Get-DiskCheck [-Computername] <string[]> [-Threshold <int>]
    [-Drive <string> {C: | D: | E: | F:}] [-Test] [<CommonParameters>]

    Get-DiskCheck -Path <string> [-Threshold <int>] [-Drive <string>
    {C: | D: | E: | F:}] [-Test] [<CommonParameters>]
```

Because the function lacks comment-based help, PowerShell displays the values for `-Drive` from the ValidateSet() attribute. And when looking at parameter details the help messages are also used.

```
PARAMETERS
    -Computername <string[]>
        Enter a computer name to check

        Required?                    true
        Position?                    0
        Accept pipeline input?       true (ByValue)
        Parameter set name           name
        Aliases                      cn
        Dynamic?                     false

    -Drive <string>

        Required?                    false
```

```
        Position?                    Named
        Accept pipeline input?       false
        Parameter set name           (All)
        Aliases                      None
        Dynamic?                     false

    -Path <string>
        Enter the path to a text file of computer names

        Required?                    true
        Position?                    Named
        Accept pipeline input?       false
        Parameter set name           file
        Aliases                      None
        Dynamic?                     false

    -Test

        Required?                    false
        Position?                    Named
        Accept pipeline input?       false
        Parameter set name           (All)
        Aliases                      None
        Dynamic?                     false

    -Threshold <int>

        Required?                    false
        Position?                    Named
        Accept pipeline input?       false
        Parameter set name           (All)
        Aliases                      None
        Dynamic?                     false
```

We'll let you test out the function yourself to see how the different parameter techniques work.

Your Turn

In the first part of the book we had you work on a function to get computer status information from a number of CIM classes. The function took a computer name as a parameter. But `Get-CimInstance` can also use CimSessions. Why not have a function that can also accept CIMSessions?

Start Here

Start with the last iteration of the function which you can find in the downloads.

Get-TMComputerStatus.ps1

```
Function Get-TMComputerStatus {
 <#
.SYNOPSIS
Get computer status information.

.DESCRIPTION
This command retrieves system information from one or more remote computers
using Get-CimInstance. It will write a summary object to the pipeline for
each computer. You also have the option to log errors to a text file.

.PARAMETER Computername
The name of the computer to query. This parameter has aliases of
CN, Machine and Name.

.PARAMETER ErrorLog
The path to the error text file. This is not implemented yet.

.PARAMETER ErrorAppend
Append errors to the error file. This is not implemented yet.

.EXAMPLE
PS C:\> Get-TMComputerStatus -computername SRV1

Computername : SRV1
TotalMem     : 33080040
FreeMem      : 27384236
Processes    : 218
PctFreeMem   : 82.7817499616083
Uptime       : 11.06:23:52.7176115
CPULoad      : 2
PctFreeC     : 54.8730920184876

Get the status of a single computer

.EXAMPLE
PS C:\> Get-Content c:\work\computers.txt | Get-TMComputerStatus |
Export-CliXML c:\work\data.xml
```

```
Pipe each computer name from the computers.txt text file to this
command. Results are immediately exported to an XML file using
Export-CliXML.

.INPUTS
System.String

.NOTES
Version     : 1.0.0

.LINK
Get-CimInstance
#>
[cmdletbinding()]
Param(
    [Parameter(ValueFromPipeline, Mandatory)]
    [ValidateNotNullorEmpty()]
    [ValidatePattern("^\w+$")]
    [Alias("CN", "Machine", "Name")]
    [string[]]$Computername,
    [string]$ErrorLog,
    [switch]$ErrorAppend
)

BEGIN {
    Write-Information "Command = $($myinvocation.mycommand)" -Tags Meta
    Write-Information "PSVersion = $($PSVersionTable.PSVersion)" -Tags Meta
    Write-Information "User = $env:userdomain\$env:username" -tags Meta
    Write-Information "Computer = $env:computername" -tags Meta
    Write-Information "PSHost = $($host.name)" -Tags Meta
    Write-Information "Test Date = $(Get-Date)" -tags Meta
    Write-Verbose "Starting $($myinvocation.mycommand)"
}

PROCESS {
  foreach ($computer in $Computername) {
     Write-Verbose "Querying $($computer.toUpper())"

     Try {
         $params = @{
          Classname    = "Win32_OperatingSystem"
          Computername = $computer
          ErrorAction  = "Stop"
```

```
        }
        $OS = Get-CimInstance @params

        $params.ClassName = "Win32_Processor"
        $cpu = Get-CimInstance @params

        $params.className = "Win32_logicalDisk"
        $vol = Get-CimInstance @params -filter "DeviceID='c:'"

        $OK = $True
    }
    Catch {
        $OK = $False
        $msg = "Failed to get system information from $computer. $($_.Exception.Mes\
sage)"
        Write-Warning $msg
        if ($ErrorLog) {
            Write-Verbose "Logging errors to $ErrorLog. Append = $ErrorAppend"
           "[$(Get-Date)] $msg" | Out-File -FilePath $ErrorLog -Append:$ErrorAppend
        }
    }
    if ($OK) {
        #only continue if successful
        [pscustomobject]@{
          Computername = $os.CSName
          TotalMem     = $os.TotalVisibleMemorySize
          FreeMem      = $os.FreePhysicalMemory
          Processes    = $os.NumberOfProcesses
          PctFreeMem   = ($os.FreePhysicalMemory/$os.TotalVisibleMemorySize)*100
          Uptime       = (Get-Date) - $os.lastBootUpTime
          CPULoad      = $cpu.LoadPercentage
          PctFreeC     = ($vol.FreeSpace/$vol.size)*100
        }
    } #if OK
   } #foreach $computer
}
END {
   Write-Verbose "Ending $($myinvocation.mycommand)"
}
} #Get-TMComputerStatus
```

Your Task

First, add whatever enhancements you'd like based on this chapter. Then add a second parameter set to accept a CIMSession. Save the new function to the Toolmaking module. Don't forget to update your help!

Our Take

Revised Get-TMComputerStatus

```
Function Get-TMComputerStatus {
 <#
.SYNOPSIS
Get computer status information.

.DESCRIPTION
This command retrieves system information from one or more remote computers
using Get-CimInstance. It will write a summary object to the pipeline for
each computer. You also have the option to log errors to a text file.

.PARAMETER Computername
The name of the computer to query. This parameter has aliases of
CN, Machine and Name.

.PARAMETER ErrorLog
The path to the error text file. This is not implemented yet.

.PARAMETER ErrorAppend
Append errors to the error file. This is not implemented yet.

.EXAMPLE
PS C:\> Get-TMComputerStatus -computername SRV1

Computername : SRV1
TotalMem     : 33080040
FreeMem      : 27384236
Processes    : 218
PctFreeMem   : 82.7817499616083
Uptime       : 11.06:23:52.7176115
CPULoad      : 2
PctFreeC     : 54.8730920184876

Get the status of a single computer
```

```
.EXAMPLE
PS C:\> Get-Content c:\work\computers.txt | Get-TMComputerStatus |
Export-CliXML c:\work\data.xml

Pipe each computer name from the computers.txt text file to this
command. Results are immediately exported to an XML file using
Export-CliXML.

.INPUTS
System.String

.NOTES
Version     : 1.0.0

.LINK
Get-CimInstance
#>
[cmdletbinding()]
Param(
    [Parameter(ValueFromPipeline, Mandatory)]
    [ValidateNotNullorEmpty()]
    [ValidatePattern("^\w+$")]
    [Alias("CN", "Machine", "Name")]
    [string[]]$Computername,
    [string]$ErrorLog,
    [switch]$ErrorAppend
)

BEGIN {
    Write-Information "Command = $($myinvocation.mycommand)" -Tags Meta
    Write-Information "PSVersion = $($PSVersionTable.PSVersion)" -Tags Meta
    Write-Information "User = $env:userdomain\$env:username" -tags Meta
    Write-Information "Computer = $env:computername" -tags Meta
    Write-Information "PSHost = $($host.name)" -Tags Meta
    Write-Information "Test Date = $(Get-Date)" -tags Meta
    Write-Verbose "Starting $($myinvocation.mycommand)"
}

PROCESS {
  foreach ($computer in $Computername) {
     Write-Verbose "Querying $($computer.toUpper())"
```

```
    Try {
        $params = @{
         Classname    = "Win32_OperatingSystem"
         Computername = $computer
         ErrorAction  = "Stop"
        }
        $OS = Get-CimInstance @params

        $params.ClassName = "Win32_Processor"
        $cpu = Get-CimInstance @params

        $params.className = "Win32_logicalDisk"
        $vol = Get-CimInstance @params -filter "DeviceID='c:'"

        $OK = $True
    }
    Catch {
        $OK = $False
        $msg = "Failed to get system information from $computer. $($_.Exception.Mes\
sage)"
        Write-Warning $msg
        if ($ErrorLog) {
            Write-Verbose "Logging errors to $ErrorLog. Append = $ErrorAppend"
           "[$(Get-Date)] $msg" | Out-File -FilePath $ErrorLog -Append:$ErrorAppend
        }
    }
    if ($OK) {
        #only continue if successful
        [pscustomobject]@{
          Computername = $os.CSName
          TotalMem     = $os.TotalVisibleMemorySize
          FreeMem      = $os.FreePhysicalMemory
          Processes    = $os.NumberOfProcesses
          PctFreeMem   = ($os.FreePhysicalMemory/$os.TotalVisibleMemorySize)*100
          Uptime       = (Get-Date) - $os.lastBootUpTime
          CPULoad      = $cpu.LoadPercentage
          PctFreeC     = ($vol.FreeSpace/$vol.size)*100
        }
    } #if OK
   } #foreach $computer
}
END {
   Write-Verbose "Starting $($myinvocation.mycommand)"
```

```
}
} #Get-TMComputerStatus
```

Using a CIMSession actually works in our favor for this function since it is making multiple calls to `Get-CimInstance`. When using a computer name this means there is extra overhead, and time, to setup each connection. Using an existing CIMSession speeds up the whole process. The new parameter belongs to its own parameter set. The parameters that don't have a specified parameter set belong to all sets.

We revised the code to create a set of temporary CIMSessions to each computer. The main code was modified to use the sessions. We're also keeping track of temporary sessions so they can be removed at the end of the function. We don't want to remove a user's existing CIMSessions.

Let's Review

What did you learn?

1. True or False: You can only have one validation attribute per parameter?
2. How many parameter sets can you define in a function?
3. Can you have multiple parameters with `Position = 0` ?

Review Answers

Did you come up with these answers?

1. False. You can have as many as make sense and the value must pass all of them.
2. You can define as many as you need. However, in our experience if you start running into more than 4 or 5 parameter sets, you might need to re-think your design strategy.
3. Yes, but only if they are in different parameter sets. Even then you will need to test this thoroughly.

Advanced Function Tips and Tricks

This chapter will be a kind of loose round-up of other cool, advanced, and useful things you can do with an advanced function.

Defining an Alias

You already know that PowerShell can define aliases for commands, right? Functions are commands, and they can have aliases. But you don't need to run `New-Alias` to make them–you can define an alias right in the function itself:

```
function Get-Foo {
 [cmdletbinding()]
 [Alias ("gf")]
 Param()
}
```

This defines an alias `gf`, and PowerShell will even display that in its auto-generated help for the command.

Specify Output Type

When you are creating a function, you are most likely creating it work with other PowerShell commands. You might write a command that will be piped to another PowerShell object. To help the next person, it helps to document what type of object your command is writing to the pipeline. One way to accomplish this is to specify an`[OutputType()]`.

```
Function Get-Runtime {
    [CmdletBinding()]
    [OutputType([timespan])]
    Param([string]$Name)
    ...
}
```

When a user looks at full help for this function, they will see `System.TimeSpan` under `OUTPUTS`. You can enter a .NET type name as we've done here, or as a string.

```
Function Get-Runtime {
    [CmdletBinding()]
    [OutputType("System.Timespan")]
    Param([string]$Name)
    ...
}
```

If you are writing a custom object to the pipeline of your own creation, perhaps from a class or something with a typename that you defined, we recommend you enter the output type as a string.

```
Function Get-ThatThing {
    [CmdletBinding()]
    [OutputType("MyThing")]
    Param([string]$Name)
    ...
}
```

If you had used `[MyThing]`, PowerShell will most likely try to find that type and complain.

You are allowed to specify multiple types.

```
Function Set-ThatThing {
    [CmdletBinding()]
    [OutputType("None","MyThing")]
    Param([string]$Name,[switch]$Passthru)
    ...
}
```

In this hypothetical function, it is assumed it doesn't right anything to the pipeline because of the `-Passthru` parameter. This command's output could nothing, or it could be a `MyThing` object. What about parameter sets?

You can have multiple `OutputType` attributes.

```
Function Get-SomethingElse {
    [CmdletBinding(DefaultParameterSetName="Name")]
    [OutputType("System.String",ParameterSetName = "Other")]
    [OutputType("MyThing",ParameterSetName = "Name")]
    Param()
    ...
}
```

The help output will show both types but there is no indication about parameter sets.

Adding Labels

We have probably mentioned this elsewhere in the book and you've likely seen it in plenty of code examples. Consider this snippet of code:

```
...
        $drives = Get-CimInstance @params
        Write-Verbose "[PROCESS] CIM query complete"

        ForEach ($drive in $drives) {
            if ($drive.DriveType -ne 5) {
                [pscustomobject]@{
                    ComputerName = $comp
                    Letter       = $drive.deviceid
                    Size         = $drive.size
                    Free         = $drive.freespace
                }
            }
        }
    }
}
}
```

It definitely helps that the code has been formatted and indented. But can you tell if you have enough closing curly brackets? Or too many? Yes, there are things you do in your editor to help, but something like this can help.

```
...
        $drives = Get-CimInstance @params
        Write-Verbose "[PROCESS] CIM query complete"

        ForEach ($drive in $drives) {
            if ($drive.DriveType -ne 5) {
                [pscustomobject]@{
                    ComputerName = $comp
                    Letter       = $drive.deviceid
                    Size         = $drive.size
                    Free         = $drive.freespace
                } #customobject
            } #if drivetype <> 5
        } #foreach drive
    } #foreach computer
```

```
} #process
} #close Get-DriveInfo function
```

Now there's no confusion.

Use Your Command Name Programmatically

We've mentioned how helpful it can be to incorporate Verbose messages into your PowerShell commands. We find it helpful to know what command is starting and when it is ending. This is particularly helpful if your command is calling other commands. The verbose messaging can help you keep track of potentially complicated command flow. One thing you can do to make life easy for you is to take advantage of the built-in `$MyInvocation` object. You can use this to programmatically capture the name of your command. Don't hard code in your command name, because you might change it. Instead, use code like this:

```
Begin {
    Write-Verbose "[$((Get-Date).TimeofDay) BEGIN  ] Starting $($myinvocation.mycomm\
and)"
} #begin
Process {
    Write-Verbose "[$((Get-Date).TimeofDay) PROCESS] Processing $computername"
    ...
}
End {
    Write-Verbose "[$((Get-Date).TimeofDay) END    ] Ending $($myinvocation.mycomman\
d)"
} #end
```

When run with verbose output you might see something like:

```
PS C:\> Get-SystemData -Verbose
VERBOSE: [09:26:41.6459721 BEGIN  ] Starting Get-SystemData
VERBOSE: [09:26:41.6469464 PROCESS] Getting system data for SRV1
VERBOSE: [09:26:41.6479456 BEGIN  ] Starting Get-OS
VERBOSE: [09:26:41.6489453 PROCESS] Processing SRV1
VERBOSE: [09:26:53.9988218 END    ] Ending Get-OS
VERBOSE: [09:26:54.0007555 PROCESS] Updating database
VERBOSE: [09:26:56.3576237 END    ] Ending Get-SystemData
PS C:\>
```

We used the same type of code to the main function and the run it calls. You should get the idea that this would make a great snippet.

ArgumentCompleter

One of the nice benefits of using the `[ValidateSet()]` attribute is that PowerShell will use it for auto completion. But what about other situations? One possibility is to use an `ArgumentCompleter`. Here's a demo function.

Get-EventLogDetail.ps1

```
Function Get-EventLogDetail {
    [cmdletbinding()]
    Param(
        [Parameter(Position = 0)]
        [ArgumentCompleter({(Get-Eventlog -List).log})]
        [string]$LogName
    )
    Write-Verbose "Getting $LogName log details"

    $params = @{
        ClassName = "win32_nteventlogfile"
        filter = "filename='$LogName'"
        ErrorAction = "Stop"
    }

    Try {
        Get-CimInstance @params |
        Select-Object -Property LogFileName, Name, FileSize, MaxFileSize,
        LastModified, NumberOfRecords,@{Name="PctUsed";
        Expression={[math]::Round(($_.filesize/$_.maxfilesize)*100,2)}}
    }
    Catch {
        Write-Error $_
    }
}
```

The `LogName` parameter has an `ArgumentCompleter`. The results of the code in the scriptblock will be passed as possible parameter values. Ideally, this code will run very quickly. In this case, the user will get a list of event logs. This doesn't prevent them from manually entering a log name.

Dynamic Parameters

Dynamic parameters are ones that are only available under certain circumstances, such as when your command is being used from a particular drive (say, a FileSystem drive, but not a Registry drive). You see this with the `-Encoding` parameter of `Get-Content`; the parameter won't work if you're focused on anything but a FileSystem drive at the time. Dynamic parameters can also be enabled by using another, static parameter of your command, in which case they become something like a "child parameter set." But dynamic parameters aren't necessarily listed in help like a static parameter, meaning they're harder for people to find and make use of. When possible, you want to avoid these, and only use them when they absolutely make sense and accomplish something you can't do in other ways.

When someone asks for help on your command, PowerShell will try to evaluate your dynamic parameters to see if they're applicable, and only show them if so. That's why they can be harder to discover - if they're not valid at the time, people won't see them in help. It is also possible to create dynamic parameters that only appear if another parameter is specified which makes them even *harder* to discover.

For example, suppose you have a parameter named `-UseAlternateLanguage`, and you're thinking you also want to add a dynamic parameter named `-LanguageToUse`. If someone specifies the first, they'll be able to use the second to pick a language. That's probably not a great use of dynamic parameters. Instead, you'd probably just pick a default language, and offer `-LanguageToUse` if someone wanted to use something different. If they didn't specify it, you'd use the default. This eliminates the need for a more complex parameter arrangement.

That's actually a good general rule to follow: try not to use one parameter simply to enable another. Instead, default to a sensible value and provide a single parameter to override that value.

Here's another example: suppose you write a command that will provision new users. You always need certain information, like their name and department. But sometimes, users will need to be provisioned in the company accounting software, where you'll also need to know their approver ID and spending limit. You might consider accomplishing that with a status `-AddToAccounting` parameter, which in turn enables dynamic parameters for `-ApproverID` and `-SpendingLimit`. However, you could accomplish something similar simply by having those latter two become mandatory parameters of their own parameter set. So your command has two parameter sets: one with the accounting stuff, and one without. Both would show up in help, making them more easily discoverable, and making their relationship more obvious.

This isn't to say that dynamic parameters are never appropriate, of course, because otherwise they probably wouldn't even be a thing. And this isn't even to say that using them is *rare*. It's just that we see a lot of people making poor parameter design decisions because they think, "well, dynamic parameters are a thing, and I should clearly be using them, because shiny." Try and avoid that. if you *can* accomplish your need with something simpler, do.

Declaring Dynamic Parameters

Here's a basic dynamic parameter declaration:

```
Param(
    [ValidateSet("User","Administrator","Guest")]
    [string]$UserLevel,
)

DynamicParam {
    If ($UserLevel -eq "Administrator") {
        # create an $AdminType parameter
    }
}
```

Notice first that `DynamicParam` is a *new and distinct construct* from the regular `Param` block. In it, you use an `If` construct to decide if the dynamic parameter is currently appropriate. In fact, you need to have some sort of condition evaluation to determine if the dynamic parameter needs to be defined. If you are basing your condition on another parameter, be aware your dynamic parameter won't be available until the user specifies the other parameter. And no, you shouldn't try to define a dynamic parameter that relies on another dynamic parameter.

In this example, the dynamic parameter will never show up in help, because when viewing the help no other parameter will have a value assigned, and so the condition will never be met. If the condition is met, then you manually - via code - create the parameter. The creation code isn't hard, but it's a bit laborious - you're essentially going to programmatically create .NET Framework objects to generate and attach new parameters.

1. You create any parameter attributes you plan to use, and add them to a collection;
2. You create the main parameter and attach the attributes;
3. You return the created parameter;

It's a bit more complex than that, so let's do some code (this isn't in the downloadable code because it's really not cut-and-pasteable; this is something you should be typing into your own functions, not reusing as boilerplate from us):

Create Attributes

You will start with something like this:

```
$attr = New-Object System.Management.Automation.ParameterAttribute
$attr.HelpMessage = "Enter admin type"
$attr.Mandatory = $true
$attr.ValueFromPipelineByPropertyName = $true
```

Other properties include:

- ParameterSetName
- Position
- ValueFromPipeline
- ValueFromRemainingArguments

Create an Attribute Collection

Next step:

```
$attrColl = New-Object System.Collections.ObjectModel.Collection[System.Attribute]
$attrColl.Add($attr)
```

Remember, each parameter can have one, and only one, attribute collection.

Create the Parameter

Here we go:

```
$param = New-Object System.Management.Automation.RuntimeDefinedParameter`
('AdminType',[string],$attrColl)
```

Really sorry about the word-wrapping there - it's unavoidable with a line that long. Remember, the backslash doesn't "exist" in the code, it' a line-wrap character here in the PDF book. We've created a new `-AdminType` parameter, which will accept a `String` object, and attached our attribute collection. But we're not quite done:

```
$dict = New-Object System.Management.Automation.RuntimeDefinedParameterDictionary
$dict.Add('AdminType',$param)
return $dict
```

That `return` keyword is what "sends" our new, dynamic parameter to PowerShell.

The Whole Picture

Here's the whole example, which *is* in our downloadable code samples for this chapter:

```
Param(
  [string]$UserLevel
)
DynamicParam {
  If ($UserLevel -eq "Administrator") {
    # create an $AdminType parameter
    $attr = New-Object System.Management.Automation.ParameterAttribute
    $attr.HelpMessage = "Enter admin type"
    $attr.Mandatory = $true
    $attr.ValueFromPipelineByPropertyName = $true
    $attrColl = New-Object System.Collections.ObjectModel.Collection[System.Attribut\
e]
    $attrColl.Add($attr)
    $param = New-Object System.Management.Automation.RuntimeDefinedParameter('AdminT\
ype',[string],$attrColl)
    $dict = New-Object System.Management.Automation.RuntimeDefinedParameterDictionary
    $dict.Add('AdminType',$param)
    return $dict
    }
}
```

Yup, that's a lot. And *each* `DynamicParam` you define will need to do that same sequence of events.

Using Dynamic Parameters

Dynamic parameters won't show up as "normal" variables like a static parameter will. Instead, you'd access them like this:

```
if ($PsBoundParameters.ContainsKey('AdminType')) {
    Write-Verbose "Admin type $($PsBoundParameters.AdminType)"
}
```

You'll find another really excellent walk-through at PowerShellMagazine[22], if you're interested.

Let's Review

Using dynamic parameters is certainly for edge cases. If you are like us you'll have to review the documentation to remember how to implement. But, let's see if anything from this chapter sunk in.

1. What are some of the drawbacks to using dynamic parameters?
2. What type of object do you need to create?
3. What might be an alternative to using a dynamic parameter?

Review Answers

And our take on the answers:

1. They are hard to implement and difficult for an end user to discover.
2. System.Management.Automation.ParameterAttribute.
3. Parameter sets

[22] http://www.powershellmagazine.com/2014/05/29/dynamic-parameters-in-powershell/

Writing Full Help

You should already know how you can add comment-based help to your tools. Typically you would create help documentation and insert it into each command. But there are some downsides to this approach:

- It can be particularly prone to errors, especially if you get the syntax wrong.
- It can be time consuming to write.
- If you need to update, you need to modify the script file itself which might lead to even more work verifying you didn't break anything in the process.
- If you need to provide help in another language, comment-base help becomes a big obstacle.

And just so you know, PowerShell itself doesn't provide help to you via comment-based help. The big boys and girls at Microsoft create special external help that they ship with their modules. You can, and should, do the same thing.

There's been a feeling for some time that comment-based help was "easier," both in terms of writing, and because it doesn't create external files that you also have to distribute. We say, "rubbish," at least now. As we'll show you, it's just as easy to create, and if you're *properly building and distributing your modules* then it's no longer harder to distribute. And it's easier to keep updated.

External Help

Typical commands such as `Get-Service` have their help content stored in special type of XML file. The file is written in an XML dialect known as MAML (Microsoft Assistance Markup Language). Use `Get-Command` to find the name of the help file.

```
PS C:\> Get-Command Get-Service | Select HelpFile

HelpFile
--------
Microsoft.PowerShell.Commands.Management.dll-Help.xml
```

Because help from Microsoft is localized, or written in your language, you'll find this file in `$pshome\en-us` where the subdirectory (en-us) is your localized language (for example, en-uk would be English, United Kingdom). The XML file will contain help for *all* commands in the designated module. Here's a taste of what that looks like.

```
Get-Command Get-Service |
Select-Object @{Name="Path";Expression={Join-Path $pshome\en-us $_.helpfile}} |
Get-Content -Head 30
```

Which should show something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<helpItems xmlns="http://msh" schema="maml">
  <!-- Updatable Help Version 5.0.7.0 -->
  <command:command xmlns:maml="http://schemas.microsoft.com/maml/2004/10"
xmlns:command="http://schemas.microsoft.com/ma
ml/dev/command/2004/10" xmlns:dev="http://schemas.microsoft.com/maml/dev/
2004/10" xmlns:MSHelp="http://msdn.microsoft.co
m/mshelp">
    <command:details>
      <command:name>Add-Computer</command:name>
      <maml:description>
        <maml:para>Add the local computer to a domain or workgroup.
</maml:para>
      </maml:description>
      <maml:copyright>
        <maml:para />
      </maml:copyright>
      <command:verb>Add</command:verb>
      <command:noun>Computer</command:noun>
      <dev:version />
    </command:details>
    <maml:description>
      <maml:para>The Add-Computer cmdlet adds the local computer or remote
computers to a domain or workgroup, or moves
them from one domain to another. It also creates a domain account if the
computer is added to the domain without an account.</maml:para>
      <maml:para>You can use the parameters of this cmdlet to specify an
organizational unit (OU) and domain controller or to perform an unsecure
join.</maml:para>
      <maml:para>To get the results of the command, use the Verbose and
PassThru parameters.</maml:para>
    </maml:description>
    <command:syntax>
      <command:syntaxItem>
        <maml:name>Add-Computer</maml:name>
        <command:parameter required="true" variableLength="false"
globbing="false" pipelineInput="false" position="1" aliases="DN,Domain">
```

```
          <maml:name>DomainName</maml:name>
          <maml:description>
            <maml:para>Specifies the domain to which the computers are
added. This parameter is required when adding the  computers to a domain.
</maml:para>
          </maml:description>
          <command:parameterValue required="true" variableLength="false">
String</command:parameterValue>
```

If you are like us, you are thinking "Oh, dear God in heaven what have I gotten myself into?" This is admittedly nasty-looking stuff and not for the faint of heart. In fact, a couple of years ago, we'd launch into a description of all the GUI-based tools you can use to create that XML, simply by laboriously copying and pasting your help content into said tool.

But don't run away. Things got better.

Using Platyps

Microsoft has an open source project on GitHub called Platyps[23]. The project's goal is to make it easier to generate external (i.e. MAML-based XML) help. This is accomplished through a set of commands that analyze your module to generate a set of Markdown help files which in turn can be used to generate external help. The Platyps commands are packaged as a module which you can install from the PowerShell Gallery:

```
Install-Module platyps
```

We won't go through every command in the module, but we will walk you through the process.

Like most open source projects, Platyps is in a constant state of development. There may be new features added after this chapter was written or new bugs introduced. If you encounter problems we encourage you to use the Issues section on project's GitHub repository.

Generate Markdown

The first step in the process is to generate a set of *Markdown* documents. If you are not familiar with it, Markdown is way to define what a document looks like, kind of like HTML, but a billion times easier (in fact, this book's source is written in Markdown). Don't worry, you don't need to understand much about Markdown, and what you do need to know you'll pick up quickly. One of the added benefits with this intermediate step is that you end up with set of help files that are

[23] https://github.com/powershell/platyps

formatted nicely for a web browser (Markdown documents also display great in GitHub, if you're hosting your code there). If you take a few minutes to look at the Markdown documents for Platyps at https://github.com/PowerShell/platyPS/tree/master/docs[24], you'll see what we mean. You could setup a web site with the help documents for your tool. Anyway, this isn't a chapter on Markdown so let's move on.

To get started, change location in PowerShell to the root folder of your module. We're assuming this directory has your .psm1 and .psd1 files. You will want to have a separate folder for your Markdown documents. We typically use Docs. For the sake of our demonstration we're going to use a copy of a module Jeff wrote for storing PSCredential objects in a json file.

```
PS C:\PSJsonCredential> mkdir Docs

    Directory: C:\PSJsonCredential


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        1/10/2020   5:09 PM                Docs
```

If you think you will be creating language specific versions of help, then create a separate documents folder for each language.

One of the great benefits of Platyps is that if you already have comment-based help, it will be used to generate the initial Markdown file. Once you have created external help, then you can delete the comment-based help from your files. Otherwise, there's no need to pre-generate any comment-based help. Let the Platyps commands do it for you.

The first cmdlet you'll use is `New-MarkdownHelp`. This command will create a Markdown help document for every command in your module. You should first load your module into your PowerShell session. Because the folder we are working with is not in one of the locations specified in `$env:PSModulepath`, we'll explicitly import the module.

```
PS C:\PSJsonCredential> import-module .\PSJsonCredential
```

Now we can create new Markdown help.

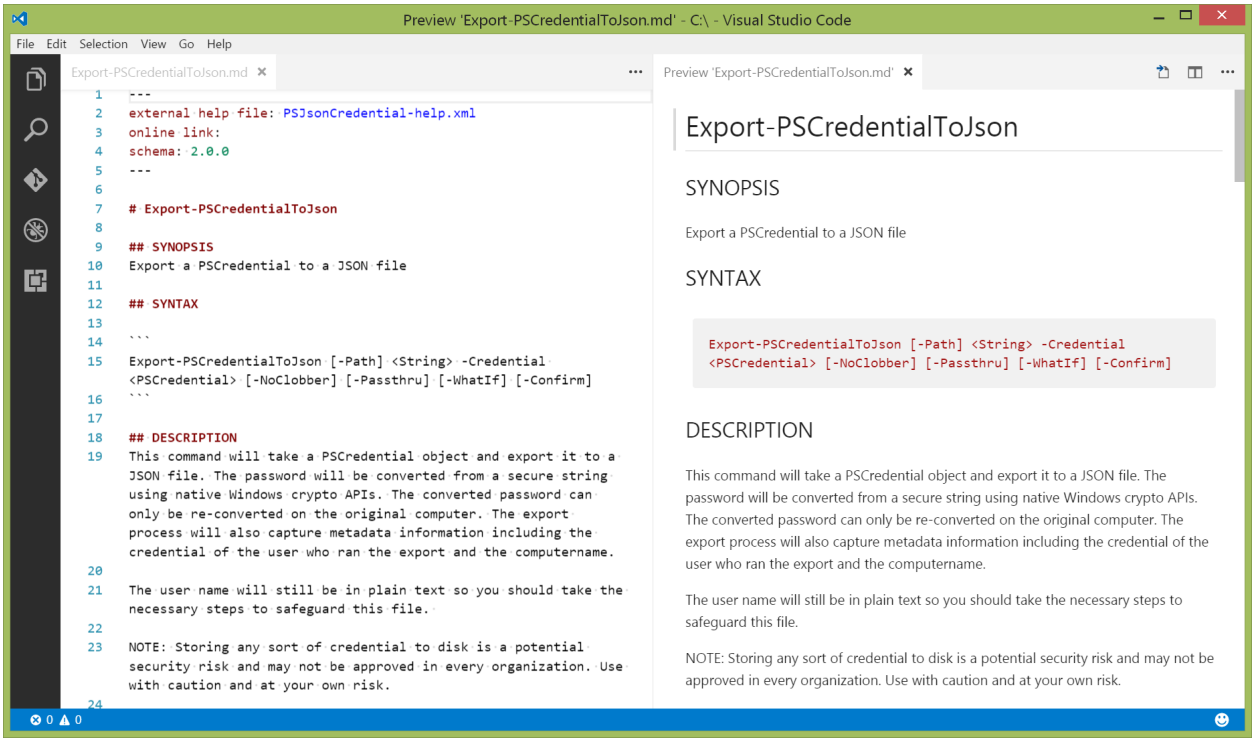
[24] https://github.com/PowerShell/platyPS/tree/master/docs

```
PS C:\PSJsonCredential> New-Markdownhelp -Module PSJsonCredential `
-OutputFolder .\Docs\ -withModulePage


    Directory: C:\PSJsonCredential\Docs


Mode             LastWriteTime       Length Name
----             -------------       ------ ----
-a----     1/10/2020   5:29 PM         2136 Export-PSCredentialToJson.md
-a----     1/10/2020   5:29 PM          749 Get-PSCredentialFromJson.md
-a----     1/10/2020   5:29 PM          755 Import-PSCredentialFromJson.md
-a----     1/10/2020   5:29 PM          735 PSJsonCredential.md
```

As you can see we get a Markdown file for each command plus one for the module which we'll get to later in the chapter. If you had existing comment based help for a command you should see it in the corresponding Markdown document. Edit as necessary. Otherwise the command generates a Markdown version of same content you would see in comment-based help. All you need to do is fill in the blanks by replacing sections like `{{Fill in the Description}}` with your content.

These files are text files so you can edit in Notepad, the PowerShell ISE or any text editor. You can also find Markdown-specific tools like MarkdownPad 2 or use VS Code. We'll open one of the files in the latter. Note, though, that because the Markdown is typically so simplistic (help files don't use boldfacing or anything fancy), there's no specific need to get a special Markdown editor if you don't already have one.

Visual Studio Code has an add-in that allows it to interpret and "render" Markdown documents, making it a pretty slick Markdown editor. Press `Ctrl+Shift+P` for the command palette and start typing "Markdown". Select "Open Preview to the Side"



Markdown in VS Code

As you can see, all of the help sections are created for you. Fill in the blanks and you are ready to go. Again, you don't have to know much about Markdown syntax. But we'll point out a couple of tips.

In **Examples** sections, any code between 3 back ticks will be formatted as code (giving you more control over your example formatting than in comment-based help). You can see the result in the preview. After the back tick-ed section of code, add any descriptive text for your example. If you want to show output of your command, insert it inside the back-ticked code block.

In the **Related Links** section, add other commands enclosed in square brackets followed by a set of parentheses:

```
[Get-Credential]()

[ConvertFrom-SecureString]()

[Import-PSCredentialFromJson](Import-PSCredentialFromJson.md)

[Get-PSCredentialFromJson](Get-PSCredentialFromJson.md)

[https://msdn.microsoft.com/en-us/library/system.management.automation.pscredential(\
v=vs.85).aspx]()
```

Inside the parentheses you can include a link. In the example, the link for the other commands is to the Markdown file in the same folder. The Microsoft link obviously is to MSDN, which is where they

keep the online version of their docs (which are also generated from the original Markdown source). If the text in the brackets is a URL, the Markdown document will automatically turn that into a live link. The text inside the square brackets is what will ultimately be displayed in the external help.

Repeat this process for your remaining Markdown documents. By the way, if you had created another set of documents for another language, you would of course translate them as necessary.

The Module Page

If you followed our example above, you should have also created a Markdown document for the module that looks like this:

```
---
Module Name: PSJsonCredential
Module Guid: a582b122-80fd-4fcb-8c01-5520737530c9
Download Help Link: {{Please enter FwLink manually}}
Help Version: {{Please enter version of help manually (X.X.X.X) format}}
Locale: en-US
---

# PSJsonCredential Module
## Description
{{Manually Enter Description Here}}

## PSJsonCredential Cmdlets
### [Export-PSCredentialToJson](Export-PSCredentialToJson.md)
{{Manually Enter Export-PSCredentialToJson Description Here}}

### [Get-PSCredentialFromJson](Get-PSCredentialFromJson.md)
{{Manually Enter Get-PSCredentialFromJson Description Here}}

### [Import-PSCredentialFromJson](Import-PSCredentialFromJson.md)
{{Manually Enter Import-PSCredentialFromJson Description Here}}
```

As you can see there are places to fill in the blanks. If you intend to create downloadable help, which we'll also get to, you can specify the online location for the "Download Help Link". As of the time we're writing this chapter, this link must be HTTP. Also, you should manually enter the help version.

Or you could use the parameters `-HelpVersion` and `-FwLink` with `New-MarkdownHelp`. You can create the module page at any time but if you've already started editing your command markdown files, run the command to a temporary folder then copy the module Markdown file to your Docs folder.

```
New-MarkdownHelp -Module PSJsonCredential -OutputFolder d:\temp `
-WithModulePage -HelpVersion 1.0.0.0 `
-fwlink http://mywebserver/help -force
Copy-Item D:\temp\PSJsonCredential.md -destination c:\psjsoncredential\docs
```

Here's the updated module page:

```
---
Module Name: PSJsonCredential
Module Guid: a582b122-80fd-4fcb-8c01-5520737530c9
Download Help Link: http://mywebserver/help
Help Version: 1.0.0.0
Locale: en-US
---

# PSJsonCredential Module
## Description
{{Manually Enter Description Here}}

## PSJsonCredential Cmdlets
### [Export-PSCredentialToJson](Export-PSCredentialToJson.md)
{{Manually Enter Export-PSCredentialToJson Description Here}}

### [Get-PSCredentialFromJson](Get-PSCredentialFromJson.md)
{{Manually Enter Get-PSCredentialFromJson Description Here}}

### [Import-PSCredentialFromJson](Import-PSCredentialFromJson.md)
{{Manually Enter Import-PSCredentialFromJson Description Here}}
```

Create External Help from Markdown

Before we create the external help we'll need a language specific folder. We know for our module that this is going to be en-US so you can simply run:

```
mkdir en-us
```

Or if you prefer a more agnostic approach try this:

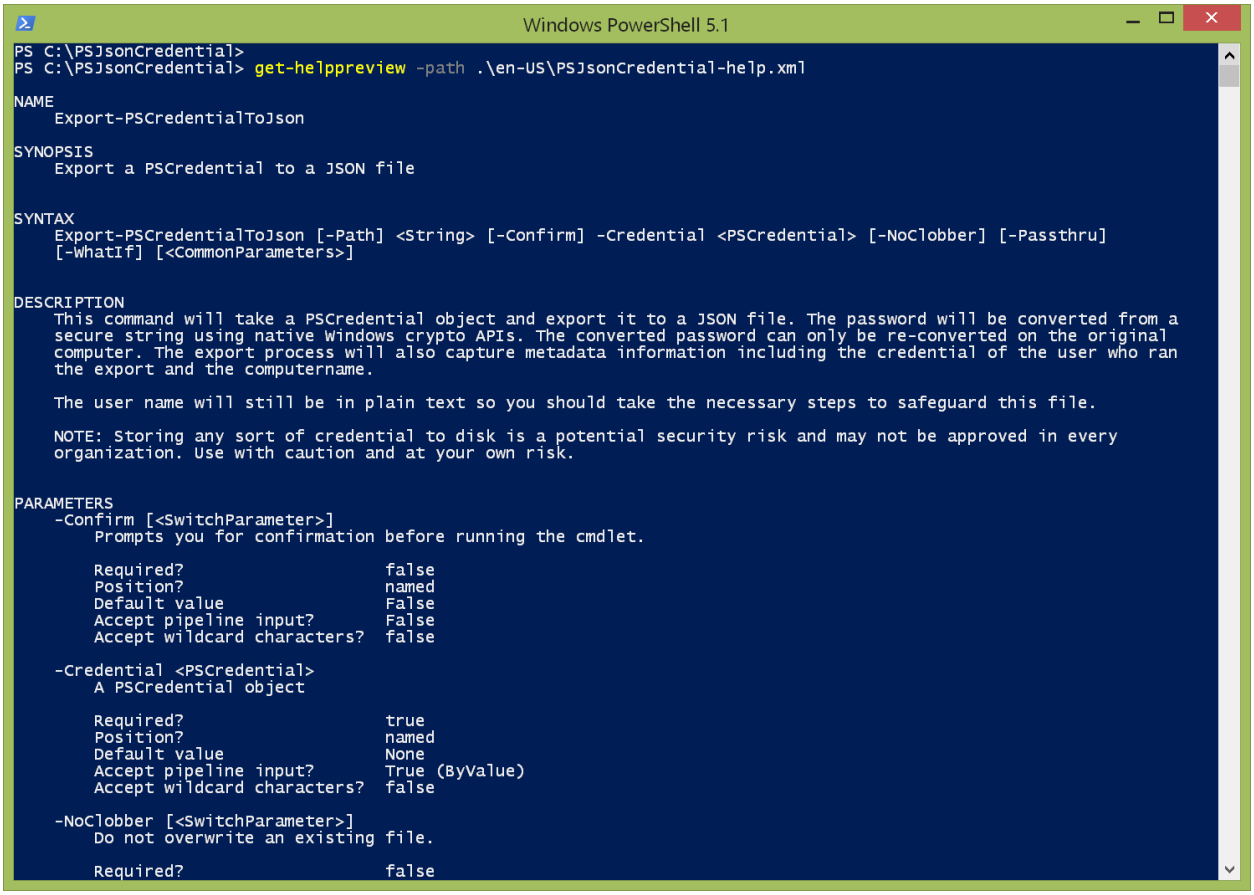
```
PS C:\PSJsonCredential> mkdir (Get-Culture).name


    Directory: C:\PSJsonCredential


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        1/10/2020   5:13 PM                en-US
```

If you need additional languages, create them as necessary.Then create new external help from your Markdown files.

```
PS C:\PSJsonCredential> New-ExternalHelp -Path .\Docs\ `
-OutputPath .\en-US\ -Force


    Directory: C:\PSJsonCredential\en-US


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        1/11/2020   8:53 AM          17370 PSJsonCredential-help.xml
```

Use the `-Force` parameter to overwrite previous versions of the xml file. You then test the help to see what it will look like in PowerShell.

```
Get-HelpPreview -Path .\en-US\PSJsonCredential-help.xml
```



Help preview

Congratulations! At this point your module now has professional grade help documentation.

Note that should you need to revise your module and help, you can use `Update-MarkdownHelp` to add command changes to your Markdown help without losing what you've created previously. When finished updating the Markdown, create new external help as before with `-Force`.

Supporting Online Help

Most PowerShell commands out of the box have a feature where you can go online to get the most current version of help.

```
help get-ciminstance -online
```

Did you know you can do the same thing? It is easier than you think, assuming you have already created the online destination. In the module or script file where your code is defined you should have a `[Cmdletbinding()]` attribute. Within this you will add a `HelpUri` settings specifying the online location.

```
Function Get-PSCredentialFromJson {

[cmdletbinding(HelpUri="http://bit.ly/Get-PSCredentialJson")]
```

You don't have to use a shortening service but it is handy should you need to redirect users to a new site or page. This link can point to any web page that provides online help. You might put the Markdown document for the command online and point the HelpUri to that.

While it isn't required, you might also want to include the URI under the Related Links section of your Markdown document.

```
## RELATED LINKS
[http://bit.ly/Get-PScredentialJson]()
```

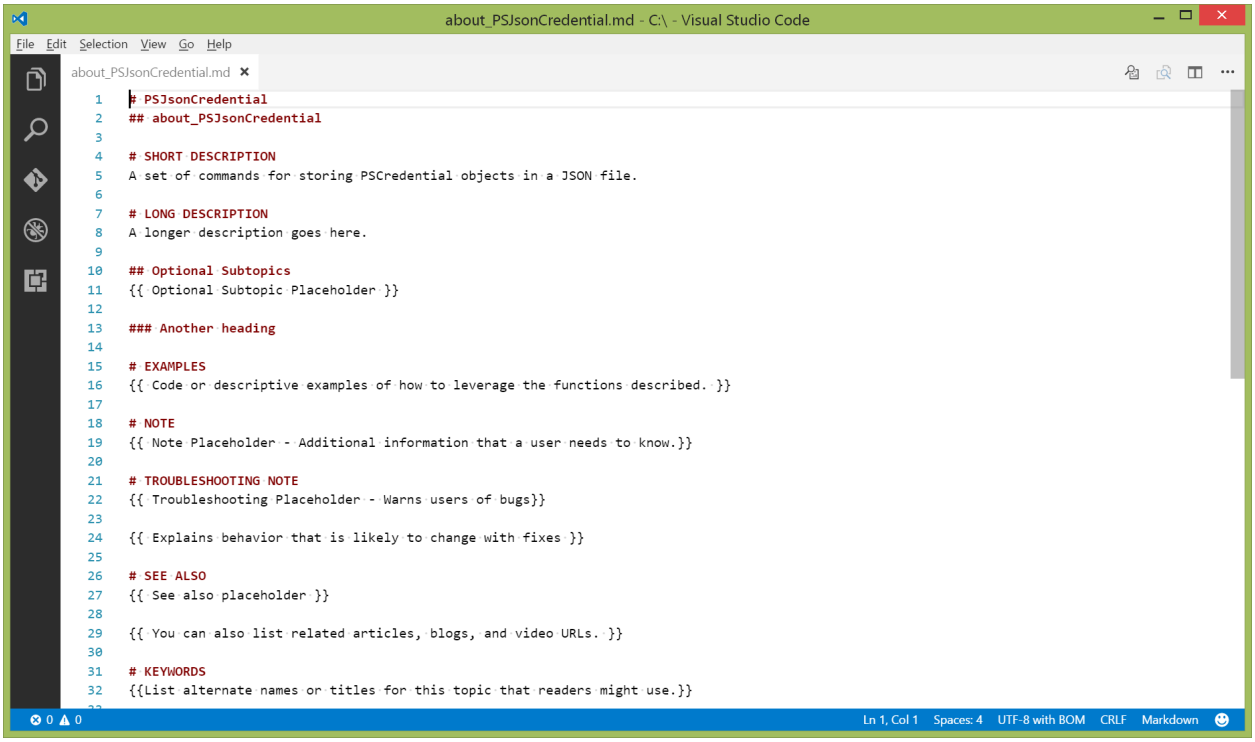
"About" Topics

Depending on your toolset, you may also want to include an **About** help topic. This file can offer more insights and guidance on how to use the commands in your tool, cover general concepts, and so on. Adding an about topic, especially for a complex toolset, is the sign of an experienced toolmaker.

You can use the Platyps module to create this as well.

```
PS C:\PSJsonCredential> New-MarkdownAboutHelp -OutputFolder .\Docs\ -AboutName PSJso\
nCredential
```

The `-AboutName` value typically will be the name of your module. This will create a file called `about_-<aboutname>.md` which you can edit as you did before.



About Markdown in VSCode

You can delete the sections wrapped in code block that are notes so that the beginning of your document looks like this.

```
# PSJsonCredential
## about_PSJsonCredential

# SHORT DESCRIPTION
{{ Short Description Placeholder }}

# LONG DESCRIPTION
{{ Long Description Placeholder }}
```

Insert your documentation as indicated. You can use the # character to indicate heading style. Each additional # character indicates another level (e.g., # is for level 1, ## is for level 2, and so on; there's not a lot of point in using more than 2 levels). You can see this easily in VSCode or anything else you use to preview your Markdown document.

When you are finished, run `New-ExternalHelp`, specifying the folder with your about Markdown document. This will create the proper text file in your culture-specific folder.

Because this is a simple text file, you could create it by hand without any intermediate Markdown. Just be sure to follow the same heading outline. The name of your file must be `about_-<modulename>.help.txt`, and place it in your language specific folder.

Making Your Help Updatable

The other professional feature in PowerShell help is the ability to download or update help from an online source. You too can use this feature. The first thing you will need is a CAB file with your updated help documentation. This file should include compressed versions of your help XML file and any About topics. Fortunately the Platyps module includes a command, `New-ExternalHelpCab` that will handle this task for you.

If you are interested, all of the sausage-making details can be found on MSDN at https://msdn.microsoft.com/en-us/library/hh852754(v=vs.85).aspx[25].

In order to use `New-ExternalHelpCab` you need to have a finished module Markdown page. The related files will be named based on information in the module page.

```
New-ExternalHelpCab -CabFilesFolder C:\PSJsonCredential\en-US\ `
-LandingPagePath C:\PSJsonCredential\Docs\PSJsonCredential.md  `
-OutputFolder c:\psJsonCredential\help
```

The first parameter is the path to your external XML help files. The second parameter points to the module Markdown page and the last parameter is the location where you'll store your files. You might get a bunch of XML-related output which you can ignore. The important part is that you should get files like this:

```
PS C:\PSJsonCredential> dir .\help\ | select-object name

Name
----
PSJsonCredential_a582b122-80fd-4fcb-8c01-5520737530c9_en-US_helpcontent.cab
PSJsonCredential_a582b122-80fd-4fcb-8c01-5520737530c9_en-US_helpcontent.zip
PSJsonCredential_a582b122-80fd-4fcb-8c01-5520737530c9_HelpInfo.xml
```

The cab file is named using the pattern:

```
<modulename>_<module guid>_<culture>_helpcontent.cab
```

The HelpInfo XML file follows a similar pattern with the module name and guid. This XML file contains the information that tells PowerShell where to download the cab file.

[25] https://msdn.microsoft.com/en-us/library/hh852754(v=vs.85).aspx

```
<?xml version="1.0" encoding="utf-8"?>
<HelpInfo xmlns="http://schemas.microsoft.com/powershell/help/2010/05">
  <HelpContentURI>http://mywebserver/help</HelpContentURI>
  <SupportedUICultures>
    <UICulture>
      <UICultureName>en-US</UICultureName>
      <UICultureVersion>1.0.0.1</UICultureVersion>
    </UICulture>
  </SupportedUICultures>
</HelpInfo>
```

Typically all of these files will go into the same location on a web server. The last piece of the puzzle is to update the module manifest and set the HelpInfoUri to this location.

```
# HelpInfo URI of this module
HelpInfoURI = 'http://mywebserver/help'
```

This location should **not** include the name of the XML file, only to its container. When you run one of the updatable help cmdlets, PowerShell checks the module for the HelpInfoUri address, connects to it and downloads the module specific HelpInfo.xml file, opens up the XML file to get the HelpContentUri which points to the location of the cab file which it then downloads.

You can test this by running `Save-Help`. We recommend using `-Verbose` so you can verify the locations.

We mentioned this earlier, but at the time of this writing you can only use HTTP locations. HTTPS does not work.

Your Turn

Let's see what you can do using the Platyps cmdlets to create professional-quality help documentation. There are commercial tools you can also use for creating comment-based help but the Platyps module is freely available which we like.

Start Here

The first thing you'll need to do is install the Platyps module. Take a few minutes to read cmdlet help and examples. Then make sure you download the code for this book. Open up the code for this chapter and you should see a module called TMSample. Your job is to create external help documentation for this module.

Your Task

This chapter's downloadable code sample contains a version of the Get-TMRemoteListeningConfiguration‘ command we've been working with plus a related command to get TrustedHosts information. You don't need to worry about running the commands. Change location to the root of the module and follow the steps we've described in this chapter. To import the module, change to the appropriate folder and run this command:

```
import-module .\TMSample.psd1
```

You don't need to create an About topic, unless you are feeling like you need an extra challenge. Once you create your help test it in PowerShell by re-importing the module and running help on the module commands.

```
import-module .\TMSample.psd1 -force
help Get-TMRemoteListeningConfiguration -full
```

Our Take

After you've finished you can compare your help with ours in the Solution folder.

Let's Review

Let's wrap up with a few review questions.

1. What special language is external help written in?
2. What are some of the benefits of using external help?
3. What type of help document can you create to provide additional information about your tool?
4. If you want to support updatable help, what setting do you need to configure in your module manifest?

Review Answers

1. External help is written in a MAML flavor of XML.
2. External help makes it easier to update help separately from your code. It also makes it easier if you need localized help for different languages.
3. You can create an About topic which can have as much detail, background or additional examples that you need.
4. HelpInfoUri.

Unit Testing Your Code

One of the most important things you can do with your code is test it. That should go without saying. And you've probably already done some testing of your scripts and commands, which is great - except you've probably done it manually. That presents two problems.

1. Manual testing is inconsistent. Sometimes, you'll remember to test certain things, and other times you'll inevitably forget something.
2. It's easy, with manual testing, to have a kind of confirmation bias. You'll deliberately forgo testing something because testing is tedious and you "just know" that thing works anyway.

Pester - which ships with Windows 10 and later, and is available for download from PowerShell Gallery - is designed to help with those problems. It's a unit testing framework that can help automate your testing. You essentially tell it what to test, and then it can test the same things, every time. If you realize that you've forgotten to test something, you can just add that test, and Pester will handle it from then on.

This chapter is intended only to be a short introduction to Pester, and to cover its most basic syntax. The full version of this book includes a Part dedicated to Pester and PowerShell unit testing for those who'd like to dig deeper.

Starting Point

To get going, let's create a short script to test. Note that this script, as presented right here (and in the code samples as Step1.ps1), *is not necessarily going to work perfectly*. That's kind of the point of it - we haven't tested it yet. This is the kind of "I just wrote it, and I hope it works" code that you might end up with after you've been in the ISE for a bit, but haven't hit "run" yet. Also note that this script is *deliberately* simplistic. We want to focus on testing it, without making this chapter into *War and Peace*. Also notice that this is a script which contains a function.

```
function Get-FileContents {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True)]
        [string[]]$Path
    )
    PROCESS {
        foreach ($folder in $path) {
            Write-Verbose "Path is $folder"
            $segments = $folder -split "\\"
            $last = $segments[-1]
            Write-Verbose "Last path is $last"
            $filename = Join-Path $folder $last
            $filename += ".txt"
            Get-Content $filename
        } #foreach folder
    } #process
}
```

If you're looking at the sample code, you'll also notice an empty "Get-FileContents.Test.ps1" file. This is where we'll start building our tests. The intent of `Get-FileContents` is to accept one or more folder paths. For each, it will take the final folder name in the path, and assume that is also the filename of a .txt file. So, if the path is c:\test\testing, then it will attempt to read the contents of c:\test\testing\testing.txt.

Sketching Out the Test

Our Tests file looks like this right now:

```
$here = Split-Path -Parent $MyInvocation.MyCommand.Path
$sut = (Split-Path -Leaf $MyInvocation.MyCommand.Path) -replace '\.Tests\.', '.'
. "$here\$sut"

Describe "Get-FileContents" {
    It "does something useful" {
        $true | Should Be $false
    }
}
```

This is boilerplate that was created by running Pester's `New-Fixture` command. It actually created our `Get-FileContents.ps1` file, and populated it with the function declaration that we added our

code to. You don't *have* to use `New-Fixture`; you can quite easily use the above boilerplate to create a Tests script for something that you've already written. This boilerplate contains a `Describe` block, which is the main structure that a Pester tests live inside. It also contains a single `It` block as a placeholder. Essentially, each `It` block represents a single test that we're going to run against our code.

Making Something to Test

Because our function is assembling file paths and attempting to read files, we need to give it something to test. Pester provides a `TESTDRIVE:` for that purpose. It's a special FileSystem PSDrive that Pester automatically sets up when you run your test. Under the hood, it lives in your system's TEMP folder, and Pester takes care not only of setting it up, but also of deleting it when your tests are complete. That makes `TESTDRIVE:` a kind of sandbox, so that you're not polluting your real filesystem with testing artifacts. So, inside our `Describe` block, we're going to set up a few folders and files to test against. We're moving on to the Step2 folder in our sample code.

```
$here = Split-Path -Parent $MyInvocation.MyCommand.Path
$sut = (Split-Path -Leaf $MyInvocation.MyCommand.Path) -replace '\.Tests\.', '.'
. "$here\$sut"

Describe "Get-FileContents" {

    MkDir TESTDRIVE:\Part1
    MkDir TESTDRIVE\Part1\Part2
    MkDir TESTDRIVE:\Part1\Part3
    "sample" | Out-File TESTDRIVE:\Part1\Part2\Part2.txt
    "sample" | Out-File TESTDRIVE:\Part1\Part3\Part3.txt
    "sample" | Out-File TESTDRIVE:\Part1\Part1.txt

    It "does something useful" {
        $true | Should Be $false
    }
}
```

Now we need to write our first test:

```
$here = Split-Path -Parent $MyInvocation.MyCommand.Path
$sut = (Split-Path -Leaf $MyInvocation.MyCommand.Path) -replace '\.Tests\.', '.'
. "$here\$sut"

Describe "Get-FileContents" {

    MkDir TESTDRIVE:\Part1
    MkDir TESTDRIVE\Part1\Part2
    MkDir TESTDRIVE:\Part1\Part3
    "sample" | Out-File TESTDRIVE:\Part1\Part2\Part2.txt
    "sample" | Out-File TESTDRIVE:\Part1\Part3\Part3.txt
    "sample" | Out-File TESTDRIVE:\Part1\Part1.txt

    It "reads part2.txt" {
        Get-FileContents -Path TESTDRIVE:\Part1\Part2 |
        Should Be "sample"
    }
}
```

We've used the `It` block to provide a brief description of what's happening. Then, we run our command with a given parameter, and we test to see that the output is what we expected. `Should` is another Pester command, and we've followed it with the `Be` operator, indicating that we expect `Get-FileContents` to return the string "sample."

Now we'll go to a console window and use `Invoke-Pester` to run our test:

```
PS x:\unit-testing-your-code\step2> Invoke-Pester
Describing Get-FileContents
 [+] reads part2.txt 87ms
Tests completed in 87ms
Passed: 1 Failed: 0 Skipped: 0 Pending: 0 Inconclusive: 0
```

We can see the output of our `Describe` block, and our `It` block as a `+` indicator, showing us that our test passed.

Expanding the Test

Let's add a couple more tests, now in Step3.

```
$here = Split-Path -Parent $MyInvocation.MyCommand.Path
$sut = (Split-Path -Leaf $MyInvocation.MyCommand.Path) -replace '\.Tests\.', '.'
. "$here\$sut"

Describe "Get-FileContents" {

    MkDir TESTDRIVE:\Part1
    MkDir TESTDRIVE:\Part1\Part2
    MkDir TESTDRIVE:\Part1\Part3
    "sample" | Out-File TESTDRIVE:\Part1\Part2\Part2.txt
    "sample" | Out-File TESTDRIVE:\Part1\Part3\Part3.txt
    "sample" | Out-File TESTDRIVE:\Part1\Part1.txt

    It "reads part2.txt" {
        Get-FileContents -Path TESTDRIVE:\Part1\Part2 |
        Should Be "sample"
    }

    It "reads part3.txt with fwd slashes" {
        Get-FileContents -PATH TESTDRIVE:/Part1/Part3 |
        Should Be "sample"
    }

    It "reads 3 files from the pipeline" {
        $results = "TESTDRIVE:\part1",
        "TESTDRIVE:\part1\part2",
        "TESTDRIVE:\part1\part3" | Get-FileContents
        $results.Count | Should Be 3
    }
}
```

The first test is making sure that forward slashes work as well as backslashes, since in PowerShell a path may legally contain either. The second test is feeding three paths, as strings, to our command, and capturing the results in $results. We know that our test files contain one line apiece, so reading three files should result in three objects in $results. We test that by piping `$results.Count` to `Should`, and checking to see that the count is indeed 3.

```
PS x:\unit-testing-your-code\step3> Invoke-Pester
Describing Get-FileContents
 [+] reads part2.txt 82ms
Get-Content : Cannot find path
'TestDrive:\Part1\Part3\TESTDRIVE:\Part1\Part3.txt' because it does
not exist.
At \\vmware-host\shared folders\Documents\GitHub\ToolmakingBook\code\
PowerShell-Toolmaking\Chapters\unit-testing-your-code\step3\Get-FileC
ontents.ps1:17 char:13
+             Get-Content $filename
+             ~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (TestDrive:\Part...Par
   t1\Part3.txt:String) [Get-Content], ItemNotFoundException
    + FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Comm
   ands.GetContentCommand

 [-] reads part3.txt with fwd slashes 42ms
   Expected: {sample}
   But was:  {}
   22:         Should Be "sample"
   at <ScriptBlock>, \\vmware-host\shared folders\Documents\GitHub\Too
lmakingBook\code\PowerShell-Toolmaking\Chapters\unit-testing-your-code
\step3\Get-FileContents.Tests.ps1: line 21
 [+] reads 3 files from the pipeline 61ms
Tests completed in 186ms
Passed: 2 Failed: 1 Skipped: 0 Pending: 0 Inconclusive: 0
```

Whoops. Our original first test passed, and our new third test passed, but the second test - with the forward slashes, as the Pester output clearly shows, failed. The exception thrown by our function indicates that the filename `TestDrive:\Part1\Part3\TESTDRIVE:\Part1\Part3.txt` couldn't be found, which makes sense, because that filename is crazy.

Returning to our code, here's the likely problem:

```
            $segments = $folder -split "\\"
```

We're breaking the path up based on backslashes, which obviously doesn't take forward slashes into account. We'll fix that by converting forward slashes as a preliminary step (this is in step4 in the sample code):

```
            $folder = $folder -replace "/","\"
            $segments = $folder -split "\\"
```

And we'll try our test again:

```
PS x:\unit-testing-your-code\step4> Invoke-Pester
Describing Get-FileContents
 [+] reads part2.txt 220ms
 [+] reads part3.txt with fwd slashes 23ms
 [+] reads 3 files from the pipeline 38ms
Tests completed in 283ms
Passed: 3 Failed: 0 Skipped: 0 Pending: 0 Inconclusive: 0
```

Fantastic! Now we're assured of that particular bug never creeping up unnoticed again.

But Wait, There's More

Pester has a lot more it can do. A key concept is *mocking*, which means sort of overriding an existing command so that it outputs exactly what you want. For example, if your code relies on the `Get-ChildItem` cmdlet, you might not feel compelled to actually test `Get-ChildItem` to make sure it's working. After all, you didn't write that command, so if it's broken, there's not much you can do anyway. Rather than setting up a directory structure to test against (as we did in our run-through above), you could instead *mock* `Get-ChildItem`, temporarily replacing it, in your tests, with your own version that always returns a specific result. It's a way of simplifying the testing process, removing external dependencies, and focusing just on *your* code. We're not going to go into mocking here, as it gets to be a fairly complex topic, and would instead refer you to *The Pester Book*, which we mentioned at the top of this chapter. You can also visit the Pester project[26] for the core Pester documentation, which covers mocks, all the other things `Should` can do, and much more.

Your Turn

Let's give you a shot at making a simple Pester test of your own. First, make sure you have Pester installed by running `Import-Module Pester` and making sure no errors occur. If you don't have it, run `Install-Module Pester` to install the module from PowerShell Gallery.

The Pester module is periodically updated so even if you are running Windows 10 you might want to run `Find-Module Pester -repository PSGallery` and compare the version to your currently installed version. Upgrade the module as necessary.

[26] https://pester.dev/docs/quick-start

Start Here

The following function *should work*, and you'll find it in the lab-start folder, in the downloadable code samples for this chapter. The purpose of this very simple command is to verify that a service is started and, if not, start it. It accepts one or more service names as strings, and returns the resulting service. If you give it a non-existent service name, it should simply skip it without error.

```
function Set-ServiceStatus {
    [CmdletBinding()]
    Param(
        [string[]]$ServiceName
    )
    foreach ($name in $ServiceName) {

        $svc = Get-Service $name -EA SilentlyContinue
        if ($svc) {
            if ($svc.Status -ne 'Running') {
                $svc | Start-Service
            }
            $svc | Get-Service
        }

    } #foreach
}
```

Your Task

Write a Pester test - we've provided you with the boilerplate in lab-start - that tests the following:

- A non-existent service name doesn't throw an error
- An existing, started service remains started
- An existing, stopped service is now started

Our Take

Our results are in lab-results, and look like this:

<<Set-ServiceStatus.Tests.ps1[27]

And our Pester run:

[27] code/PowerShell-Toolmaking/Chapters/unit-testing-your-code/lab-results/Set-ServiceStatus.Tests.ps1

```
PS X:\unit-testing-your-code\lab-results> Invoke-Pester

Status   Name               DisplayName
------   ----               -----------
Running  bits               Background Intelligent Transfer Ser...
Describing Set-ServiceStatus
 [+] starts BITS 1.39s
 [+] starts BITS, skips FAKE 542ms
 [+] starts 2 services 1.09s
Tests completed in 3.02s
Passed: 3 Failed: 0 Skipped: 0 Pending: 0 Inconclusive: 0
```

Notice that our third `It` block contains two `Should` tests. This is fine, because both tests are needed to ensure a correct result from the entire block. If either `Should` fails, the entire `It` is a fail. Of course, there are a lot of other ways you could have gone about this, so don't be alarmed if your test code is different.

Let's Review

Run through these review questions to make sure you picked up the key points from this chapter:

1. What does an `It` block represent?
2. Why might you have code other than `It` blocks within a `Describe` block?
3. What does `Should` do?

Review Answers

Here are our answers:

1. A single atomic pass/fail test of your code's output or results.
2. Code that sets up conditions which your `It` blocks will test against.
3. Compares a given value to an expected value, generating a Pass/Fail result.

Extending Output Types

One of the coolest things about PowerShell is its Extensible Type System, or ETS. To understand that, we'll need to cover a bit of boring (but important) terminology, and then we can show you why the ETS can be so awesome.

Understanding Types

In programming, a *type* is a description of what some programmatic structure looks like. For example, in .NET Framework, the `System.String` type includes methods for manipulating strings, counting the number of characters in the string, and so on. These are also referred to as the *interface* of the type - the means by which you, as a programmer, interact with it. You'll also see the word *class*, which refers to the means by which a type is implemented - its internal state, and the actual code that makes it work. But what's important here is the *type*.

Another important concept in programming is the so-called *contract* that a type represents. Just like a legal contract, a type's interface - its properties, methods, and other *members*, are meant to be carved in stone. The contract is there to help ensure forward compatibility. For example, if you write code based on the `ToShortDateString()` method of the `System.DateTime` type, you want to have some assurances that Microsoft won't eliminate that method, thus breaking your code, in some future release. That's why you'll sometimes see type names with a sort of version number, like `System.DateTime2` (although such a thing doesn't really exist yet). That defines a new type, and Microsoft could define it in any way they wanted, without breaking the contract for the original `System.DateTime`. However, it's generally OK to *add* things to a type's interface. You haven't written any code which depends on `System.Int32` *not having* a method called `ToFormattedString()`, so Microsoft could add such a method without breaking your old code. Adding to an interface isn't a *great* practice, because you start to have to worry about things like, "which version of .NET can my code run on, since some versions have such-and-such a member and others don't," and so it's pretty rare for framework developers, like Microsoft, to do that.

But PowerShell represents a slightly different situation.

The Extensible Type System

PowerShell's ETS doesn't *permanently* add anything to a type's interface. Instead, it *temporarily* extends the interface, and only does so within PowerShell. In a way, you can think of it as a thin "wrapper" around the original interface, with a few things gently stuck on for just that moment. Most of the time, the things PowerShell adds to a type are there solely for PowerShell's use, or to improve consistency in a systems administration context.

For example, Windows services are represented by the .NET Framework `System.ServiceProcess.ServiceControlle` type. The service's name is found in the `ServiceName` property of that type. That's all well and good, except that *most* .NET Framework types have a "Name" property. Because "Name" is so widely used, PowerShell has certain features which default to using the "Name" property. Services being such a commonly accessed administrative thing, it would be super-inconvenient for those features to simply not work. And so the ETS adds an *AliasProperty* called `Name` to `System.ServiceProcess.ServiceController`. The old `ServiceName` property is still there, so the type's contract is still valid, but `Name` also exists and contains the same data, so PowerShell's various features will work.

The ETS supports several different members that can be added to a type:

- ScriptMethod - actual PowerShell script that executes when the method is called
- ScriptProperty - actual PowerShell script that returns a property value when the property is accessed
- AliasProperty - points to an existing property using an alternate name
- PropertySet - a defined list of properties that can be referenced with a single name
- NoteProperty - a property containing a static value

One PropertySet that you'll often see is **DefaultDisplayPropertySet**. This is a list of property names that PowerShell should display, by default, when displaying the object. If the list contains 5 or fewer properties, PowerShell will always attempt to use a table-style display; for more properties, it will use a list-style display.

Extending an Object

For this example, we'll use a slight variation of the `Get-TMComputerStatus` function.

Start.ps1

```
Function Get-TMComputerStatus {

[cmdletbinding()]
Param(
    [Parameter(ValueFromPipeline, Mandatory)]
    [ValidateNotNullorEmpty()]
    [Alias("CN", "Machine", "Name")]
    [string[]]$Computername,
    [string]$ErrorLog,
    [switch]$ErrorAppend
)

BEGIN {
```

```
    Write-Verbose "Starting $($myinvocation.mycommand)"
}

PROCESS {
  foreach ($computer in $Computername) {
     Write-Verbose "Querying $($computer.toUpper())"

     Try {
         $params = @{
          Classname    = "Win32_OperatingSystem"
          Computername = $computer
          ErrorAction  = "Stop"
        }
        $OS = Get-CimInstance @params

        $params.ClassName = "Win32_Processor"
        $cpu = Get-CimInstance @params

        $params.className = "Win32_logicalDisk"
        $vol = Get-CimInstance @params -filter "DeviceID='c:'"

        $OK = $True
     }
     Catch {
         $OK = $False
         $msg = "Failed to get system information from $computer. $($_.Exception.Mes\
sage)"
         Write-Warning $msg
         if ($ErrorLog) {
             Write-Verbose "Logging errors to $ErrorLog. Append = $ErrorAppend"
            "[$(Get-Date)] $msg" | Out-File -FilePath $ErrorLog -Append:$ErrorAppend
         }
     }
     if ($OK) {
        #only continue if successful
        $obj = [pscustomobject]@{
          Computername = $os.CSName
          TotalMem     = $os.TotalVisibleMemorySize
          FreeMem      = $os.FreePhysicalMemory
          Processes    = $os.NumberOfProcesses
          PctFreeMem   = ($os.FreePhysicalMemory/$os.TotalVisibleMemorySize)*100
          Uptime       = (Get-Date) - $os.lastBootUpTime
          CPULoad      = $cpu.LoadPercentage
```

```
          PctFreeC     = ($vol.FreeSpace/$vol.size)*100
        }
        $obj
     } #if OK
    } #foreach $computer
}
END {
    Write-Verbose "Starting $($myinvocation.mycommand)"
}
} #Get-TMComputerStatus

Get-TMComputerStatus $env:computername
```

Specifically, we're going to be messing with this code:

```
if ($OK) {
  $obj = [pscustomobject]@{
  Computername = $os.CSName
  TotalMem     = $os.TotalVisibleMemorySize
  FreeMem      = $os.FreePhysicalMemory
  Processes    = $os.NumberOfProcesses
  PctFreeMem   = ($os.FreePhysicalMemory/$os.TotalVisibleMemorySize)*100
  Uptime       = (Get-Date) - $os.lastBootUpTime
  CPULoad      = $cpu.LoadPercentage
  PctFreeC     = ($vol.FreeSpace/$vol.size)*100
 }
$obj
} #if OK
```

We mentioned, way back, that we're in the habit of storing our newly created objects in a variable (`$obj` in this case), in case we ever want to modify the new object prior to outputting it. Here's where you'll see that practice in use. Now, this gets a little complicated, because the DefaultDisplayPropertySet is actually a child of a `PSStandardMembers` member set. Here we go:

```
# create a default display property set
[string[]]$props = 'ComputerName','Uptime','Processes','PctFreeMem','PctFreeC'
$ddps = New-Object -TypeName System.Management.Automation.PSPropertySet `
        DefaultDisplayPropertySet, $props
$pssm = [System.Management.Automation.PSMemberInfo[]]$ddps
$obj | Add-Member -MemberType MemberSet `
        -Name PSStandardMembers `
        -Value $pssm

$obj
```

You'll find the complete thing in `End.ps1` in the code download. For comparison, here's the output before:

```
Computername : BOVINE320
TotalMem     : 33442624
FreeMem      : 11314680
Processes    : 311
PctFreeMem   : 33.833110703275
Uptime       : 1.04:09:16.9994278
CPULoad      : 29
PctFreeC     : 30.2393308009492
```

And the output after:

```
ComputerName Uptime                   PctFreeMem         PctFreeC
------------ ------                   ----------         --------
BOVINE320    1.04:10:36.8124820 35.4735920243579 30.2408134074583
```

Our `DefaultDisplayPropertySet` is what made this happen. The other properties remain and can be seen by piping the command to `Select-Object`:

```
Get-TMComputerStatus $env:computername | select-object  *
```

ScriptMethods, ScriptProperties, AliasProperties, and NoteProperties are all far easier to make - simply pipe your object to `Add-Member`, specify the `-MemberType`, give it a `-Name`, and a `-Value`. For ScriptMethod and ScriptProperty, the value is a {script block}, meaning PowerShell code inside curly brackets.

In the download folder you will find another copy of our function called `using-add-member.ps1`. In this folder we've added a few more object members.

```
#adding an alias
$obj | Add-Member -MemberType AliasProperty `
                    -Name Memory `
                    -Value TotalMem

#adding a script method
$obj | Add-Member -MemberType ScriptMethod `
                    -Name Ping `
                    -Value { Test-NetConnection $this.computername }

#adding a script property
$obj | Add-Member -MemberType ScriptProperty `
                    -Name TopProcesses `
                    -Value {
                    Get-Process -ComputerName $this.computername |
                    Sort-Object -Property WorkingSet -Descending |
                    Select-Object -first 5
                    }
$obj
```

Now when we run the function for the local host we can see these additions with `Get-Member`.

```
 TypeName: System.Management.Automation.PSCustomObject

Name               MemberType     Definition
----               ----------     ----------
Memory             AliasProperty  Memory = TotalMem
PSStandardMembers  MemberSet      PSStandardMembers {DefaultDisplayPropertySet}
Equals             Method         bool Equals(System.Object obj)
GetHashCode        Method         int GetHashCode()
GetType            Method         type GetType()
ToString           Method         string ToString()
Computername       NoteProperty   string Computername=BOVINE320
CPULoad            NoteProperty   uint16 CPULoad=7
FreeMem            NoteProperty   uint64 FreeMem=11732260
PctFreeC           NoteProperty   double PctFreeC=30.2406928049765
PctFreeMem         NoteProperty   double PctFreeMem=35.0817567425331
Processes          NoteProperty   uint32 Processes=305
TotalMem           NoteProperty   uint64 TotalMem=33442624
Uptime             NoteProperty   timespan Uptime=1.04:14:28.8250846
Mem                PropertySet    Mem {Computername, TotalMem, FreeMem, PctFreeMem}
Ping               ScriptMethod   System.Object Ping();
TopProcesses       ScriptProperty System.Object TopProcesses {get= ...
```

If we save the command output to a variable we can see these new object properties and methods. The alias is an alternate property name:

```
PS C:\> $o = Get-TMComputerStatus $env:computername
PS C:\> $o.memory
33442624
PS C:\> $o.TotalMem
33442624
```

The property set is a way of predefining a group of properties so that instead of running this:

```
PS C:\>  $o | Select-object Computername,TotalMem,FreeMem,PctFreeMem

Computername TotalMem  FreeMem       PctFreeMem
------------ --------  -------       ----------
BOVINE320    33442624 12348324 36.9239088416029
```

We can run this:

```
PS C:\>  $o | Select-object Mem

Computername TotalMem  FreeMem       PctFreeMem
------------ --------  -------       ----------
BOVINE320    33442624 12348324 36.9239088416029
```

The script property uses PowerShell to get a value. The code is invoked anytime you access the property. In our function, we created a property that reflects the top 5 processes.

```
PS C:\> $o.TopProcesses

$o.TopProcesses

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)     Id  SI ProcessName
-------  ------    -----      -----     ------     --  -- -----------
      0       0     4416    1700048     237.52   2480   0 Memory Compression
    283      46  1857712    1350032   3,738.92  29556   1 TabNine
   1570     217   978384     842932   2,322.17  22960   1 firefox
   1467      86   612828     440168     427.52   5952   1 powershell_ise
      0       0  1218936     426460     573.06  14504   0 vmmem
```

Finally the script method can be invoked to *do* something.

```
PS C:\> $o.ping()

ComputerName            : BOVINE320
RemoteAddress           : fe80::ddae:8ade:c3ff:e584%19
InterfaceAlias          : vEthernet (LabNet)
SourceAddress           : fe80::ddae:8ade:c3ff:e584%19
PingSucceeded           : True
PingReplyDetails (RTT) : 0 ms
```

Using Update-TypeData

There's nothing wrong with using `Add-Member` for simple extensions. It is certainly much easier than the traditional way of using complicated XML files, which we're going to spare you. But, we want to mention another cmdlet that you might also consider, especially if you are building a toolset that will be working with custom objects.

You can use `Update-TypeData` to achieve many of the same results that we showed with `Add-Member`. The primary difference is that you will need to explicitly specify a type name. Open up the downloaded files for this chapter and you'll find the beginnings of a module file (Info.psm1). This module has the same function, with modifications.

The most important change is that we have inserted a custom type name into the output object.

```
$obj.psobject.TypeNames.Insert(0,"TMComputerStatus")
```

The command tells PowerShell to insert 'TMComputerStatus' as the primary type name. If you create an object with this function and pipe it to `Get-Member` you'll see this as the typename instead of PSCustomObject.

We will use this name to update the type. We've removed most of the `Add-Member` commands from the function and used `Update-TypeData` later in the psm1 file.

```
$myType = "myMachineInfo"

Update-TypeData -TypeName $myType -DefaultDisplayPropertySet 'ComputerName',
'OSVersion','Cores','RAM' -force

Update-TypeData -TypeName $myType -MemberType AliasProperty -MemberName Free `
                                  -Value SysDriveFreeSpace -force

Update-TypeData -TypeName $myType -MemberType ScriptMethod -MemberName Ping `
                                  -Value {
```

```
                                    Test-NetConnection $this.computername
                                   } -force

Update-TypeData -TypeName $myType -MemberType ScriptProperty -MemberName `
                                TopProcesses -Value {
                                Get-Process -ComputerName $this.computername |
                                Sort-Object -Property WorkingSet -Descending |
                                Select-Object -first 5
                                } -force
```

We use the `-force` parameter to overwrite any previous updates to this typename. The only type extension we left in the function with `Add-Member` was the PropertySet. There appears to a bug with `Update-TypeData` and this property extension. When working in the PowerShell ISE it *looks* like it should work, but when PowerShell goes to execute the command it errors.

One advantage to this approach is that the type extension is now separate from the function. We can now extend or modify the object type without having to edit the function and run the risk of screwing something up. Technically we could also dynamically extend the type after the fact for any objects previously created.

Define a PSTypename

You will find plenty of code samples in the wild that use the `Insert()` method which is why we showed it. However when defining a custom object, you can also define the type name as part of the property hashtable.

```
 $obj = [pscustomobject]@{
    PSTypeName   = "TMComputerStatus"
    Computername = $os.CSName
    TotalMem     = $os.TotalVisibleMemorySize
    FreeMem      = $os.FreePhysicalMemory
    Processes    = $os.NumberOfProcesses
    PctFreeMem   = ($os.FreePhysicalMemory/$os.TotalVisibleMemorySize)*100
    Uptime       = (Get-Date) - $os.lastBootUpTime
    CPULoad      = $cpu.LoadPercentage
    PctFreeC     = ($vol.FreeSpace/$vol.size)*100
 }
 Write-Output $obj
```

With this approach you don't need to insert anything and all of your type extension commands using `Update-TypeData` can be handled elsewhere in your module.

You might also want to take a look at Jeff's PSTypeExtensionTools[28] module.

[28] https://github.com/jdhitsolutions/PSTypeExtensionTools

Next Steps

Because this was a pretty straightforward exercise, we're not going to include a formal hands-on exercise. We encourage you to try out the sample code. Any code you need to add to your work can be simple cut-and-paste job of what we did. In fact, encourage you to try this out in one of your own functions that produce a custom object.

Advanced Debugging

You should already be familiar with the main debugging mechanisms in PowerShell. Those will get you through a lot of different scenarios, but there will definitely be times when you need a little more power in your debugging toolset. With that in mind, we'll build on those basic debugging concepts and tools.

Getting Fancy with Breakpoints

The most basic use of breakpoints is their "interactive" mode. That is, when we set a *breakpoint* on a line number, and script execution reached that line number, the script stops and we drop into the debug console. Again sticking with basic usage, you might only set breakpoints on particular line numbers, which is a pretty common need. But we can do so much more!

Types of Breakpoints

PowerShell actually supports different breakpoint triggers:

- Breaking on a given line number, or on a line and a given column number. The latter lets you break at a specific point in a long, multi-command pipeline, for example. You engage these using the `-Line` and `-Column` parameters of `Set-PSBreakpoint`, and can also manage these visually in the ISE.
- Breaking on a given command. This triggers the breakpoint whenever the specified command is about to run. This is *really* useful for long scripts where you need to stop before, say, each use of `Write-Output`, but you don't want to manually create breakpoints on each line. Use the `-Command` parameter to create these.
- Breaking on a variable. This triggers the breakpoint when a given variable is read, changed (written), or either of those. Specify the variable name with `-Variable` (keeping in mind that you *only* specify the name, not the `$`), and use `-Mode` to specify Read, Write, or ReadWrite.

Variable breakpoints in particular are like magic, and they're similar to "watches" that some integrated development environments (like Visual Studio) let you create in "real" programming languages. Remember, script logic errors are nearly always caused by a variable (or property value) containing *something other than you expected*, and so breaking when a variable changes is a perfect time to validate your assumptions about what the variable contains.

Breakpoint Actions

When triggered, a breakpoint's default action is to dump you into the `DBG\>` debug console. But you can always specify a separate `-Action`, by passing a script block to the parameter. That script block can *run any legal PowerShell code*, and specifying an action disables the dump-to-debug-console behavior. You might write action code that logs some message to a file, or even dumps the entire `VARIABLE:` drive to a text file so that you can analyze it later. This is a great tool for *unattended debugging*, such as with scripts that work fine interactively, but that fail when run as a scheduled task. By having automated breakpoints, you can gather evidence about the actual execution environment, even though you can't "personally" be there while the code is running.

For example:

```
Set-PSBreakpoint -Script .\Mine.ps1
                 -Variable data
                 -Mode ReadWrite
                 -Action { Dir VARIABLE: | Out-File .\vars.txt -Append }
```

This would dump the entire VARIABLE: drive each time `$data` was read or changed. After the script ran, you could analyze that file while walking through your code, and you'd *know*, at each step, what each variable contained.

There are a couple of rules about action scripts:

- The script block will run each time the breakpoint is triggered.
- If you run the `break` keyword in the script, you'll stop execution of your code.
- If you run `continue` in the script, the action code will exit and your script will resume execution.

Getting Strict

The `Set-StrictMode` command isn't a debugging technique per se; it's actually designed to help prevent certain types of bugs. Consider this command (we'll truncate the output to save space, but it's the columns to focus on):

```
PS C:\> get-service | select name,satus

Name                                      satus
----                                      -----
AJRouter
ALG
AppIDSvc
Appinfo
AppMgmt
AppReadiness
AppVClient
```

You see the problem, right? We misspelled "status," and got a blank "satus" column. Typos like this cause problems all the time. Here's another:

```
PS C:\> $a = 3
PS C:\> $b = 4
PS C:\> $a + $v
3
```

Clearly, not what we intended, but we hit "v" because it's next to "b" on the keyboard. Oops. The point is that, in a script, PowerShell's casual treatment of non-existent variables and property names can cause difficult-to-diagnose problems.

```
PS C:\> Set-StrictMode -Version Latest
PS C:\> $a = 3
PS C:\> $b = 4
PS C:\> $a + $v
The variable '$v' cannot be retrieved because it has not been set.
At line:1 char:6
+ $a + $v
+      ~~
    + CategoryInfo          : InvalidOperation: (v:String) [], Runti
   meException
    + FullyQualifiedErrorId : VariableIsUndefined
```

You should take the time to read full help and examples for `Set-StrictMode`. The best way to avoid the most problems is to set the `-version` parameter to "Latest"

This is much more desirable behavior. Now, instead of blithely accepting `$v` and treating it as if it contains zero, the shell is telling us that the variable hasn't been created.

```
PS C:\> get-service | select name,satus

Name                                satus
----                                -----
AJRouter
ALG
AppIDSvc
Appinfo
AppMgmt
AppReadiness
```

Sadly, strict mode doesn't affect the `Select-Object` command the same way. But it *will* throw an error in a script that:

- Tries to access an uninitialized variable
- Tries to access a property that doesn't exist (commands like `Select-Object` get a pass for a couple of somewhat arcane reasons)
- Tries to call a function using method-like syntax such as `Get-Service('something')`
- Tries to create a nameless variable (`${}`)

You can throw the strict mode setting right at the top of your functions to take advantage of these extra protections.

Getting Remote

Finally, PowerShell 4.0 and later supports *remote debugging*. This is useful when a script is running on a remote machine, which may also have modules and other dependencies that your local computer does not. Remote debugging makes it easier to debug a script in its "natural habitat," so to speak, since things like property values, OS features, and so on will differ from machine to machine. *Whenever possible, we try to debug a script when it's running on the same machine we plan to run it on in production.* Remote debugging aids in this tremendously, but it does require PowerShell v4 or later on both ends. The machine the script will run on must also be configured to accept Remoting sessions.

Really, you're just going to be using the standard `-PSBreakpoint` commands, but you'll work with them in a remote session that's connected to the machine where your script will run. Also note that this is designed (presently) to work in the PowerShell console, not the ISE.

1. Start by opening a remote session to the machine in question, by using `Enter-PSSession`.
2. Create breakpoints as usual by running `Set-PSBreakpoint`.
3. Run your script.

4. When you trigger a breakpoint, you'll have the usual `DBG\>` debugging console prompt.

In a remote debugging prompt, you'll have a new set of special commands:

- The `Help` command (`?`) lists all of these commands
- The `List` command will list your script's source
- The Show Call Stack ('k') command will show the current call stack
- The `Continue`, `StepInto`, and `StepOver` commands control debug execution

PowerShell 4.0 (and later) also supports disconnected sessions, and these are permitted for remote debugging. With this feature, you can disconnect a session that's in the `DBG\>` debugging prompt, and then later reconnect and resume debugging. You may actually run into this feature accidentally, as sometimes triggering a breakpoint will interrupt the remote session connection, forcing you to use `Enter-PSSession` or `Connect-PSSession` to reconnect.

Microsoft has a great blog article[29] with more examples on remote debugging.

Let's Review

We don't really have a challenge for you to try but we do want to make sure you picked up on a few key points.

1. What are the different breakpoint triggers?
2. What cmdlet can you use to help you avoid or minimize problems with something as simple as mistyping a variable name?

Review Answers

Did you come up with these answers?

1. Line number, command, or variable
2. Set-StrictMode

[29] https://blogs.technet.microsoft.com/heyscriptingguy/2013/11/17/remote-script-debugging-in-windows-powershell/

Command Tracing

This is another advanced debugging technique that we've used time and time again. It's absolutely invaluable for figuring out what PowerShell is doing with all the input you pass to a given command, whether via parameters or via the pipeline. As an example, we'll run through one of PowerShell's native commands, but this is just as useful for debugging your own commands.

Getting in PowerShell's Brain

Consider this command:

```
"g*","s*" | Get-Alias
```

It's our hope that the array of strings, `g*` and `s*`, will be connected to the `-Name` parameter of `Get-Alias`. But we want to *see* it happening. We want inside PowerShell's brain, to see it making that connection. Fortunately, PowerShell includes an X-Ray like cmdlet called `Trace-Command`. With this command we can look inside PowerShell and see what is happening.

```
trace-command -expression {"g*","s*" | Get-Alias } -name parameterbinding -pshost
```

We're telling PowerShell to *trace* the same command which we are defining inside a scriptblock. We've asked it specifically to show us *parameter binding* information in the host window; review the command's help for other things it can display. If you try this command, (and why wouldn't you), this generates a truly horrifying amount of output, so we'll run through the relevant chunks with you.

First up, we see that PowerShell always binds *named* parameters first, followed by positional ones. We didn't technically use either; we relied on pipeline input. PowerShell then checks to make sure all of the command's mandatory parameters have received input:

```
ParameterBinding Information: 0 : BIND NAMED cmd line args [Get-Alias]
ParameterBinding Information: 0 : BIND POSITIONAL cmd line args [Get-Alias]
ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet [Get-Alias]
```

This teaches us something: named and positional parameters will override pipeline input, because they happen before pipeline input is processed.

Next, PowerShell calls the `BEGIN` block of every command in the pipeline. This is how commands can "bootstrap" themselves, and it teaches us that `BEGIN` blocks won't have access to piped-in input values, because the pipeline hasn't been engaged yet.

```
ParameterBinding Information: 0 : CALLING BeginProcessing
```

Next, the shell starts working on the pipeline. It sees that the pipeline contains objects of the String type, and it looks for a parameter that can accept that type, ByValue, from the pipeline.

```
ParameterBinding Information: 0 : BIND PIPELINE object to parameters: [Get-Alias]
ParameterBinding Information: 0 :     PIPELINE object TYPE = [System.String]
ParameterBinding Information: 0 :     RESTORING pipeline parameter's original values
```

The shell finds the `-Name` parameter will meet the needs without converting, or *coercing*, the data into another type, and so it attaches, or *binds*, the first input value, `g*`, to the `-Name` parameter. This behavior confirms that only one value at a time is sent through the pipeline.

```
ParameterBinding Information: 0 :     Parameter [Name] PIPELINE INPUT ValueFromPipel\
ine NO COERCION
ParameterBinding Information: 0 :     BIND arg [g*] to parameter [Name]
```

We see that `-Name` expects an array of values, but the pipeline only contains one value. So PowerShell creates a single-item array, and then attaches it to the parameter.

```
ParameterBinding Information: 0 :         Binding collection parameter Name: argumen\
t type [String], parameter type [System.String[]], collection type Array, element ty\
pe [System.String], no coerceElementType
ParameterBinding Information: 0 :         Creating array with element type [System.S\
tring] and 1 elements
ParameterBinding Information: 0 :         Argument type String is not IList, treatin\
g this as scalar
ParameterBinding Information: 0 :         Adding scalar element of type String to ar\
ray position 0
```

PowerShell then runs the command's parameter validation attributes, if any, and we see that the value binding was successful.

```
ParameterBinding Information: 0 :         Executing VALIDATION metadata: [System.Man\
agement.Automation.ValidateNotNullOrEmptyAttribute]
ParameterBinding Information: 0 :         BIND arg [System.String[]] to param [Name]\
 SUCCESSFUL
ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet [Get-Alias]
```

There's another value in the pipeline, so the process repeats:

```
ParameterBinding Information: 0 : BIND PIPELINE object to parameters: [Get-Alias]
ParameterBinding Information: 0 :     PIPELINE object TYPE = [System.String]
ParameterBinding Information: 0 :     RESTORING pipeline parameter's original values
ParameterBinding Information: 0 :     Parameter [Name] PIPELINE INPUT ValueFromPipel\
ine NO COERCION
ParameterBinding Information: 0 :     BIND arg [s*] to parameter [Name]
ParameterBinding Information: 0 :         Binding collection parameter Name: argumen\
t type [String], parameter type [System.String[]], collection type Array, element ty\
pe [System.String], no coerceElementType
ParameterBinding Information: 0 :         Creating array with element type [System.S\
tring] and 1 elements
ParameterBinding Information: 0 :         Argument type String is not IList, treatin\
g this as scalar
ParameterBinding Information: 0 :         Adding scalar element of type String to ar\
ray position 0
ParameterBinding Information: 0 :         Executing VALIDATION metadata: [System.Man\
agement.Automation.ValidateNotNullOrEmptyAttribute]
ParameterBinding Information: 0 :         BIND arg [System.String[]] to param [Name]\
 SUCCESSFUL
```

`Trace-Command` can return a wealth of information based on type of information you are looking for. This is what the -Name parameter is providing. Run `Get-TraceSource` to see all of your options. Or to get the complete picture, run a trace command like this:

```
trace-command -expression {"g*","s*" | Get-Alias } -name * -pshost
```

The Debug output should show you everything PowerShell is doing when processing your expression. If you prefer to save the trace information to a file use the -FilePath parameter.

```
trace-command -expression {"g*","s*" | Get-Alias } -name * -filepath trace.txt
```

Command tracing is a useful tool for seeing *exactly* how PowerShell is dealing with parameter input, and has helped us out of many sticky situations by helping us better understand what's happening in PowerShell's head.

Analyzing Your Script

One of the neater projects that have come out of Microsoft is the PowerShell Script Analyzer. This is available as `PSScriptAnalyzer` in PowerShell Gallery (meaning you can use `Install-Module` to install it). It's a static code analyzer, which means it doesn't *run* your code; it merely *gazes upon* your code and offers suggestions related to best practices, coding style, and so on. It can analyze code inside .ps1 and .psm1 files.

Performing a Basic Analysis

We're going to start with the code from our "Extending Output Types" chapter, but we'll provide a standalone copy as `Script.ps1` in the downloadable code for this chapter (just as a convenience, if you're following along).

The PowerShell Script Analyzer consists of a set of *rules*. You can run an analysis using only specific rules, or excluding certain rules. We'll run against the full rule set.

You can also create custom rules, and there are a few community projects that define new Script Analyzer rules for various purposes.

```
PS C:\analyzing-your-script> Invoke-ScriptAnalyzer .\Script.ps1
PS C:\analyzing-your-script>
```

WHAAAT? That's awesome! "No news is good news," meaning the analysis didn't find anything it felt it needed to recommend. WE ARE AMAZING CODERZ. Well… sort of. You see, the analysis is only as good as the rules it supports. Let's do this: we'll add a $Password parameter to our script.

```
[cmdletbinding()]
Param(
    [Parameter(ValueFromPipeline, Mandatory)]
    [ValidateNotNullorEmpty()]
    [Alias("CN", "Machine", "Name")]
    [string[]]$Computername,
    [string]$ErrorLog,
    [switch]$ErrorAppend,
    [string]$Password
)
```

And try again:

```
PS C:\analyzing-your-script> Invoke-ScriptAnalyzer .\Script.ps1 |
Select -expand message

Parameter '$Password' should use SecureString, otherwise this will exp
ose sensitive information. See ConvertTo-SecureString for more informa
tion.
```

The Analyzer has a rule about parameters named $Password which accept a `[string]`, because that implies you're passing passwords in clear text, which is obviously a Bad Idea. We triggered that rule, so you can see what it does. The Analyzer presently comes with just under 50 rules. You can see them all by running `Get-ScriptAnalyzerRule`.

Analyzing the Analysis

A thing to remember is that the Analyzer can't catch every possible bad thing you might do in your code. It's largely just matching regular expressions against known problem conditions, and alerting you to them. But it's a good "first pass" on making sure you haven't egregiously broken any really obvious best practices.

The rule collection has been cultivated over the years by a group of PowerShell subject matter experts and MVPs, many of them drawn from community best practices. However, you do not need to treat them as gospel. We've encountered warnings that don't take into account what the rest of the command might be doing or how the command will be used. But for beginners, the rules do make a good sanity check.

If you plan on publishing your project to the PowerShell Gallery, you will want to make sure you are compliant with the script analyzer. Microsoft will run your submission through the analyzer automatically and kick it back to you if there are problems.

If you are interested in learning more about this tool, head over to the project's GitHub repository at https://github.com/PowerShell/PSScriptAnalyzer[30].

Your Turn

Let's see how Script Analyzer can help improve your code.

Start Here

In the downloadable sample code for this chapter, we've provided you with a Start.ps1 script. Load it up in the ISE and take a look at it.

[30] https://github.com/PowerShell/PSScriptAnalyzer

Start.ps1

```
function Query-Disks {
    [CmdletBinding(SupportsShouldProcess)]
    Param(
        [Parameter(Mandatory)]
        [string[]]$ComputerName = 'localhost'
    )
    foreach ($comp in $computername) {
        $logfile = "errors.txt"
          write-host "Trying $comp"
       try {
            gcim win32_logicaldisk -comp $comp -ea stop
      } catch {

        }}
}
```

Your Task

Use Script Analyzer to analyze the script. Improve the script based on the Script Analyzer's feedback.

Our Take

We've presented a possible solution in `Improved.ps1`, located in the same folder as the script you analyzed. This addresses all of Script Analyzer's concerns - try running an analysis and see what you get.

Analyzing `Start.ps1`, we had the following complaints:

- Mandatory Parameter 'ComputerName' is initialized in the Param block. To fix a violation of this rule, please leave it unintialized.
- File 'Start.ps1' uses Write-Host. Avoid using Write-Host because it might not work in all hosts, does not work when there is no host, and (prior to PS 5.0) cannot be suppressed, captured, or redirected. Instead, use Write-Output, Write-Verbose, or Write-Information.
- The cmdlet 'Query-Disks' uses an unapproved verb.
- The variable 'logfile' is assigned but never used.
- 'Query-Disks' has the ShouldProcess attribute but does not call ShouldProcess/ShouldContinue.
- The cmdlet 'Query-Disks' uses a plural noun. A singular noun should be used instead.
- 'gwmi' is an alias of 'Get-WmiObject'. Alias can introduce possible problems and make scripts hard to maintain. Please consider changing alias to its full content.

- Empty catch block is used. Please use Write-Error or throw statements in catch blocks.

Here's what we did to address them:

- We removed the default value for `-ComputerName`. It would never be used anyway, as the parameter was marked as mandatory.
- We switched `Write-Host` to `Write-Verbose`, thus saving a puppy.
- We changed our verb to the approved `Get` verb, and our noun to a singular.
- We added error logging in the `Catch` block.
- We got rid of `Get-WmiObject`, resolving the alias complaint. The next complaint would have been to not use the deprecated WMI commands, so we switched to `Get-CimInstance`.
- We removed the `SupportsShouldProcess` attribute. We see a lot of people throw that in when it isn't needed, and in this case, it isn't.

We also cleaned up the indentation, which Analyzer should honestly have complained about, but didn't, when we ran it.

Result.ps1

```
function Get-Disk {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory)]
        [string[]]$ComputerName
    )
    foreach ($comp in $computername) {
        $logfile = "errors.txt"
        Write-Verbose "Trying $comp"
        try {
            Get-CimInstance -ClassName win32_logicaldisk -ComputerName $comp -ea stop
        } catch {
            $comp | Out-File $logfile -Append
        }
    }
}
```

Once huge advantage and reason to use VS Code as your primary development tool is that the Script Analyzer is built in. You get real time scanning as you write you code. And if there is an update to rules, you should get them in the next VS Code update.

Controlling Your Source

We're going to go out on a limb and say that if are spending time and energy in creating a PowerShell-based tool, you would hate to see all that work go to waste or get lost. Yet for many IT Pros that is the exactly the risk they are taking every day. The real reason is that for the longest time IT Pros, and often their managers, treated scripting as an ad-hoc and throwaway activity. We cranked out a script to solve an immediate problem then went on to the next fire.

Recently though, IT Pros and their more enlightened managers, have come to understand that scripting and automation are key components to how they run their organization. For groups moving to a DevOps paradigm this is even more important. Even if you aren't moving to the DevOps model, you need to begin thinking like a developer. The effort you are investing in your module or toolset is just as important as a developer in your company working on a new application.

This means you need to place equal importance on documentation, testing and source control which is the focal point of this chapter.

The Process

When we talk about *source control* the name should say it all. You need to have a mechanism to **control** the **source code** of your PowerShell tool. It doesn't matter if you are writing a PowerShell function in Notepad or developing a full-blown module in Visual Studio Code. It also doesn't matter if you are developing PowerShell solutions in a collaborative environment, or working alone. You need to protect yourself with source control.

The important thing to understand is that source control is a model. There are many, many ways to implement it (we'll review a few in a moment) but all solutions incorporate these concepts

- Check in
- Check out
- Version history
- Rollback

In short you write some code and check it in to a source control system. Later you, or a teammate, can check out the code for further work. The changed code is put back into source control, often with some description about what changed and why. This versioning information is what makes it possible to go back to earlier versions. Again, this description is merely intended as a generic overview.

Tools and Technologies

Source control tools generally fall into two categories, centralized and de-centralized. A centralized system tends to have a central server or repository that everyone connects to get and put code. Often in a centralized system only one person can work on a given piece of code at once. No one else can make any changes while the code is checked out. Visual Source Safe is a good example.

In a decentralized system, everyone has a copy and anyone can make any changes they want. Of course, there needs to be a mechanism to synchronize everyone's code and handle conflicts. Git is perhaps the best well known example of this model.

We're not going to tell you what to use. Your company may already have a source control mechanism in place that you will want, or have, to use. But you should use something. Between the two of use we've used a variety of source control platforms over the years. Yes, there will be a bit of a learning curve but accept it as the cost of being a professional PowerShell toolmaker. The point is, if you are **not** using some sort of source control today, you should be.

Here are a source control solutions you might consider. Obviously there are many on the market and we aren't recommending anything.

git and GitHub

Without a doubt one of the most popular source control systems today is *git*. This is an open source product originally developed to manage the source code for the Linux kernel so you can imagine how robust this has to be. In the git model, you have a repository of all related files for your project. The repository itself is handled with some complex file system voodoo which you don't have to worry too much about.

In a git environment you *commit* your changed files to the repository. Other people can *clone* your repository which gives them a working copy. They can then *fetch* and *pull* any of your changes. Otherwise, changes they make locally are committed to their repository which can be *pushed* to the remote repository. Depending on permissions, they might send you a *pull request* which basically says, "I made some changes you might like to have so pull them from my repository."

For many IT Pros, this collaboration is implemented through the GitHub web site. This free service lets you set up your own repositories which other people can clone or *fork*. It is entirely possible to manage everything from the web, but most people will have a Github master repository and a local clone. This way you can work and test your code locally and push changes to GitHub.

For example, Jeff wrote a PowerShell tool that adds remote tab functionality to the PowerShell ISE. The tool has been published to the PowerShell Gallery as ISRemoteTab but the source code is an open source project on GitHub. Locally, he has a copy of the files in git repository which is configured with a remote branch.

```
PS S:\ISERemoteTab> git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
PS S:\ISERemoteTab> git branch
  dev
* master
* PS S:\ISERemoteTab> git log -1
commit cd473151cf24c15304fdb21acf2e99147918192b
Author: jdhitsolutions <jhicks@jdhitsolutions.com>
Date:   Thu Jan 12 12:18:27 2017 -0500

    revised license
PS S:\ISERemoteTab> git remote
origin
PS S:\ISERemoteTab> git remote -v
origin  https://github.com/jdhitsolutions/New-ISERemoteTab.git (fetch)
origin  https://github.com/jdhitsolutions/New-ISERemoteTab.git (push)
```

When he makes changes he can commit them locally and then push them to Github.

As you might expect there is much to learn about using git and GitHub. Fortunately, because these platforms are so widely used there is a ton of reference and training material online. You can also find a number of PowerShell related projects in the PowerShell gallery:

```
find-module -tag git -Repository PSGallery
```

Jeff also has a PowerShell function to create a GitHub repository from the command line which he blogged about[31].

You can get started as well as download the current version of git[32].

Azure DevOps Server

Formerly known as TeamFoundationServer or TFS, is a Microsoft product related to its Visual Studio product line. The new version is an Azure service that allows you to track and share code in team or collaborative environment. Because it runs in Azure, some of the setup is not as complicated as the older TeamFoundationServer product. Microsoft offers a free version for individuals or small teams that can run on your local computer.

Learn more about Azure DevOps Server[33].

[31] bit.ly/2jgskzo
[32] https://git-scm.com/
[33] https://azure.microsoft.com/en-us/services/devops/server/

Subversion

Another popular source control system is Subversion, also known as SVN. This is an open source project from the Apache Software Foundation. SVN is similar to git in that you have a repository where you can commit changes. You can also have multiple *branches* for different development efforts.

You use a subversion client to check code in and out of the repository. SVN only maintains command line clients but you can find a number of graphical clients online.

Learn more about Subversion[34].

Mercurial

One last project we want to at least introduce you to is Mercurial. This is a decentralized source control system based on Python. It is similar to git in that you can have a server based master repository and local versions. Even though the underlying technology differs you have the same concepts of forking, committing, pulling and pushing. Typically the server component is hosted by a site like Bitbucket which has free and paid accounts. You would then use the Mercurial client to interact with the local and remote repositories.

You can find a number of PowerShell-related modules in the Chocolatey gallery if you have that defined.

```
find-module -tag mercurial -Repository chocolatey
```

And lest you think Mercurial can't handle your PowerShell module, Facebook apparently uses Mercurial for its source control!

You can learn more and download Mercurial[35].

Let's Review

We hope you gleaned a few tidbits from this chapter. Let's check.

1. True or False: source code is for developers only
2. What are the two different source control models?
3. What are some of the benefits of using source control?

[34] http://subversion.apache.org/
[35] https://www.mercurial-scm.org/

Review Answers

Did you come close to these answers?

1. A very big FALSE. Even as an IT Pro you need to start thinking and acting like a developer.
2. Centralized and de-centralized.
3. Built-in versioning and history so that you can roll back to a prior version if necessary. In a team environment you can usually track who made what change and when so there is accountability. Finally, source control can serve as a backup mechanism, especially if you have a remote version somewhere of your local repository.

Converting a Function to a Class

Classes were a new feature introduced in PowerShell v5. They've continued to evolve since then, and we get asked about them all the time - hence, this chapter. On the surface, they're *really really* similar to a function, and so converting a function to a class is usually straightforward although a more accurate description might be *transforming*.

Class Background

But before we go any further, let's talk about what these are and set some expectations. We see a lot of people diving into classes because they're the new shiny and because all the "real" developers are using them. There are reasons to use classes, and certain things they do, but they will not magically make your PowerShell commands "better" somehow.

Most of the time, you will find that functions are entirely adequate for creating the PowerShell commands you need. Classes come into play when you need to create a much more formal programming structure that requires object-oriented programming features. And, if you've used classes in other languages, know that PowerShell ain't other languages. Its classes are their own things, and assuming that they "just work" like some other language will lead you to a bad, dark place in life. Classes were introduced *mainly* to improve DSC resource authoring, and if you're working outside the DSC space (which we don't touch on in this book), classes are less "polished" in some ways than you might hope. For example, debugging classes is a bit trickier prior to PowerShell 5.1, which improved debugging support.

A **Class** is kind of like a blueprint of something, or a template. A blueprint is wonderful, but you can't live in one, right? When you create an *instance* of the blueprint - that is, a house - you have something to live in. And that same blueprint can create multiple instances, giving you multiple, identical houses. The class simply describes what your *thing* should look like and how it might behave.

A **Property** is one of the **members** that a class can contain. A property contains a bit of information, and it may permit you to read that information, change the information, or both. Reading and writing - that is, *getting* and *setting* - are accomplished by two hidden methods. You don't need to worry about these when *using* an object, because as you've seen in working with PowerShell, .NET Framework just "makes it work." However, when creating a class, you sometimes do need to worry about these "getter" and "setter" methods, although PowerShell does handle them for you if you just need basic implementations. A property consists of a data type, a name, and sometimes a default value.

A **Method** is another member that a class can contain. A method tells the class to take some action - basically, it's a mini-function contained within the class. Like functions, methods can accept input arguments, and they can produce results.

Think of it this way: you may have made a module to help manage some line-of-business application. Your module contains commands like `Get-AppUser`, `Set-AppUser`, `Remove-AppUser`, and `New-AppUser`. Alternatively, you could create an `AppUser` class. It would contain methods for retrieving users, changing their attributes, deleting them, and creating them. The code would look remarkably familiar either way, but the class structure is more formal and a bit more complex than the module structure, which is just a bunch of functions.

A **Constructor** is a special method that's used to create a new instance of a class. Constructs can accept input parameters. So, using the example above, you might be able to run AppUser('username') to create a new instance of your AppUser class, pre-populated with a given user's information.

Here's a very simple class definition:

```
class AppUser {
    # Properties
    [string]$UserName
    [int]$EmployeeID

    # Constructors
    AppUser () {
        #your code
    }
    AppUser ([string]$UserName) {
        #your code
    }

    # Methods
    [void] Delete() {
        #your code
    }
    [void] Update() {
        #your code
    }

}
$x = New-Object -Type AppUser
```

We've defined a class with two properties, two ways of instantiating it, and two methods. This is obviously just the framework; we'd need to add code to make all this work. At the bottom, you'll see where we instantiated the class using `New-Object`. You can also create a new instance by invoking the built-in New() static method: `[AppUser]::new()`.

This brings up an interesting point. Right now, PowerShell doesn't "know" where your classes live. It's not like functions where, if they're stored in a module that's in the right folder, PowerShell can

magically load up the function on-demand. With classes, you have to make sure the class definition is loaded, or PowerShell won't know what it is. For example, in the above, we're *using* the class in the same script that *defines* the class, which will work. This can make classes a bit less convenient. Most of the time, you're stuck with dot-sourcing the class definition into whatever script needs to use it.

PowerShell supports **inheritance**. That means you could take the AppUser class we created, and *inherit* it in your own class definition. Your definition could add new properties and methods, and the ones we created would still function.

There's a great overview[36] of classes that goes into more detail and is especially useful if you have a background in another class-based language. We're glossing over some fine detail and a lot of permutations in this chapter.

Another key thing in classes is the `return` keyword. In a normal PowerShell function, `return` is basically an alias to `Write-Output`: it writes objects to the pipeline. In a class method, however, `return` writes to the pipeline *and then exits the method immediately*. This is consistent with the keyword's behavior in almost every other programming language, ever. In a PowerShell class, you must specify the type of object the method will emit, if any, and use the return keyword. In our example, the two methods don't write anything to the pipeline so we use `[void]`. But if we want a method to write something to the pipeline we need to specify the datatype and return it.

```
[timespan]GetAge() {
      $t = <code to calculate timespan>
      return $t
    }
```

A major upside to classes is that *they are objects* (well, an instance of a class is an object). So instead of your functions outputting "static" objects that only have properties, which only contain static information, classes can be very dynamic. You could create a class that was capable of refreshing its property values, for example, or that provided helpful methods for working with whatever it is the object represents. However, if all your command needs to do is *do* something, or produce static output, then classes can be harder to work with than functions while giving you no advantages. There's a good introductory writeup[37] on classes that creates a Computer object, essentially creating a wrapper around some existing AD commands. It's a good introduction to class syntax, but it doesn't create a lot of functional advantages over just running commands - that's important to realize, from a design perspective.

With all that in mind, *we almost never "convert" a function into a class.* A class is something we kind of design. But, we're going to go through the "conversion" routine here, because it's a useful way of taking something we've already done with you, and leveraging that knowledge to do something new. So think of this as "conversion for teaching's sake," rather than, "oh, yeah, we convert all the time." OK?

[36] https://xainey.github.io/2016/powershell-classes-and-concepts/
[37] http://powershelldistrict.com/powershell-class/

Starting Point

We're going to take the function from the end of the error handling chapter as our starting point. It's here for your reference, and more easily readable in the code downloads for this chapter. It is very similar to the `Get-TMComputerStatus` function.

```
Function Get-MachineInfo {

[CmdletBinding()]
Param(
    [Parameter(ValueFromPipeline,
        Mandatory)]
    [Alias('CN', 'MachineName', 'Name')]
    [string[]]$ComputerName
)

BEGIN {}

PROCESS {
    foreach ($computer in $computername) {
        Try {
            Write-Verbose "Connecting to $computer"
            $params = @{
                ComputerName  = $Computer
                ErrorAction   = 'Stop'
            }

            $session = New-CimSession @params

            Write-Verbose "Querying $computer"
            #define a hashtable of parameters to splat
            #to Get-CimInstance
            $cimparams = @{
                ClassName  = 'Win32_OperatingSystem'
                CimSession = $session
                ErrorAction = 'stop'
            }
            $os = Get-CimInstance @cimparams

            $cimparams.Classname = 'Win32_ComputerSystem'
            $cs = Get-CimInstance @cimparams
```

```
            $cimparams.ClassName  = 'Win32_Processor'
            $proc = Get-CimInstance @cimparams | Select-Object -first 1

            $sysdrive = $os.SystemDrive
            $cimparams.Classname = 'Win32_LogicalDisk'
            $cimparams.Filter = "DeviceId='$sysdrive'"
            $drive = Get-CimInstance @cimparams

            Write-Verbose "Outputting for $($session.computername)"
            $obj = [pscustomobject]@{
                ComputerName      = $session.computername.ToUpper()
                OSVersion         = $os.version
                OSBuild           = $os.buildnumber
                Manufacturer      = $cs.manufacturer
                Model             = $cs.model
                Processors        = $cs.numberofprocessors
                Cores             = $cs.numberoflogicalprocessors
                RAM               = $cs.totalphysicalmemory
                Architecture      = $proc.addresswidth
                SystemFreeSpace   = $drive.freespace
            }
            Write-Output $obj

            Write-Verbose "Closing session to $computer"
            $session | Remove-CimSession
        }
        Catch {
            Write-Warning "FAILED to query $computer. $($_.exception.message)"
        }
    } #foreach

} #PROCESS

END {}

} #end function

Get-MachineInfo -ComputerName $env:COMPUTERNAME
```

It's worth noting that classes don't support comment-based help, nor are they supported by the help system at all, so we're going to lose that. We'd need to publish instead some kind of "about" help file along with our code to document how to use it.

Doing the Design

We need to look at this function and decide on a class design.

- We will call the class `TMMachineInfo`. It is a Bad Idea to choose an existing class name (and there are quadrillions) for your new class.
- We have a list of 10 properties that our class will expose. These are the 10 properties that our function's output objects contain.
- We will have a constructor that accepts a computer name as its input argument. Note that our design will only query one computer at a time; this is the usual pattern for classes because each instance of the class can only represent one thing. If we need to query multiple computers, we'd write a script that created multiple instances of the class, using something like a `ForEach` loop.

You can see that we're moving *some* of our functionality into the class, but not everything. Classes are meant to be pretty tightly scoped, and should only include things that *represent* something. In our case, the class *represents* machine information *for a single machine*, and so we're scoping our functionality to that.

Making the Class Framework

Here's our basic framework (in ClassFramework.ps1 in the sample code):

```
class TMMachineInfo {

    # Properties
    [string]$ComputerName
    [string]$OSVersion
    [string]$OSBuild
    [string]$Manufacturer
    [string]$Model
    [string]$Processors
    [string]$Cores
    [string]$RAM
    [string]$SystemFreeSpace
    [string]$Architecture

    # Methods
    # none at this time

    # Constructors
    TMMachineInfo([string]$ComputerName) {
```

```
        # insert code to create a new instance of $this
    }

} #class definition
```

Some notes:

- Even though this *looks* like a function, the properties are not parameters so don't try to separate them by commas.
- All of our properties will be *writable*, which is not really correct. We can't change the RAM on a machine just by setting this property, so it should be read-only. Unfortunately, without getting into some odd, convoluted code, read-only properties aren't currently a thing. Users will be able to change any of our properties, and those changes will *appear to have "taken"*, but they will obviously not change the actual environment. If you really want to limit access, there is an option to mark a property as hidden.
- Technically, adding a constructor is entirely optional. An instance of The class can still be created. But since the class makes no sense without the computer name we're going to build a specific constructor.
- There is no provision for defining a default value in the constructor.

It's possible to create our own "setter," rather than using PowerShell's implied one, for each property. We could then throw an error if someone tried to set the property. That's… *interesting*, but not the same as creating a read-only property.

It's also worth noting that, inside the constructor, `$ComputerName` is the argument passed to the constructor. `$this.ComputerName` would be used to modify the property of the class. That can be a confusing scoping issue, and it's worth paying attention to.

Coding the Class

You'll find this in the downloadable code as Coded.ps1.

```
class TMMachineInfo {

    # Properties
    [string]$ComputerName
    [string]$OSVersion
    [string]$OSBuild
    [string]$Manufacturer
    [string]$Model
    [string]$Processors
    [string]$Cores
    [string]$RAM
    [string]$SystemFreeSpace
    [string]$Architecture
    hidden[datetime]$Date

    # Constructors
    TMMachineInfo([string]$ComputerName) {

        Try {
           $params = @{
               ComputerName  = $Computername
               ErrorAction   = 'Stop'
           }

           $session = New-CimSession @params

           #define a hashtable of parameters to splat
           #to Get-CimInstance
           $cimparams = @{
               ClassName  = 'Win32_OperatingSystem'
               CimSession = $session
               ErrorAction = 'stop'
           }
           $os = Get-CimInstance @cimparams

           $cimparams.Classname = 'Win32_ComputerSystem'
           $cs = Get-CimInstance @cimparams

           $cimparams.ClassName  = 'Win32_Processor'
           $proc = Get-CimInstance @cimparams | Select-Object -first 1

           $sysdrive = $os.SystemDrive
           $cimparams.Classname = 'Win32_LogicalDisk'
```

```
            $cimparams.Filter = "DeviceId='$sysdrive'"
            $drive = Get-CimInstance @cimparams

            $session | Remove-CimSession
            #use the computername from the CIM instance
            $this.ComputerName = $os.CSName
            $this.OSVersion = $os.version
            $this.OSBuild = $os.buildnumber
            $this.Manufacturer = $cs.manufacturer
            $this.Model = $cs.model
            $this.Processors = $cs.numberofprocessors
            $this.Cores = $cs.numberoflogicalprocessors
            $this.RAM = ($cs.totalphysicalmemory / 1GB)
            $this.Architecture = $proc.addresswidth
            $this.SystemFreeSpace = $drive.freespace
            $this.date = Get-Date
        }
        Catch {
            throw "Failed to connect to $computername. $($_.exception.message)"
        } #try/catch
    }
} #class

New-Object -TypeName TMMachineInfo -ArgumentList "localhost"
```

We've included a line at the end of the script to try it out, which gave us:

```
ComputerName    : BOVINE320
OSVersion       : 10.0.19041
OSBuild         : 19041
Manufacturer    : LENOVO
Model           : 30C2CTO1WW
Processors      : 1
Cores           : 8
RAM             : 31.8933715820313
SystemFreeSpace : 79033331712
Architecture    : 64
```

We decided to add an additional property called Date but it is hidden, although it's only "lightly" hidden. Passing our object to `Get-Member -force` can still show the hidden property. This is just a display convenience, not a security thing. It won't show up in the default display, but it can be accessed by the property name.

```
PS C:\> New-Object -TypeName TMMachineInfo -argumentlist ThinkP1 | Select Date,Compu\
ternname,OS*

Date                 ComputerName OSVersion  OSBuild
----                 ------------ ---------  -------
6/3/2020 12:52:34 PM THINKP1      10.0.18363 18363
```

So it works - but because a class is a lower-level beastie than a function, in many ways, we've "lost" some functionality. We'd have to make up for that in the script itself. Notice that, to create our output, we simply set the properties of `$this`, which represents *the current instance of the class.*

Adding a Method

Let's try one more thing. Now, this isn't because we think this is a good idea - it's more to show you a couple of things. First, here's a revision to our code.

```
class TMMachineInfo {

# Properties
[string]$ComputerName
[string]$OSVersion
[string]$OSBuild
[string]$Manufacturer
[string]$Model
[string]$Processors
[string]$Cores
[string]$RAM
[string]$SystemFreeSpace
[string]$Architecture
hidden[datetime]$Date

#Methods
[void]Refresh() {

    Try {
        $params = @{
                ComputerName  = $this.Computername
                ErrorAction   = 'Stop'
                }

            $session = New-CimSession @params
```

```
            #define a hashtable of parameters to splat
            #to Get-CimInstance
            $cimparams = @{
                ClassName  = 'Win32_OperatingSystem'
                CimSession = $session
                ErrorAction = 'stop'
            }
            $os = Get-CimInstance @cimparams

            $cimparams.Classname = 'Win32_ComputerSystem'
            $cs = Get-CimInstance @cimparams

            $cimparams.ClassName  = 'Win32_Processor'
            $proc = Get-CimInstance @cimparams | Select-Object -first 1

            $sysdrive = $os.SystemDrive
            $cimparams.Classname = 'Win32_LogicalDisk'
            $cimparams.Filter = "DeviceId='$sysdrive'"
            $drive = Get-CimInstance @cimparams

        $session | Remove-CimSession

        #use the computername from the CIM instance
        $this.ComputerName = $os.CSName
        $this.OSVersion = $os.version
        $this.OSBuild = $os.buildnumber
        $this.Manufacturer = $cs.manufacturer
        $this.Model = $cs.model
        $this.Processors = $cs.numberofprocessors
        $this.Cores = $cs.numberoflogicalprocessors
        $this.RAM = ($cs.totalphysicalmemory / 1GB)
        $this.Architecture = $proc.addresswidth
        $this.SystemFreeSpace = $drive.freespace
        $this.date = Get-Date
    }
    Catch {
        throw "Failed to connect to $this.computername. $($_.Exception.message)"
        } #try/catch
    }

# Constructors
TMMachineInfo([string]$ComputerName) {
    $this.ComputerName = $ComputerName
```

```
    $this.Refresh()
}
} #class

Clear-Host
$obj = New-Object -TypeName TMMachineInfo -ArgumentList "localhost"
$obj | Select Date,Computername,SystemFreeSpace
Get-Process | Out-File "$Home\delete_me.txt"
$obj.Refresh()
$obj | Select Date,Computername,SystemFreeSpace
```

- In our constructor, we've removed most of the code. The constructor now passes its arguments into properties, and calls a new `Refresh()` method.
- The `Refresh()` method has all of our original code, although we now use `$this` to access the computer name we're supposed to query

In the script, we're writing a tiny text file out, just to change the free space number on the drive:

```
PS C:\> $obj = New-Object -TypeName TMMachineInfo -ArgumentList "localhost"
PS C:\> $obj | Select Date,Computername,SystemFreeSpace
PS C:\> Get-Process | Out-File "$Home\delete_me.txt"
PS C:\> $obj.Refresh()
PS C:\> $obj | Select Date,Computername,SystemFreeSpace

Date                ComputerName SystemFreeSpace
----                ------------ ---------------
6/3/2020 1:10:21 PM BOVINE320    79029981184
```

As you can see, `Refresh()` does indeed re-query the information. The method doesn't need to return anything, because all it's doing is changing the properties of the instance itself.

Making Classes Easy To Use

The scripts we've been showing you are for educational purposes. In order to use our class definitions you would need to know how to use `New-Object` and know the type name. But that might be a bit much to ask of a help desk user you want to use your tool. Instead, you might give them a command like `Get-MachineInfo` which still uses all of the class goodness, but hides all the messy dev-stuff.

In the chapter download file you will find a module version of the code we've been using called `TMMachineInfo`. If you look at the psm1 file you will find the class definition and two functions. The first function is essentially a wrapper for the New() constructor:

```
Function Get-MachineInfo {
    [cmdletbinding()]
    [alias("gmi")]
    Param(
        [Parameter(Position = 0, ValueFromPipeline)]
        [Alias("cn")]
        [ValidateNotNullorEmpty()]
        [string[]]$Computername = $env:COMPUTERNAME
    )

    Begin {
        Write-Verbose "[BEGIN  ] Starting: $($MyInvocation.Mycommand)"
    } #begin

    Process {
        foreach ($computer in $computername) {
            Write-Verbose "[PROCESS] Getting machine information from $($computer.to\
Upper())"
            New-Object -TypeName TMMachineInfo -ArgumentList $computer
        }
    } #process

    End {
        Write-Verbose "[END    ] Ending: $($MyInvocation.Mycommand)"
    } #end
}
```

By creating a function around it we can parameterize it, support processing pipelined input and more. The function will write an object to the pipeline that has a type name based on the class name.

```
PS C:\> $c = get-machineinfo
PS C:\> $c | get-member

   TypeName: TMMachineInfo

Name            MemberType Definition
----            ---------- ----------
Equals          Method     bool Equals(System.Object obj)
GetHashCode     Method     int GetHashCode()
GetType         Method     type GetType()
Refresh         Method     void Refresh()
ToString        Method     string ToString()
```

```
Architecture    Property   string Architecture {get;set;}
ComputerName    Property   string ComputerName {get;set;}
Cores           Property   string Cores {get;set;}
Manufacturer    Property   string Manufacturer {get;set;}
Model           Property   string Model {get;set;}
OSBuild         Property   string OSBuild {get;set;}
OSVersion       Property   string OSVersion {get;set;}
Processors      Property   string Processors {get;set;}
RAM             Property   string RAM {get;set;}
SystemFreeSpace Property   string SystemFreeSpace {get;set;}
```

We didn't want to force the user to have to invoke an object method. Instead we wrote a function.

```
Function Update-MachineInfo {
    [cmdletbinding()]
    [alias("umi")]
    Param(
        [Parameter(Position = 0, ValueFromPipeline)]
        [ValidateNotNullorEmpty()]
        [TMMachineInfo]$Info,
        [switch]$Passthru
    )

    Begin {
        Write-Verbose "[BEGIN  ] Starting: $($MyInvocation.Mycommand)"
    } #begin

    Process {
        Write-Verbose "[PROCESS] Refreshing: $(($Info.ComputerName).ToUpper())"
        $info.Refresh()

        if ($Passthru) {
            #write the updated object back to the pipeline
            $info
        }
    } #process

    End {
        Write-Verbose "[END    ] Ending: $($MyInvocation.Mycommand)"
    } #end
}
```

Again, a function offers the benefits of help documentation, parameters, validation and verbose

output. We are even able in the function to write the updated object back to the pipeline with the `-Passthru` parameter, even though the `Refresh()` method doesn't return anything. You should also notice the typename on the `-Info` parameter.

```
[TMMachineInfo]$Info,
```

We wrote this thinking the help desk might use the command to get some information:

```
$info = get-content c:\work\servers.txt | get-machineinfo
```

Then later in the day, they update it:

```
$info | update-machineinfo -passthru
```

We think a key takeaway is that in some ways the class simplified the process of working with objects in PowerShell. But you need to think about *how* they will be used which might mean some additional tooling in the form of some wrapper or "helper" functions.

Wrapping Up

Classes have their place. The ones in PowerShell are a little under-baked compared to most languages, and understanding that classes aren't a direct replacement for functions is important. Classes are lower-level, meaning PowerShell does less (like validation) *for* you. They're also a different way of packaging functionality, and they don't make sense in every case. In fact, as we admitted upfront, in our example above *classes didn't make sense*. Our function did a much better job of accomplishing the job. But now you can at least get a side-by-side comparison of functions and classes, and hopefully a real feel for when they may make sense or not.

If you want to play with PowerShell class-based tools further or see how you might build a tool around a class, you might want to take a look at Jeff's PSChristmas module on Github at https://github.com/jdhitsolutions/PSChristmas[38] or the myStarship[39] module. Yes, a little silly but hopefully educational.

[38] https://github.com/jdhitsolutions/PSChristmas
[39] https://github.com/jdhitsolutions/myStarShip

Publishing Your Tools

Inevitably, you'll come to point where you're ready to share your tools. Hopefully, you've put those into a PowerShell module, as we've been advocating throughout this book, because in most cases it's a module that you'll share.

Begin with a Manifest

You'll typically need to ensure that your module has a .psd1 manifest file, since most repositories will use information from that to populate repository metadata. Here's the manifest from our downloadable sample code.

```
#
# Module manifest for module 'PowerShell-Toolmaking'
#
# Generated by: Don Jones & Jeffery Hicks
#

@{

# Script module or binary module file associated with this manifest.
RootModule = 'PowerShell-Toolmaking.psm1'

# Version number of this module.
ModuleVersion = '2.0.0.0'

# Supported PSEditions
CompatiblePSEditions = @("Desktop")

# ID used to uniquely identify this module
GUID = '3926b244-469c-4434-a4b1-70ce3b0bfb5d'

# Author of this module
Author = 'Don Jones & Jeffery Hicks'

# Company or vendor of this module
CompanyName = 'Unknown'
```

```
# Copyright statement for this module
Copyright = '(c) 2017-2020 Don Jones & Jeffery Hicks. All rights reserved.'

# Description of the functionality provided by this module
Description = "Sample for for 'The PowerShell Scripting and Toolmaking Book' by Don \
Jones and Jeffery Hicks published on Leanpub.com."

# Minimum version of the Windows PowerShell engine required by this module
# PowerShellVersion = ''

# Name of the Windows PowerShell host required by this module
# PowerShellHostName = ''

# Minimum version of the Windows PowerShell host required by this module
# PowerShellHostVersion = ''

# Minimum version of Microsoft .NET Framework required by this module. This prerequi\
site is valid for the PowerShell Desktop edition only.
# DotNetFrameworkVersion = ''

# Minimum version of the common language runtime (CLR) required by this module. This\
 prerequisite is valid for the PowerShell Desktop edition only.
# CLRVersion = ''

# Processor architecture (None, X86, Amd64) required by this module
# ProcessorArchitecture = ''

# Modules that must be imported into the global environment prior to importing this \
module
# RequiredModules = @()

# Assemblies that must be loaded prior to importing this module
# RequiredAssemblies = @()

# Script files (.ps1) that are run in the caller's environment prior to importing th\
is module.
# ScriptsToProcess = @()

# Type files (.ps1xml) to be loaded when importing this module
# TypesToProcess = @()

# Format files (.ps1xml) to be loaded when importing this module
# FormatsToProcess = @()
```

```
# Modules to import as nested modules of the module specified in RootModule/ModuleTo\
Process
# NestedModules = @()

# Functions to export from this module, for best performance, do not use wildcards a\
nd do not delete the entry, use an empty array if there are no functions to export.
FunctionsToExport = '*'

# Cmdlets to export from this module, for best performance, do not use wildcards and\
 do not delete the entry, use an empty array if there are no cmdlets to export.
CmdletsToExport = '*'

# Variables to export from this module
VariablesToExport = '*'

# Aliases to export from this module, for best performance, do not use wildcards and\
 do not delete the entry, use an empty array if there are no aliases to export.
AliasesToExport = '*'

# DSC resources to export from this module
# DscResourcesToExport = @()

# List of all modules packaged with this module
# ModuleList = @()

# List of all files packaged with this module
# FileList = @()

# Private data to pass to the module specified in RootModule/ModuleToProcess. This m\
ay also contain a PSData hashtable with additional module metadata used by PowerShel\
l.
PrivateData = @{

    PSData = @{

        # Tags applied to this module. These help with module discovery in online ga\
lleries.
        # Tags = @()

        # A URL to the license for this module.
        # LicenseUri = ''
```

```
        # A URL to the main website for this project.
        # ProjectUri = ''

        # A URL to an icon representing this module.
        # IconUri = ''

        # ReleaseNotes of this module
        # ReleaseNotes = ''

    } # End of PSData hashtable

} # End of PrivateData hashtable

# HelpInfo URI of this module
# HelpInfoURI = ''

# Default prefix for commands exported from this module. Override the default prefix\
 using Import-Module -Prefix.
# DefaultCommandPrefix = ''

}
```

A lot of this is commented out, which is the default when you use `New-ModuleManifest`. The specifics you *must* provide will differ based on your repository's requirements, but in general we recommend at least the following be completed:

- RootModule. This is actually mandatory for the .psd1 to work, and it should point to the "main" .psm1 file of your module.
- ModuleVersion. This is generally mandatory, too, and is at the very least a very good idea.
- GUID. This is mandatory, and generated automatically by `New-ModuleManifest`.
- Author.
- Description.

Take note of your author name and try to be consistent. You want to make it easy for people to find the other amazing tools you have published.

These are, incidentally, the minimums for publishing to PowerShell Gallery. we also recommend, in the strongest possible terms, that you specify the FunctionsToExport array, as well as VariablesToExport, CmdletsToExport, and AliasesToExport if those are applicable. Ours, as you'll see above, are set to `*`, which is a bad idea. In our specific example here, it makes sense, because our root module is actually empty - we aren't exporting anything; the module is just a kind of container for our sample

code to live in. But in your case, the recommended best practice is to explicitly list function, alias and variable (without the $ sign) names which will achieve two benefits:

- Auto-discovery of your commands will be faster, since PowerShell can just read the .psd1 rather than parsing the entire .psm1.
- Some repositories may be able to provide per-command search capabilities if you specify which commands your module offers.

Publishing to PowerShell Gallery

PowerShellGallery.com is a Microsoft-owned, public NuGet repository for released code. It can host PowerShell modules, DSC resources, and other artifacts. Start by heading over to PowerShellGallery.com and logging in or registering, using your Microsoft ID. Once signed in, click on your name. As part of your Gallery profile, you'll be able to request, view, and see your API key. This is a long hexadecimal identifier that you'll need when publishing code. Keep this secure.

With your API key in hand, it's literally as easy as going into PowerShell and running `Publish-Module` (which is part of the PowerShellGet module, which ships with PowerShell v5 and later and can be downloaded from PowerShellGallery.com for other PowerShell versions). Provide the name of your module, and your API key (via the `-NuGetApiKey` parameter), and you're good to go.

```
Publish-Module -path c:\scripts\MyAwesomeModule -nugetapikey $mykey
```

You may be prompted for additional information if it can't be found in your module manifest.

Be aware that publishing a module will include all files and folders in your module location. Hidden files and folders should be ignored but make sure you have cleaned up any scratch, test or working files.

You'll likely receive a confirmation email from the Gallery, which may include a number of PSScriptAnalyzer notifications. As we describe in the chapter on Analyzing Your Script, the Gallery automatically runs a number of PSScriptAnalyzer best practices rules on all submitted code, and you should try hard to confirm with these unless you've a specific reason not to.

So what's appropriate for PowerShell Gallery publication?

- Production-ready code. Don't submit untested, pre-release code unless you're doing so as part of a public beta test, and be sure to clearly indicate that the code isn't production-ready (for example, using `Write-Warning` to display a message when the module is loaded).
- Open-source code. Gallery code is, by implication, open-source; you should consider hosting your development code in a public OSS repository like GitHub, and only publish "released" code to the Gallery. Be sure not to include any proprietary information.

- Useful code. There are like thirty seven million 7Zip modules in the Gallery. More are likely not needed. Try to publish code that provides unique value.

Items *can* be removed from Gallery if you change your mind, but Microsoft doesn't have the ability to go out and delete whatever people may have already downloaded. Bear that in mind before contributing.

Publishing to Private Repositories or Galleries

Microsoft's vision is that organizations will host private and internal repositories. You may want to use a private repository merely for testing purposes. Ideally these internal repositories will be based on Nuget. Setting up one of these is outside the scope of this book. However you can setup a repository with a simple file share.

We've created a local file share and made sure that the admins group has write access.

```
New-SMBShare -name MyRepo -path c:\MyRepo -FullAccess Administrators `
-ReadAccess Everyone
```

Don't put any other files in this folder other than what you publish otherwise you will get errors when using `Find-Module`.

Next, you can register this file share as a repository.

```
Register-PSRepository -name MyRepo -SourceLocation c:\MyRepo `
-InstallationPolicy Trusted
```

We set the repository to be trusted because we know what is going in and we don't want to be bothered later when we try to install from it. If you forget, you can modify the repository later:

```
Set-PSRepository -Name MyRepo -InstallationPolicy Trusted
```

Now you can publish locally:

```
Publish-Module -Path c:\scripts\onering -Repository MyRepo
```

This local repository can be used just like the PowerShell gallery.

```
PS C:\> find-module -Repository MyRepo

Version    Name           Type       Repository      Description
-------    ----           ----       ----------      -----------
0.0.1.0    onering        Module     MyRepo          The module that ...
```

You can even install locally to verify everything works as expected.

```
PS C:\> install-module onering -Repository MyRepo
PS C:\> get-command -module OneRing -ListImported

CommandType     Name                                Version    Source
-----------     ----                                -------    ------
Function        Disable-Ring                        0.0.1.0    OneRing
Function        Enable-Ring                         0.0.1.0    OneRing
Function        Get-Ring                            0.0.1.0    OneRing
Function        Remove-Ring                         0.0.1.0    OneRing
Function        Set-Ring                            0.0.1.0    OneRing
```

We set this up locally as a proof of concept. It shouldn't take that much more work to setup a repository on a company file share. Just mind your permissions.

Your Turn

We aren't going to offer a real hands-on lab in this chapter, mainly because we think it's a bad idea to use a public repo like PowerShell Gallery as a "lab environment!" It's also non-trivial to set up your own private repository, and if you go through that trouble, we think you'll want it to be in *production*, not in a lab, so that you can benefit from that work.

That said, we do want to encourage you to sign into the PowerShell Gallery and create your API key, as we've described doing in this chapter. It's a first step toward getting ready to publish your own code.

Let's Review

We aren't going to ask you publish anything to the gallery. You may never have a need to publish or share your work. But let's see if you picked up anything in this chapter.

1. The Microsoft PowerShell Gallery is based on what technology?
2. What important file is required to publish to the gallery that contains critical module metadata?
3. What should you publish to any repository?

Review Answers

Hopefully you came up with answers like this:

1. Nuget
2. A module manifest.
3. Any unique project that offers value and is production ready. You can publish your project that might be in beta or under development but that should be made clear to any potential consumer such as through version numbering.

Part 3: Controller Scripts and Delegated Administration

With your tools constructed and tested, it's time to put them to work - and that means writing controller scripts. Not sure what that means? Keep reading. We'll look at several kinds, from simple to complex, and look at a couple of other, unique ways in which you can put your tools to work.

Basic Controllers: Automation Scripts and Menus

We've written a lot about tools and toolmaking in the first two parts of this book; this part is more about the *controllers* that use those tools. Just like a human hand controls a hammer to some useful purpose, like building a house, a *controller script* takes your tools and puts them to some useful purpose. In this chapter, we'll start with two very basic kinds of controller scripts.

Building a Menu

One of the simplest things you can do to expose tools to less-technical colleagues is through a simple, text-based menu - a scenario where `Read-Host` and `Write-Host` are totally important. Well-designed tools will prompt for mandatory parameters automatically, making menu-driven tool use even easier. You can write your own prompts for any non-mandatory parameters, if desired.

We're going to assume your menu-driven script is going to invoke your own commands and functions. For our demonstrations we're going to use out-of-the-box cmdlets. In the download files for this chapter you'll find a copy of `basicmenu.ps1`.

<<BasicMenu.ps1[40]

This script isn't perfect but it demonstrates some basic concepts. It uses a here string to define a set of menu choices. This menu is display as the prompt for `Read-Host`. We've also prompted for a computer name. Once the user has entered a value you need to take some action based on the value. For that we use a simple Switch construct.

```
PS C:\> c:\scripts\basicmenu.ps1

        MyMenu
--------------------------
1. Get services
2. Get processes
3. Get System event logs
4. Check free disk space (MB)
5. Quit

Select a menu choice: 4
```

[40] code/PowerShell-Toolmaking/Chapters/basic-controller-scripts-and-menus/basicmenu.ps1

```
Enter a computername or press Enter to use the localhost:
27005.375
PS C:\>
```

As we said there are some issues. Primarily you may want to repeat the menu until the user is finished. The first version also didn't do a good job of validating choices which is very important. In `basicmenu-improved.ps1` you'll find this function.

<<BasicMenu-Improved.ps1[41]

This version adds the necessary validation and clears the screen each time making it visually more appealing. The key difference though is that after executing code in the Switch construct, the function calls itself again.

Of course since the menu is just a display on the screen, you can dress up with `Write-Host`. `FancyMenu.ps1` has a more elaborate version.

<<FancyMenu.ps1[42]

This version will center the menu title and display it in Cyan. The menu itself is displayed in yellow. We'll let you load the function into your PowerShell session and see for yourself.

You could also dress up the menu using ANSI escape sequences or adding borders. Take a look at Jeff's PSScriptTools[43] module which you can download from the PowerShell Gallery.

Using UIChoice

For longish menus, the approach we just showed works best. But there is another option for a selection menu of sorts. You've probably seen it whenever you get a confirmation prompt. You can create a similar menu command.

For each choice you need to create an object like this:

```
$a = [System.Management.Automation.Host.ChoiceDescription]::new("Running &Services")
```

The parameter value is the text that will be displayed. Put an `&` in front of the character you want the user to type to select that choice.

Optionally, you can create a help message:

[41] code/PowerShell-Toolmaking/Chapters/basic-controller-scripts-and-menus/basicmenu-improved.ps1
[42] code/PowerShell-Toolmaking/Chapters/basic-controller-scripts-and-menus/FancyMenu.ps1
[43] https://github.com/jdhitsolutions/PSScriptTools

```
$a.HelpMessage = "Get Running Services"
```

Now for the cool part. Eventually, the user will make a choice which will select the a choice object. We're going to add a new member to the object and define a ScriptMethod.

```
$a | Add-Member -MemberType ScriptMethod -Name Invoke -Value {
    Get-Service | where {$_.status -eq "running"}
} -force
```

When this object is selected we can run the Invoke() method and execute the scriptblock. You'll repeat this process for all choices, adding each one to an array.

To run, use the `PromptForChoice()` method specifying a title, message, the array variable and the default choice which is the corresponding index number from the array.

```
$r = $host.ui.PromptForChoice("TITLE HERE","MESSAGE:",$collection,0)
```

In the code samples dot source `choicemenu.ps1` to load the `Invoke-Choice` function.

<<ChoiceMenu.ps1[44]

Here's a taste of what it looks like in the console.

```
PS C:\> invoke-choice

Help Desk Menu
Select a task:
[S] Running Services  [P] Top Processes  [D] Disk Status  [Q] Quit  [?] Help (default is "Q"): ?
S - Get Running Services
P - Get top processes sorted by workingset
D - Get fixed disk information
Q - Quit and exit
[S] Running Services  [P] Top Processes  [D] Disk Status  [Q] Quit  [?] Help (default is "Q"): d
Enter a computername or press Enter to use the localhost:

DeviceID DriveType ProviderName VolumeName Size         FreeSpace    PSComputerName
-------- --------- ------------ ---------- ----         ---------    --------------
C:       3                      Windows    254721126400 78698328064  BOVINE320
D:       3                      Data       511974567936 178824142848 BOVINE320



Help Desk Menu
Select a task:
[S] Running Services  [P] Top Processes  [D] Disk Status  [Q] Quit  [?] Help (default is "Q"): q
Have a nice day.
PS C:\> _
```

Choice menu

[44] code/PowerShell-Toolmaking/Chapters/basic-controller-scripts-and-menus/ChoiceMenu.ps1

If you run this in the PowerShell ISE you'll get a graphical popup for the menu.

Writing a Process Controller

The tools that you have been building are essentially building blocks that you can assemble with other commands in order to achieve some end result. A command like `Get-Service` is useful we suppose on its own. But its real value comes from how you might integrate it into a larger process. The same should be true of your commands. Often you may have a process built around them. If you've been smart about it, you've probably figured out how to execute those commands in a number of pipelined expressions at a PowerShell prompt. But let's go a step further.

Again, we're going to use some common cmdlets to demonstrate concepts. You of course would be using your own tools. Let's say that each Monday morning you need to go through a list of servers and find all errors and warning from the System event log that occurred in the last 48 hours and that you need to email a report to the server management team.

You might open up PowerShell every Monday morning and run a command like this:

```
$body = Get-Content s:\servers.txt |
Where-Object { Test-Wsman $_ -erroraction silentlycontinue }  |
Foreach-Object {
 Get-Eventlog -LogName System -EntryType Error,Warning `
 -After (Get-Date).AddHours(-48) -ComputerName $_ }  |
Select-Object Machinename,EntryType,TimeGenerated,Source,Message |
Out-string

Send-MailMessage -to team@company.com -Subject "Weekend Error Report" `
 -Body $body
```

Yeah, it might get the job done, but do you really want to type that every week? And what about when you are on vacation or out sick? How flexible is this? What you need is a controller script that orchestrates the commands. Here is a possible solution, which you'll also find in the chapter code downloads as `ProcessController.ps1`.

<<ProcessController.ps1[45]

Can you see some advantages in a controller script? We've parameterized a lot of it and set some defaults. These are settings we would expect to use most of the time. Now every Monday, someone on the team can run this command:

[45] code/PowerShell-Toolmaking/Chapters/basic-controller-scripts-and-menus/processcontroller.ps1

```
s:\eventlogreport.ps1 -computername (Get-Content s:\servers.txt) `
-after (Get-Date).AddHours(36) -sendto admins@company.com
```

But the controller script allows the flexibility to specify different a different set of computers and event logs.

```
s:\eventlogreport.ps1 -computer $web -logname application -entrytype error `
-after (Get-Date).AddHours(-12)
```

Even better - once you have a controller script you could setup a PowerShell scheduled job and never have to worry about it again!

Your Turn

Let's see how much you picked up in this chapter. We're going to have you create a controller/menu type script. We've given you plenty of examples to take as a starting point. The sample scripts are in the code downloads so feel free to copy and paste.

Start Here

We'd like to see you build something that the help desk could run to provide system information using `Get-CimInstance`. You can assume they already have the necessary credentials to remotely query a machine. You will also need to prompt the user for a computername.

Your Task

Create a menu with these items:

- LogicalDisks
- Services
- Operating system
- Computer system
- Processes

Prompt the user to select one and then run corresponding `Get-Ciminstance` command to display the results. You should include some way for the user to specify a computername. Ideally, the menu should re-display until the user decides to quit

Our Take

If you wrote something that displayed a menu and executed a corresponding command, you succeeded. Our solution is probably "over-engineered" but we wanted to demonstrate as many techniques as possible.

<<CimMenuSolution.ps1[46]

Our solution displays a menu using `Write-Host` and then uses the choice prompt technique. After the command is executed, the script is re-run until the user opts to quit.

```
 System Information Menu

    1 LogicalDisks
    2 Services
    3 Operating system
    4 Computer system
    5 Processes
    6 Quit


Make a selection:
[1] 1 Disks  [2] 2 Services  [3] 3 OS  [4] 4 Computer  [5] 5 Processes  [6] 6 Quit  [?] Help (default is "6"): 3

Enter a computername (leave blank for localhost):

PSComputerName Caption                 Version    InstallDate
-------------- -------                 -------    -----------
BOVINE320      Microsoft Windows 10 Pro 10.0.19041 5/30/2020 2:55:28 PM


Press Enter to continue...:
```

System Information Menu

Let's Review

We really don't have much in the way of review questions for this chapter. The primary take away is that you may want to wrap your tools in some sort of controller or "menu-ing" function. Again, think about who will be using your toolset and how they might interact with it. If you go down the path we've demonstrated in this chapter be sure to include plenty of documentation or training.

[46] code/PowerShell-Toolmaking/Chapters/basic-controller-scripts-and-menus/CimMenuSolution.ps1

Graphical Controllers in WPF

For many IT Pros, PowerShell means running scripts and commands from a command line. While we appreciate the elegance of the pipeline, that doesn't mean the GUI is dead. The PowerShell model has always been to first "do it" at the prompt. If you need a GUI, then build one on top of your command line version. Microsoft has followed this model for years.

You may want to do something similar. Perhaps you want to provide a graphical interface for a user to enter some values that can be passed to an underlying PowerShell command. Or maybe you want to build a complete, stand-alone GUI for a less-PowerShell savvy user. As with the rest of PowerShell toolmaking, you have to determine who will be using your graphical tool and their expectations.

Design First

We named this chapter "graphical controllers" for a reason. We don't regard a GUI as a *tool* in the sense that we've used the word in this book. A GUI doesn't *do* things; it provides a means of *accessing tools*. That means your functional code - the code *doing* something for you - should be written as functions. Your GUI should provide an *interface* to those tools. The code in your GUI should be the bare minimum necessary to collect input, run tools using that input, and display output. The less code in your GUI, the better a pattern you'll have. You'll have an easier time testing and troubleshooting, too.

WinForms or WPF

You've probably heard a lot of talk about WinForms (Windows Forms) in the PowerShell community. We're sure that if you searched, you would find a lot of valuable examples and tips. WinForms are also based on .NET classes which, make them easy to use in PowerShell. WinForms have been around for a long time. So why would you choose WPF (Windows Presentation Foundation)?

There are two primary reasons.

First, WinForms doesn't scale. And by scaling, we mean video resolution. Today, it is not uncommon to have very high-resolution monitors, now pushing 4K and beyond. If you have one of these displays, you may have run an older application that didn't display well. The font was probably way too small to read without resorting to some sort of display jujitsu–more than likely, that application was written with WinForms.

WPF, on the other hand, is designed to display the same way regardless of screen resolution. You don't have to worry about how your form will be displayed. Plus, we think WPF has a more "modern" look-and-feel.

The second reason is that with WPF, it is much easier to separate your display code from any logic (meaning, tools) behind it. It isn't required, but you can have the code that creates your form in a separate file from your code that implements it. We'll get to that in a bit.

Again, regardless of which approach you take, there is an important design pattern we want to stress. **WPF itself is not the tool**. The tool is the underlying PowerShell command that you have created. WPF is merely a graphical layer. This has always been the model going all the way back to the days of Exchange 2007. The GUI sits on top of the PowerShell commands. Now, creating the GUI, in this case, WPF will take its own chunk of PowerShell coding. We will spend the first part of this chapter explaining those nuts and bolts, and then we'll pull everything together.

Coding with WPF is a huge topic and typically very developer-oriented. We're going to give you enough information packaged for IT Pros who are just getting started with this stuff. We are intentionally glossing over a number of features that more experienced WPF developers would use. We wanted to keep this chapter as manageable as possible for you.

WPF Architecture

At its simplest, WPF is based on a concept of layers with nested objects. At the top is a *window*. Within the window, you will typically add either a *stack panel* or a *grid*. At this level, insert all of the graphical elements such as text boxes and buttons to the stack panel or grid and then add that to the window. Once everything is assembled, you can display it.

There are a few potential "gotchas". First, your PowerShell session must be running in single-threaded apartment (STA) mode. That is the default, but if you started PowerShell in multi-threaded (MTA) mode, you'd most likely see errors when you start to create WPF elements.

The second thing to watch for is the need to load any required .NET libraries. If you are using the PowerShell ISE to develop and run your WPF-based scripts, you might find that everything works fine. But in the PowerShell console, you might get errors telling you that you need to load an assembly or two. If so, insert these lines at the beginning of your script.

```
Add-Type -AssemblyName PresentationFramework
Add-Type -AssemblyName PresentationCore
Add-Type -AssemblyName WindowsBase
```

That should cover just about everything, and it won't hurt to include them.

PowerShell Core (6.x) does not support WPF. Nor does PowerShell 7.x on **non-Windows** platforms. And while you *could* use WPF **on** a Server Core installation, you shouldn't. Create your graphical tool to run from a client desktop that manages remote servers.

Using .NET

Let's start with the most basic of WPF scripts, and we're going to do it all with .NET code. One of the added benefits of this approach is that you can pipe objects as you create them to `Get-Member` to learn more about their properties.

You can find all of our examples in the chapter's code downloads. Remember, all we're doing right now is demonstrating the mechanics of WPF.

First, we need to create a top-level form or window.

```
$form = New-Object System.Windows.Window
```

We should give the form a title and specify how large we want it to be.

```
$form.Title = "Hello WPF"
$form.Height = 200
$form.Width = 300
```

Nothing too complicated. Next, let's add a button.

```
$btn = New-Object System.Windows.Controls.Button
```

What text should we put on it?

```
$btn.Content = "_OK"
```

The underscore in front of the O is an accelerator and completely optional. When displayed, you could hit `Alt+O` instead of clicking the button. We should also define how large the button should be and how to position it.

```
$btn.Width = 65
$btn.HorizontalAlignment = "Center"
$btn.VerticalAlignment = "Center"
```

Where did we get these values? Search for the class name, like `System.Windows.Controls.Button` and you'll see links to the MSDN documentation. That's a good place to get started. If you get stuck building your WPF tool, head to the forums at PowerShell.org to get a nudge in the right direction. But expect to go through a lot of trial and error when using .NET code like this.

We have a button, but it won't do anything unless we program it. Windows is an event-driven operating system. We click, drag, and drop all the time, and when these events happen (fire), Windows responds accordingly. We need to provide instruction about what to do if the button is clicked. This is accomplished through an *event handler*.

The handler is essentially a PowerShell scriptblock. This scriptblock can be as simple or as complex as you need it to be, and you can reference other form elements (we'll give you an example later in the chapter). All you need to do is add something to the _Click event.

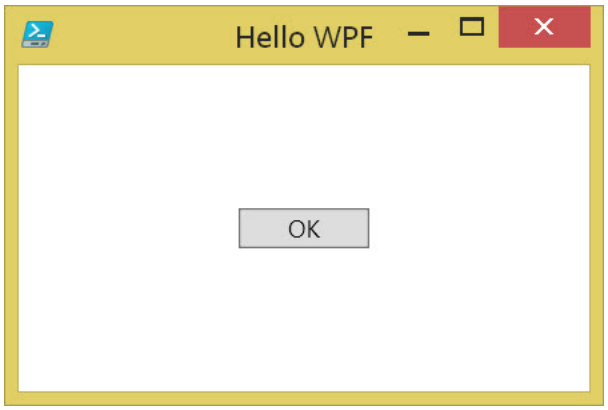
```
$btn.Add_click({
  $msg = "Hello, World and $env:username!"
  Write-Host $msg -ForegroundColor Green
})
```

Once the button is finished, we can add it to the parent container, in this case, the window itself.

```
$form.AddChild($btn)
```

To display the form, invoke the `ShowDialog()` method.

There is a `Show()` method, but if you use it, you'll have to close the PowerShell session to get rid of the form.



hello world

Notice that while the form is displayed, the script is still running, which means you don't get your prompt back. However, if you click OK, you should get a message written in green to the host. Click the X to close the form and get your prompt back.

In our sample form, you'll also see an alternate command in the click handler. Uncomment the `Write-Output` command so you end up with this:

```
Write-Output $msg
```

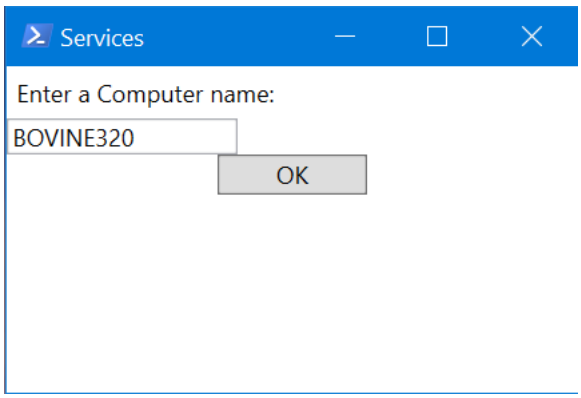
You should comment out the `Write-Host` statement. Re-run the demo and click OK. What happened? Nothing. While a WPF script is running, you are blocked from the rest of the pipeline. We'll show you some ways around this, but this is an important piece of information.

Let's look at another example that is a bit more practical, plus we can demo the stack panel element. This is `Stack-Services.ps1`.

```
#WPF Demonstration using a stack panel

#add the assembly
Add-Type -AssemblyName PresentationFramework

#create the form
$form = New-Object System.Windows.Window

#define what it looks like
$form.Title = "Services"
$form.Height = 200
$form.Width = 300

#create the stack panel
$stack = New-Object System.Windows.Controls.StackPanel

#create a label and assign properties
$label = New-Object System.Windows.Controls.Label
$label.HorizontalAlignment = "Left"
$label.Content = "Enter a Computer name:"

#add to the stack
$stack.AddChild($label)

#create a text box and assign properties
$TextBox = New-Object System.Windows.Controls.TextBox
$TextBox.Width = 115
$TextBox.HorizontalAlignment = "Left"

#set a default value
$TextBox.Text = $env:COMPUTERNAME

#add to the stack
$stack.AddChild($TextBox)
```

```
#create a button and assign properties
$btn = New-Object System.Windows.Controls.Button
$btn.Content = "_OK"
$btn.Width = 75
$btn.VerticalAlignment = "Bottom"
$btn.HorizontalAlignment = "Center"

#this will sort of work
$OK = {
  Write-Host "Getting services from $($textbox.Text)" -ForegroundColor Green
  Get-Service -ComputerName $textbox.Text | Where-Object status -eq 'running'
}

#add an event handler
$btn.Add_click($OK)

#add to the stack
$stack.AddChild($btn)

#add the stack to the form
$form.AddChild($stack)

#show the form
[void]($form.ShowDialog())
```

The script comments should explain what we're doing. Notice in the $OK scriptblock how we're referencing the computername from the $textbox variable. Go ahead and run the script.



get service with WPF

In a stack panel, all of the child objects are "stacked" like building blocks. Not necessarily elegant, but for a simple form, it is easy to pull together. What happens when you click OK? We want to

display all the running services for the specified computer. The `Write-Host` command runs but not `Get-Service`. Again, this is because of blocking. Why not use WPF to also display the results.

Our WPF examples using `Get-Service` won't work in PowerShell 7 because the `-ComputerName` parameter has been removed.

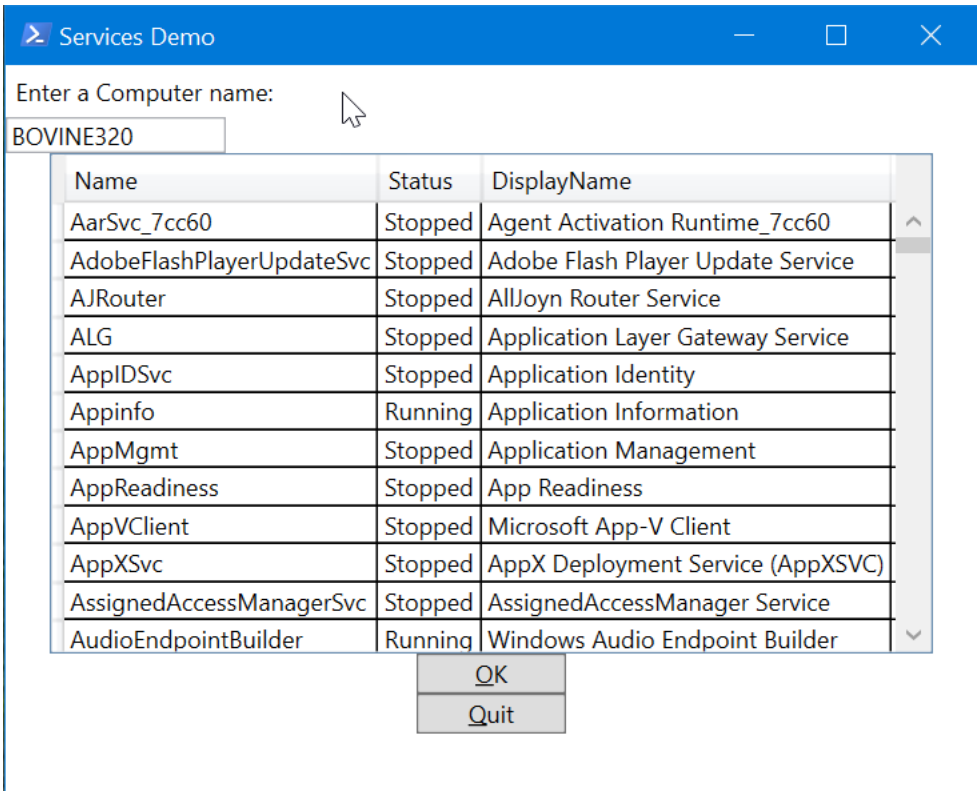
Here's a revised version called `Display-Services.ps1` which uses a new control, a *datagrid*, to display the results.

```
# Display-Services.ps1

Add-Type -AssemblyName PresentationFramework

$form = New-Object System.Windows.Window

#define what it looks like
$form.Title = "Services Demo"
$form.Height = 400
$form.Width = 500

$stack = New-Object System.Windows.Controls.StackPanel

#create a label
$label = New-Object System.Windows.Controls.Label
$label.HorizontalAlignment = "Left"
$label.Content = "Enter a Computer name:"

#add to the stack
$stack.AddChild($label)

#create a text box
$TextBox = New-Object System.Windows.Controls.TextBox
$TextBox.Width = 110
$TextBox.HorizontalAlignment = "Left"
$TextBox.Text = $env:COMPUTERNAME

#add to the stack
$stack.AddChild($TextBox)

#create a datagrid
$datagrid = New-Object System.Windows.Controls.DataGrid
$datagrid.HorizontalAlignment = "Center"
```

```
$datagrid.VerticalAlignment = "Bottom"
$datagrid.Height = 250
$datagrid.Width = 441

$datagrid.CanUserResizeColumns = "True"
$stack.AddChild($datagrid)

#create a button
$btn = New-Object System.Windows.Controls.Button
$btn.Content = "_OK"
$btn.Width = 75
$btn.HorizontalAlignment = "Center"

#this will now work
$OK = {
    Write-Host "Getting services from $($textbox.Text)" -ForegroundColor Green
    $data = Get-Service -ComputerName $textbox.Text |
    Select-Object -Property Name, Status, Displayname
    $datagrid.ItemsSource = $data
}

#add an event handler
$btn.Add_click($OK)

#add to the stack
$stack.AddChild($btn)

#add the stack to the form
$form.AddChild($stack)

#run the OK scriptblock when form is loaded
$form.Add_Loaded($OK)

$btnQuit = New-Object System.Windows.Controls.Button
$btnQuit.Content = "_Quit"
$btnQuit.Width = 75
$btnQuit.HorizontalAlignment = "center"

#add the quit button to the stack
$stack.AddChild($btnQuit)

#close the form
$btnQuit.add_click({$form.Close()})
```

```
#show the form and suppress the boolean output
[void]($form.ShowDialog())
```

In the OK scriptblock, we can now run `Get-Service` and select the properties we want to display. The data can be used the ItemsSource for the datagrid object. We've also added a handler for when the form is loaded. We decided that when the form is loaded use the local host to go ahead and display the service information.



Services form

If you have another computer to test, enter the name and click the OK button or use the Alt+O shortcut. When finished, use the Quit button we added.

Using XAML

We showed you the .NET pieces so that you would understand the objects behind WPF. For simple projects, using native classes isn't too bad. For more complicated layouts, you'll end up with so much trial and error that you'll put a permanent head-shaped dent into your desk. You'll also recall at the beginning of the chapter, we mentioned the concept of separating the presentation from the logic. That's where XAML comes into play.

If you were a developer creating a WPF application, you would end up with some specialized XML called XAML that describes the graphical layout.

XAML Sample

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Disk Report" Height="355" Width="535" Background="#FFBDB3B3">
<Grid>
    <Button x:Name="btnRun" Content="_Run" HorizontalAlignment="Left"
    Height="20" Margin="343,291,0,0" VerticalAlignment="Top" Width="74"/>
    <Button x:Name="btnQuit" Content="_Quit" HorizontalAlignment="Left"
    Margin="433,291,0,0" VerticalAlignment="Top" Width="75"
    RenderTransformOrigin="0.365,-0.38"/>
    <ComboBox x:Name="comboNames" HorizontalAlignment="Left" Height="20"
    Margin="11,25,0,0" VerticalAlignment="Top" Width="166"/>
    <Label x:Name="label" Content="Select a computer"
    HorizontalAlignment="Left" Height="27" Margin="9,3,0,0"
    VerticalAlignment="Top" Width="206"/>
    <DataGrid x:Name="dataGrid" HorizontalAlignment="Left" Height="229"
    Margin="10,55,0,0" VerticalAlignment="Top" Width="498"/>
</Grid>
</Window>
```

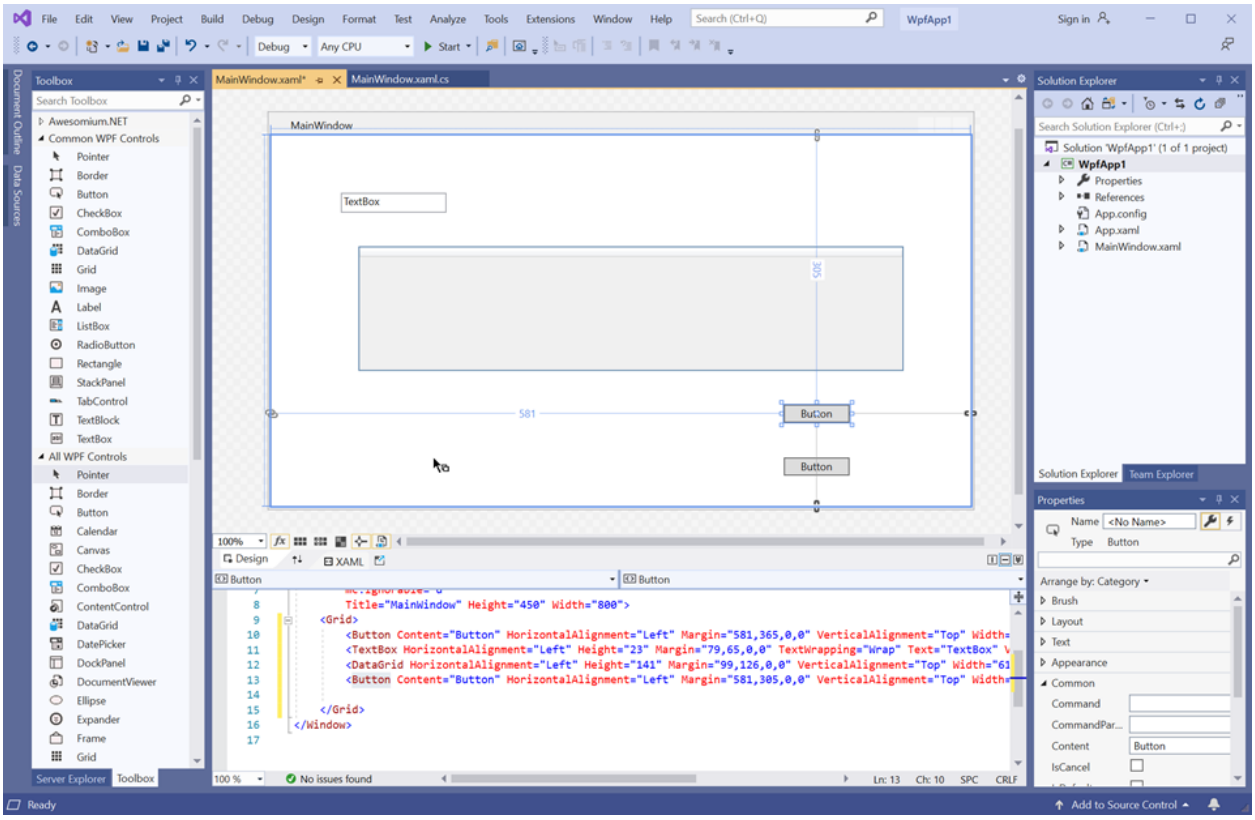
Now, before you begin skipping to the next chapter, let us explain something. While you *could* write this off the top of your head, and there are free XAML editors that can help, you don't need to. What you are really looking for is a graphical editor where you can drag and drop the graphical elements and in turn, generate the XAML.

The tool you are looking for is Visual Studio Community Edition, and it is a free download. Visit https://visualstudio.microsoft.com/vs/community/[47] to get the most current version.

When you install Visual Studio Community Edition, it will want to include a ton of stuff. Unless you intend to develop other applications, you don't need to include anything extra. The last time we installed the application, we selected only the .NET desktop development workload. Be aware that even this may require a multi-gig install. You might be able to trim this down by deselecting optional packages.

Once installed, open the application and select `File - New Project`, then select `WPF Application`. Visual Studio will create a new project, although you won't be using it.

[47] https://visualstudio.microsoft.com/vs/community/



Visual Studio Community Edition

On the left side is your tool palette. Grab a grid control and drag and drop it on the main form. Next, do the same for a button control. As each item is selected, the XAML is updated. You can set control properties such as the name directly in the XAML or in the Properties panel on the right side. It will take a bit of time to learn where everything is.

Visual Studio will automatically name controls like Button and Button1. You should rename them to reflect what they will eventually do. We like to use some type of prefix like “btn” which would lead us to rename them “btnRun” and “btnQuit”. Eventually you will need to “find” these controls so proper naming is important.

Continue dragging and dropping controls as necessary to get the look and feel you need. You aren’t putting any logic or commands to this project. All you need is the XAML that Visual Studio is generating. When you are finished, save your project. You can now either copy the XAML from Visual Studio and paste it into a new file or under File there is a menu choice to save the main window XAML.

The Visual Studio XAML includes references that you won’t need in PowerShell. In the XAML file you will find something like this:

```
<Window x:Class="WpfDiskReport.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:WpfDiskReport"
```

You can edit it down to:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

When you start using the XAML in PowerShell, you'll get an error message about some namespace if you missed something. Delete the corresponding line in the XAML and try again.

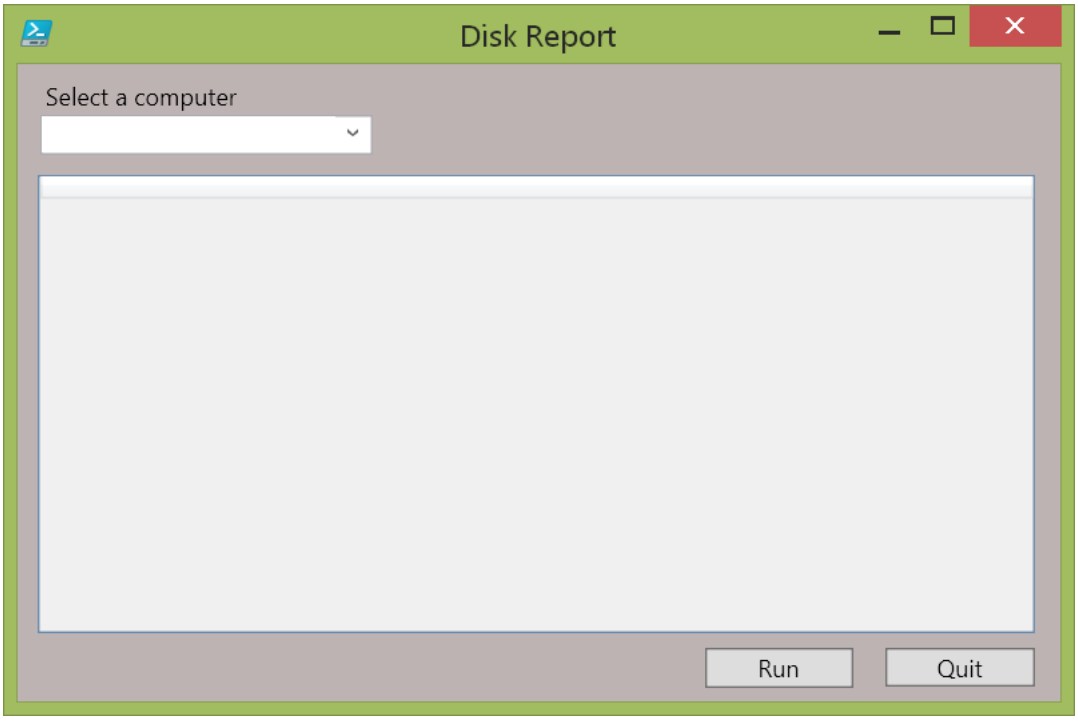
How do we use it?

A Complete Example

First, we're starting with a PowerShell command that we already know works from the console. It could be a script you've developed or a function that is part of your module. We have a sample script called `DiskStats.ps1`.

```
$cimParams = @{
    Computername = "localhost", $env:computername
    classname    = "win32_logicaldisk"
    filter       = "drivetype=3"
}
Get-CimInstance @cimParams |
Select-Object -Property @{Name = "Computername";Expression = {$_.SystemName}},
DeviceID, @{Name = "SizeGB";Expression = {$_.Size/1GB -as [int]}},
@{Name = "FreeGB";Expression = { [math]::Round($_.Freespace/1GB, 2)}},
@{Name = "PctFree";Expression = { ($_.freespace/$_.size)*100 -as [int]}}
```

We designed the form so that the user could select a computer name from a drop-down box, get the disk usage data and display it directly in the form.



Disk Report form

Now that you know the goal let's get there.

First, we need to bring in the XAML content into an XML document. If the XAML is in an external file we can use a line like this:

```
[xml]$xaml = Get-Content $psscriptroot\diskstat.xaml
```

Or you can include it directly into the PowerShell script file.

DiskStats XAML

```
[xml]$xaml = @"
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Disk Report" Height="355" Width="535" Background="#FFBDB3B3">
    <Grid>
        <Button x:Name="btnRun" Content="_Run" HorizontalAlignment="Left"
        Height="20" Margin="343,291,0,0" VerticalAlignment="Top" Width="74"/>
        <Button x:Name="btnQuit" Content="_Quit" HorizontalAlignment="Left"
        Margin="433,291,0,0" VerticalAlignment="Top" Width="75"
        RenderTransformOrigin="0.365,-0.38"/>
        <ComboBox x:Name="comboNames" HorizontalAlignment="Left" Height="20"
        Margin="11,25,0,0" VerticalAlignment="Top" Width="166"/>
        <Label x:Name="label" Content="Select a computer"
```

```
        HorizontalAlignment="Left" Height="27" Margin="9,3,0,0"
        VerticalAlignment="Top" Width="206"/>
        <DataGrid x:Name="dataGrid" HorizontalAlignment="Left"
        Height="229" Margin="10,55,0,0" VerticalAlignment="Top" Width="498"/>
    </Grid>
</Window>
"@
```

To use this XAML, we need a special object to read it.

```
$reader = New-Object system.xml.xmlnodereader $xaml
$form = [windows.markup.xamlreader]::Load($reader)
```

Remember, all the XAML does is describe what the GUI bits look like. We need to provide the logic, which means we need to assign handlers to the different elements. The next step is to discover those elements in the form and create the objects.

```
$grid = $form.FindName("dataGrid")
$run = $form.FindName("btnRun")
$quit = $form.FindName("btnQuit")
$drop = $form.FindName("comboNames")
```

Now you can see why we stressed the importance of good control names.

When the Run button is clicked, obviously, we want to run our code using the computer name from the combo box. So we'll add a `_Click handler` as we did earlier in this chapter.

```
$run.Add_Click({
$grid.clear()

#or call your external command
$data = @(Get-CimInstance -class win32_logicaldisk -filter "drivetype=3"
-ComputerName $drop.Text |
Select-Object -property @{Name="Computername";Expression={$_.SystemName}},
DeviceID,@{Name="SizeGB";Expression={$_.Size/1GB -as [int]}},
@{Name="FreeGB";Expression = {[math]::Round($_.Freespace/1GB,2)}},
@{Name="PctFree";Expression = {($_.freespace/$_.size)*100 -as [int]}})

$grid.ItemsSource = $data
})
```

The Quit button needs to close the form.

```
$quit.Add_Click({$form.Close()})
```

For the combo box we want to read in a list of computer names and allow the user to enter a separate value. We also want the combo box to have focus when the form is launched.

```
$drop.IsEditable = $True

#hard coded demo names
$names = $env:computername,"localhost"

$names | foreach {
 $drop.Items.Add($_) | Out-Null
}

$drop.focus()
```

And that's it! The only thing that remains is to show the form.

```
$form.ShowDialog() | Out-Null
```

We'll let you play with the sample script to see it in action.

Just the Beginning

As you probably figured out, there is *a lot* to WPF. We've only scratched the surface. There are so many more controls to learn how to use plus things like running WPF in a separate runspace, or using synchronized hashtables. This is a topic we hope we can cover in more detail at some point since it comes up often.

If you'd like to see another example, install Jeff's ISERemoteTab module from the PowerShell gallery and dig through the source code. You'll see the console-oriented function to create a specialized remote tab in the PowerShell ISE plus a separate function that creates a WPF controller for the command that makes it easier to use. Or for something a bit more advanced, look at the source code for ConvertTo-WPFGrid[48] in his PSScriptTools module.

Recommendations

As you have probably gathered by now, creating a WPF based PowerShell tool is not a "quick and dirty" task. Creating a well-designed tool will take some time and experience. With that in mind, here are a few recommendations:

[48] https://github.com/jdhitsolutions/PSScriptTools/blob/master/functions/ConvertTo-WPFGrid.ps1

- Start simple and small. Don't try to create a mammoth WPF-based tool on your first attempt.
- Start with a command that you already know works in the PowerShell console. You'll drive yourself nuts trying to write and troubleshoot a PowerShell command while you are trying to create WPF code.
- Consider who will be using your graphical tool and their expectations.
- How will your tool be maintained? The answer might determine if you keep the XAML in the same file as your script or as an external file. Or you might use .NET classes directly.

Your Turn

Naturally, the best way to learn this is to get your hands dirty so let's see what you've picked up from the chapter. Can you create a graphical PowerShell tool?

Start Here

Back in the chapter on converting functions to classes, you should have created a module with a function to get computer information. Or check in the code downloads for that chapter looking at the TMMachineInfo module.

Your Task

Create a new function called `Show-MachineInfo` that will create a WPF GUI where the user can enter a computer name and see the results in the form. You can keep things simple and display the results in a TextBlock control. We wanted to keep this simple enough that you could use .NET classes or feel free to try out using XAML.

Our Take

We hope you had fun with this exercise. We've included a sample module solution in the chapter's download files. Our approach was to use a stack panel and .NET to keep it simple. Once we got the form code working, we wrapped it into a function.

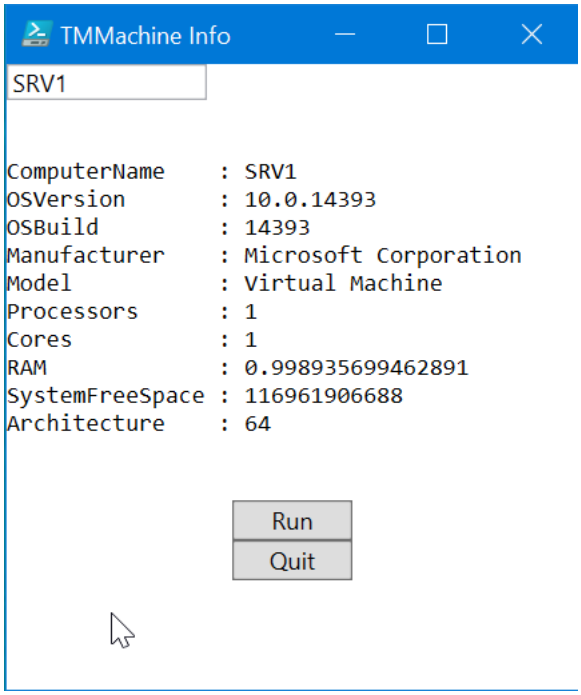
Show-MachineInfo

```
Function Show-MachineInfo {
    [cmdletbinding()]
    [alias("smi")]

    Param(
        [Parameter(Position = 0)]
        [Alias("cn")]
        [ValidateNotNullorEmpty()]
        [string]$Computername = $env:COMPUTERNAME
    )

    $form = New-Object System.Windows.Window
    $form.Title = "TMMachine Info"
    $form.Width = 300
    $form.Height = 350

    $stack = New-Object System.Windows.Controls.StackPanel

    $txtInput = New-Object System.Windows.Controls.TextBox
    $txtInput.Width = 100
    $txtInput.HorizontalAlignment = "left"
    $txtInput.Text = $Computername

    $stack.AddChild($txtInput)

    $txtResults = New-Object System.Windows.Controls.TextBlock
    $txtResults.FontFamily = "Consolas"
    $txtResults.HorizontalAlignment = "left"

    $txtResults.Height = 200

    $stack.AddChild($txtResults)

    $btnRun = New-Object System.Windows.Controls.Button
    $btnRun.Content = "_Run"
    $btnRun.Width = 60
    $btnRun.HorizontalAlignment = "Center"

    $OK = {
        #get machine info from the name in the text box.
        #we're trimming the value in case there are extra spaces
        $data = Get-MachineInfo -Computername ($txtInput.text).trim()
```

```
        #set the value of the txtResults to the data as a string
        $txtResults.text = $data | Out-String
    }

    $btnRun.Add_click($OK)

    $stack.AddChild($btnRun)

    $btnQuit = New-Object System.Windows.Controls.Button
    $btnQuit.Content = "_Quit"
    $btnQuit.Width = 60
    $btnQuit.HorizontalAlignment = "center"

    $btnQuit.Add_click({$form.close()})

    $stack.AddChild($btnQuit)
    $form.AddChild($stack)
    $form.add_Loaded($ok)

    [void]($form.ShowDialog())
}
```

Now the help desk could run the command, which will default to the local computer or they could enter a computer name.

```
PS C:\> show-machineinfo srv1
```



Show-MachineInfo

As long as the form is running, they can enter any other computer name and click Run.

Let's Review

Before we go, let's make sure you've understood some of the key concepts from this chapter.

1. What are some of the benefits of using WPF instead of WinForms?
2. What are some reasons for creating graphical PowerShell tools?
3. What type of file contains the form description?
4. What is the design pattern when it comes to WPF and PowerShell?

Review Answers

We came up with these answers.

1. WPF scales better at higher resolutions and gives your tool a more modern feel.
2. You might want to provide a graphical input form for your command. You might want to display results in a graphical form. Or you may need to have a PowerShell-based tool that does not require the user to type anything at a PowerShell prompt other than perhaps a command to launch the WPF script.
3. XAML
4. WPF itself is not the tool. It is merely a graphical enabler or interface to an underlying PowerShell command.

Proxy Functions

In PowerShell, a *proxy function* is a specific kind of wrapper function. That is, it "wraps" around an existing command, usually with the intent of either:

- Removing functionality
- Hard-coding functionality and removing access to it
- Adding functionality

In some cases, a proxy command is meant to "replace" an existing command. This is done by giving the proxy the same name as the command it wraps; since the proxy gets loaded into the shell last, it's the one that actually gets run when you run the command name.

There's a way, using a fully-qualified command name, to regain access to the wrapped command, so proxy functions shouldn't be seen as security mechanism. They're more of a functional convenience.

For Example

You're probably familiar with PowerShell's `ConvertTo-HTML` command. We'd like to make a version that "replaces" the existing command, providing full access to it but always injecting a particular CSS style sheet, so that the resulting HTML can be a bit prettier.

Creating the Proxy Base

PowerShell actually automates the first step, which is generating a "wrapper" that exactly duplicates whatever command you're wrapping. Here's how to use it (we'll put our results into a Step1 subfolder in this chapter's sample code):

```
$cmd = New-Object System.Management.Automation.CommandMetaData (Get-Command ConvertT\
o-HTML)
[System.Management.Automation.ProxyCommand]::Create($cmd) |
Out-File ConvertToHTMLProxy.ps1
```

Here's the rather-lengthy result (once again, apologies for the backslashes, which represent line-wrapping; it's unavoidable in this instance, but the downloadable sample code won't show them):

```
[CmdletBinding(DefaultParameterSetName='Page',
HelpUri='http://go.microsoft.com/fwlink/?LinkID=113290',
RemotingCapability='None')]
param(
    [Parameter(ValueFromPipeline=$true)]
    [psobject]
    ${InputObject},

    [Parameter(Position=0)]
    [System.Object[]]
    ${Property},

    [Parameter(ParameterSetName='Page', Position=3)]
    [string[]]
    ${Body},

    [Parameter(ParameterSetName='Page', Position=1)]
    [string[]]
    ${Head},

    [Parameter(ParameterSetName='Page', Position=2)]
    [ValidateNotNullOrEmpty()]
    [string]
    ${Title},

    [ValidateNotNullOrEmpty()]
    [ValidateSet('Table','List')]
    [string]
    ${As},

    [Parameter(ParameterSetName='Page')]
    [Alias('cu','uri')]
    [ValidateNotNullOrEmpty()]
    [uri]
    ${CssUri},

    [Parameter(ParameterSetName='Fragment')]
    [ValidateNotNullOrEmpty()]
    [switch]
    ${Fragment},

    [ValidateNotNullOrEmpty()]
    [string[]]
```

```
    ${PostContent},

    [ValidateNotNullOrEmpty()]
    [string[]]
    ${PreContent})

begin
{
    try {
        $outBuffer = $null
        if ($PSBoundParameters.TryGetValue('OutBuffer', [ref]$outBuffer))
        {
            $PSBoundParameters['OutBuffer'] = 1
        }
        $wrappedCmd = $ExecutionContext.InvokeCommand.GetCommand('Microsoft.PowerShe\
ll.Utility\ConvertTo-Html',
        [System.Management.Automation.CommandTypes]::Cmdlet)
        $scriptCmd = {& $wrappedCmd @PSBoundParameters }
        $steppablePipeline = $scriptCmd.GetSteppablePipeline($myInvocation.CommandOr\
igin)
        $steppablePipeline.Begin($PSCmdlet)
    } catch {
        throw
    }
}

process
{
    try {
        $steppablePipeline.Process($_)
    } catch {
        throw
    }
}

end
{
    try {
        $steppablePipeline.End()
    } catch {
        throw
    }
}
```

```
<#

.ForwardHelpTargetName Microsoft.PowerShell.Utility\ConvertTo-Html
.ForwardHelpCategory Cmdlet

#>
```

This isn't wrapped in a function, so that's the first thing we'll do in the next step (which we'll put into a file in Step2, so you can differentiate).

Modifying the Proxy

In addition to wrapping our proxy code in a function, we're going to play with the `-Head` parameter. We're not going to remove access to it; we want users to be able to pass content to `-Head`. We just want to intercept it, and add our stylesheet to it, before letting the underlying `ConvertTo-HTML` command take over. So we'll need to test and see if our command was even run with `-Head` or not, and if it was, grab that content and concatenate our own. The final result:

```
function NewConvertTo-HTML {
[CmdletBinding(DefaultParameterSetName='Page',
 HelpUri='http://go.microsoft.com/fwlink/?LinkID=113290',
 RemotingCapability='None')]
param(
    [Parameter(ValueFromPipeline=$true)]
    [psobject]
    ${InputObject},

    [Parameter(Position=0)]
    [System.Object[]]
    ${Property},

    [Parameter(ParameterSetName='Page', Position=3)]
    [string[]]
    ${Body},

    [Parameter(ParameterSetName='Page', Position=1)]
    [string[]]
    ${Head},

    [Parameter(ParameterSetName='Page', Position=2)]
    [ValidateNotNullOrEmpty()]
    [string]
```

```
    ${Title},

    [ValidateNotNullOrEmpty()]
    [ValidateSet('Table','List')]
    [string]
    ${As},

    [Parameter(ParameterSetName='Page')]
    [Alias('cu','uri')]
    [ValidateNotNullOrEmpty()]
    [uri]
    ${CssUri},

    [Parameter(ParameterSetName='Fragment')]
    [ValidateNotNullOrEmpty()]
    [switch]
    ${Fragment},

    [ValidateNotNullOrEmpty()]
    [string[]]
    ${PostContent},

    [ValidateNotNullOrEmpty()]
    [string[]]
    ${PreContent})

begin
{
    try {
        $outBuffer = $null
        if ($PSBoundParameters.TryGetValue('OutBuffer', [ref]$outBuffer))
        {
            $PSBoundParameters['OutBuffer'] = 1
        }
        $wrappedCmd = $ExecutionContext.InvokeCommand.GetCommand('Microsoft.PowerShe\
ll.Utility\ConvertTo-Html',
        [System.Management.Automation.CommandTypes]::Cmdlet)

        # create our css
        $css += @'
        <style>
        th { color:white; background-color: black;}
        body { font-family: Calibri; padding: 2px }
```

```
        </style>
'@

        # was -head specified?
        if ($PSBoundParameters.ContainsKey('head')) {
            $PSBoundParameters.head += $css
        } else {
            $PSBoundParameters += @{'Head'=$css}
        }

        $scriptCmd = {& $wrappedCmd @PSBoundParameters }
        $steppablePipeline = $scriptCmd.GetSteppablePipeline($myInvocation.CommandOr\
igin)
        $steppablePipeline.Begin($PSCmdlet)
    } catch {
        throw
    }
}

process
{
    try {
        $steppablePipeline.Process($_)
    } catch {
        throw
    }
}

end
{
    try {
        $steppablePipeline.End()
    } catch {
        throw
    }
}
<#

.ForwardHelpTargetName Microsoft.PowerShell.Utility\ConvertTo-Html
.ForwardHelpCategory Cmdlet

#>
}
```

Our changes begin at around line 63, with the `#create our css` comment. Under that, we check to see if `-head` had been specified; if it was, we append our CSS to it. If not, we add a "head" parameter to $PSBoundParameters. Then we let the proxy function continue just as normal.

You may want to clean up references to the original version by deleting the HelpUri link in cmdletbinding as well as the forwarded help link at the end. Or if you have created your own help documentation you can delete the forward links altogether.

Adding or Removing Parameters

You're likely to run into occasions when you do want to add or remove a parameter. For example, a new parameter might simplify usage or unlock functionality; removing a parameter might enable you to hard-code a value than the ultimate user shouldn't be changing. The real key is the `$PSBoundParametersCollection`.

Adding a Parameter

Adding a parameter is as easy as declaring it in your proxy function's `Param()` block. Add whatever attributes you like, and you're good to go. You just want to *remove* the added parameter from `$PSBoundParameters` before the underlying command executes, since that command won't know what to do with your new parameter.

```
$PSBoundParameters.Remove('MyNewParam')
$scriptCmd = {& $wrappedCmd @PSBoundParameters }
```

Just remove it before that `$scriptCmd` line, and you're good to go.

Removing a Parameter

This is even easier - just delete the parameter from the `Param()` block! If you're removing a parameter that's mandatory, you'll need to internally provide a value with it. For example:

```
$PSBoundParameters += @{'RemovedParam'=$MyValue}
$scriptCmd = {& $wrappedCmd @PSBoundParameters }
```

This will re-connect the `-RemovedParam` parameter, feeding it whatever's in $MyValue, before running the underlying command.

Your Turn

Now it's your turn to create a proxy function.

Start Here

In this exercise, you'll be extending the `Export-CSV` command. However, you're not going to "overwrite" the existing command. Instead, you'll be creating a *new* command that uses `Export-CSV` under the hood.

Your Task

Create a proxy function named `Export-TDF`. This should be a wrapper around `Export-CSV`, and should not include a `-Delimiter` parameter. Instead, it should hard-code the delimiter to be a tab. Hint: you can specify a tab by putting a backtick, followed by the letter "t," inside double quotes.

Our Take

Here's what we came up with - also in the lab-results folder in the downloadable code.

```
function Export-TDF {
[CmdletBinding(DefaultParameterSetName='Delimiter',
SupportsShouldProcess=$true,
ConfirmImpact='Medium',
HelpUri='http://go.microsoft.com/fwlink/?LinkID=113299')]
param(
    [Parameter(
      Mandatory=$true,
      ValueFromPipeline=$true,
      ValueFromPipelineByPropertyName=$true
      )]
    [psobject]$InputObject,

    [Parameter(Position=0)]
    [ValidateNotNullOrEmpty()]
    [string]$Path,

    [Alias('PSPath')]
    [ValidateNotNullOrEmpty()]
    [string]$LiteralPath,
```

```
    [switch]$Force,

    [Alias('NoOverwrite')]
    [switch]$NoClobber,

    [ValidateSet('Unicode','UTF7','UTF8','ASCII','UTF32',
    'BigEndianUnicode','Default','OEM')]
    [string]$Encoding,

    [switch]$Append,

    [Parameter(ParameterSetName='UseCulture')]
    [switch]$UseCulture,

    [Alias('NTI')]
    [switch]$NoTypeInformation
    )

begin
{
    try {
        $outBuffer = $null
        if ($PSBoundParameters.TryGetValue('OutBuffer', [ref]$outBuffer))
        {
            $PSBoundParameters['OutBuffer'] = 1
        }
        $wrappedCmd = $ExecutionContext.InvokeCommand.GetCommand('Microsoft.PowerShe\
ll.Utility\Export-Csv',
         [System.Management.Automation.CommandTypes]::Cmdlet)
        $PSBoundParameters += @{'Delimiter'="`t"}
        $scriptCmd = {& $wrappedCmd @PSBoundParameters }
        $steppablePipeline = $scriptCmd.GetSteppablePipeline($myInvocation.CommandOr\
igin)
        $steppablePipeline.Begin($PSCmdlet)
    } catch {
        throw
    }
}

process
{
    try {
        $steppablePipeline.Process($_)
```

```
    } catch {
        throw
    }
}

end
{
    try {
        $steppablePipeline.End()
    } catch {
        throw
    }
}

} #close function
```

We really just removed one parameter definition and added one line of code to hard-code the delimiter. We removed the {} around the parameter names and lined things up in the Param() block the way we would normally write code. We also removed the forwarded help links. We would still need to create new comment based help for this command. Probably copying a lot of the help from the original command.

Once you understand the concepts, you can use Jeff's Copy-Command[49] function from the PSScriptTools module.

Let's Review

See if you can answer a couple of questions on proxy functions:

1. The boilerplate proxy function behaves exactly like what?
2. If you define an additional parameter in a proxy function, what must you do before the wrapped command is allowed to run?
3. If you delete a non-mandatory parameter definition in a proxy function, what must you do before the wrapped command is allowed to run?

Review Answers

Here are our answers:

[49] https://github.com/jdhitsolutions/PSScriptTools/blob/master/docs/Copy-Command.md

1. The command it wraps.
2. Remove the new parameter from $PSBoundParameters.
3. You don't need to do anything since the wrapped command can run without the removed parameter.

Just Enough Administration: A Primer

This is going to be an interesting chapter. On one hand, the topic doesn't have anything to do with creating better PowerShell tools and scripts. But it does affect *why* you might be creating something in the first place and *how* it might be used. This is, in other words, about *using* your tools, much like a controller script.

We're sure you are familiar with the concept of "least privilege." Microsoft believes this should apply to admins as well. PowerShell is an awesome tool for getting things done, especially across remote computers. But by default you have to have full admin rights on the remote server and you have access to *everything*. That may not always be desirable. Perhaps you want to give the help desk access to manage a few key services and nothing else. Or you want to give a department secretary a tool to manage the print spooler on the department print server? Or you need to give a developer team PowerShell remote access to a dev server.

This is where the idea of *Just Enough Administration*, or JEA, comes into play. To be honest, we've had something like this for quite awhile, but it was difficult to implement. Today the PowerShell team has made this much easier. We're going to cover enough basics to get you started. A good place to get started online for more information is at https://docs.microsoft.com/en-us/powershell/scripting/learn/remoting/jea/overview[50], or https://msdn.microsoft.com/en-us/library/dn896648.aspx[51].

Requirements

In order to work with JEA you will need a Windows platform (both client and server) with Windows Management Framework 5.1. You'll need full admin rights on the remote server to set up the JEA configuration. The original version of JEA depended on PowerShell's Desired State Configuration (DSC); the version we're working with is now standalone, and does not require DSC. Keep that in mind as you're exploring online, as the information you find won't necessarily be applicable in every case.

JEA support is *included* in PowerShell v5 and later, so you'll need that as well. If you've installed the WMF 5.1 you're good to go.

Working with JEA is not always simple as there are a lot of moving parts to get right. And since the whole point of JEA is to minimize access which should improve security, you definitely should be testing everything in a non-production setting. The last thing you want is a poorly developed JEA solution that leaves the server vulnerable.

[50] https://docs.microsoft.com/en-us/powershell/scripting/learn/remoting/jea/overview?view=powershell-7
[51] https://msdn.microsoft.com/en-us/library/dn896648.aspx

Theory of Operation

Hopefully, you're familiar with PowerShell Remoting. Normally, when you run a command like `Invoke-Command` or `Enter-PSSession`, you connect to the default *endpoint* on your target computer. That endpoint is wide-open, and allows only Administrators (by default) to connect. It basically lets you do anything you have permission to do.

But Remoting can define many endpoints on a single computer, and each endpoint can be deeply customized. An endpoint has an Access Control List, or ACL, which determines who can connect. Instead of being wide-open, it can have only a tiny set of commands that you define. It can be configured to run those commands under an alternate "Run As" account, the credentials to which are stored as part of the endpoint. These features are a little tricky to set up, and what JEA really does is make all that easier to use and manage. The idea is to set up a kind of "jump server" filled with JEA-managed endpoints. Each endpoint has very tightly locked-down capabilities, and only permits connections from specific users or groups. By connecting to a JEA endpoint, you can accomplish tasks that your normal account doesn't have permission to, and you can do it in a way that minimizes danger and damage if a piece of malware compromises your account. JEA is heavily used in Microsoft products like Azure Stack, and you'll see more of it in the coming years.

These endpoints can contain *your* tools as well as native PowerShell ones - and that's why this chapter is included in this book. This chapter is meant only to be a primer to JEA - an introduction. If it interests you, there's a lot more to learn about, and we'll continue to provide reference URLs as appropriate.

Roles

JEA can be considered a *role-based* administrative system. You decide what type of role to create, and what commands that role will be able to execute. These details are stored in a role capability file which, is a special type of PowerShell file that has a .psrc file extension. Remember, your goal here is to provide access to the tools and commands needed to achieve some role-related task, such as clearing a print queue, and **nothing more**.

Fortunately, you don't have to create the .psrc file by hand. Instead you'll use the `New-PSRoleCapabilityFile` cmdlet. At a minimum all you need to specify is a path.

```
New-PSRoleCapabilityFile -Path .\MyRoleFile.psrc
```

If you open the file you'll see that it looks a lot like a module manifest which makes sense because the file is describing the limitations. You may want to restrict access to:

- Providers like the registry
- specific cmdlets

- specific parameters with specific cmdlets
- specific external commands
- specific functions
- specific aliases
- specific variables

Essentially, unless you specify it, it won't be included in the role capability file. We're going to setup a role for the help desk to manage shares but in a limited manner.

New Role Capability File

```
$params = @{
    Path              = ".\ShareAdmins.psrc"
    Description       = "Share Admin"
    VisibleFunctions  = @("Get-SMBShare", "Get-SMBShareAccess", "Get-ShareSize")
    VisibleAliases    = "gcim"
    ModulesToImport   = "ShareAdmin"
    VisibleCmdlets    = @{Name = "Get-CimInstance"; Parameters = @{ Name = 'classname\
';
            ValidateSet                                                    = 'win32_share'
        }, @{Name = "filter"}
    }
```

Note that even though you could use wildcards with command names the recommended best practice is to explicitly list each command. This eliminates the possibility of providing access to an unanticipated command.

You may be wondering why `Get-SMBShare` and `Get-SMBShareAccess` are listed as functions and not cmdlets. If you run `Get-Command get-smbshare` you'll see that this command is actually a function.

Now for the tricky part. Your role configuration file needs to be part of a module. The module doesn't even have to do anything. You could have an empty .psm1 file, but the module have a subfolder called RoleCapabilities. In our case though, and most likely yours, we are going to include some custom tools in this module. The functions you define can use *any* command and won't be restricted. Although we do recommend you use the full cmdlet name to avoid any problems. We added this function to the module.

Get-ShareSize

```
Function Get-ShareSize {
[cmdletbinding()]
Param(
[Parameter(
    Position = 0,
    Mandatory,
    ValueFromPipelineByPropertyName
    )]
[string]$Path
)

Begin {
  Write-Verbose "[BEGIN  ] Starting: $($MyInvocation.Mycommand)"
} #begin

Process {
  Write-Verbose "[PROCESS] Getting share size for $path"

  #use full cmdlet names to avoid problems
  #these commands do not need to be specified in the psrc file
  $stats = Microsoft.PowerShell.Management\Get-Childitem -Path $Path -Recurse `
  -file | Microsoft.PowerShell.Utility\Measure-Object -Property Length -sum

  Microsoft.PowerShell.Utility\New-Object -TypeName PSObject -Property @{
    Path = $path
    FileCount = $stats.count
    FileSize = $stats.sum
  }
}

End {
  Write-Verbose "[END    ] Ending: $($MyInvocation.Mycommand)"
} #end

}
```

If you recall from the role file, we included this function name in the visible functions.

When you are finished, the module will need to be copied to the remote computer. For now, we'll create it in our working directory and copy files to the required locations.

Copy to module

```
#region copy to module

# Create a folder for the module
$modulename = "ShareAdmin"
#the path could also be a directory in $env:psmodulepath
$modulePath = Join-Path -path . -ChildPath $modulename
New-Item -ItemType Directory -Path $modulePath

<# Create an empty script module and module manifest. At least one file in the
module folder must have the same name as the folder itself.
#>
$path = (Join-Path -path $modulePath -ChildPath "$modulename.psm1")
New-Item -ItemType File -Path $path
$manifest = (Join-Path -path $modulePath -ChildPath "$modulename.psd1")
New-ModuleManifest -Path $manifest -RootModule "$modulename.psm1"

# Create the RoleCapabilities folder and copy in the PSRC file
$rcFolder = Join-Path -path $modulePath -ChildPath "RoleCapabilities"
```

Endpoints

Next, you need to create the endpoint. This is the PowerShell remoting configuration that you will create. You can see all current endpoints with the `Get-PSSessionConfiguration` cmdlet. Creating a new endpoint used to be much harder, but now we have an easy to use cmdlet. You will need to create a file with a .pssc file extension.

```
New-PSSessionConfigurationFile -Path .\MyEndpoint.pssc
```

Well - easy but not necessary simple. You could open up the file and manually edit it, assuming you knew what you were doing.

Perhaps the most important step here is to define the session type. For JEA you want to use `RestrictedRemoteServer`. This means the session will operate in something called `NoLanguage` mode which severely restricts what the user has access to. This mode will provide access to these cmdlets and aliases, which means you do not have to include them in your role configuration:

- The `Clear-Host` (cls, clear) command
- The `Exit-PSSession` (exsn, exit) command
- The `Get-Command` (gcm) command

- The `Get-FormatData` command
- The `Get-Help` command
- The `Measure-Object` (measure) command
- The `Out-Default` command
- The `Select-Object` (select) command

You also need to specify what account to run under. If at all possible, you should use a local virtual account (read up on those if you're not familiar with them). If you'll need a greater level of permissions, check the documentation for using different types of accounts.

The last bit is to tie in the role capabilities defined earlier as that's the whole point. You'll do this with a hashtable of a group, which the delegated user will belong to, and an hashtable that indicates the role capabilities. The value must be the name of the .psrc file but without the extension.

```
$roles = @{
  "Company\JEA_ShareAdmins" = @{RoleCapabilities = 'ShareAdmins'}
}
```

You can have multiple roles defined, and someone could belong to multiple groups. For your primer purposes we're keeping it simple.

When you are ready, go ahead and create the .pssc file.

```
$params = @{
 Path = ".\ShareAdmin.pssc"
 SessionType = "RestrictedRemoteServer"
 RunAsVirtualAccount = $True
 RoleDefinitions = $roles
 Description = "JEA Share Admin endpoint"
}
New-PSSessionConfigurationFile @params
```

Once you've completed your configuration you should test it.

```
Test-PSSessionConfigurationFile .\ShareAdmin.pssc
```

The configuration needs to be set up **on** the remote server so you'll need to copy the necessary files to the server.

```
$s = New-PSSession -ComputerName chi-fp02
#copy the pssc file to C:\
copy .\shareadmin.pssc  -Destination C:\ -ToSession $s

#copy the module with the role configuration
copy .\ShareAdmin -Container -Recurse `
-Destination 'C:\Program Files\WindowsPowerShell\Modules' -ToSession $s
```

The last step is to register it and bring the special endpoint to life with `Register-PSSessionConfiguration`.

```
invoke-command {
 Register-PSSessionConfiguration -Path c:\shareadmin.pssc -Name "ShareAdmins"
} -Session $s
```

This will restart with WinRM service on the remote computer, breaking any open sessions.

You can try it out using an account that is a member of the specified group.

```
$enter = @{
  ComputerName = "chi-fp02"
  ConfigurationName =-"ShareAdmins"
  Credential = "company\jshields"
}
Enter-PSSession @enter
```

The user will only have access to the commands you've specified.

```
[chi-fp02]: PS>get-command | select name

Name
----
Clear-Host
Exit-PSSession
Get-CimInstance
Get-Command
Get-FormatData
Get-Help
Get-ShareSize
Get-SmbShare
Get-SmbShareAccess
```

```
Measure-Object
Out-Default
Select-Object
```

You can only run what has been specified in the role capability file, including the `Get-ShareSize` function from our module.

```
[chi-fp02]: PS>gcim win32_share -filter "name='it'"

Name Path      Description
---- ----      -----------
IT   E:\Shared IT Data Share


[chi-fp02]: PS>gcim win32_bios
Cannot validate argument on parameter 'ClassName'. The argument "win32_bios"
does not belong to the set "win32_share" specified by the ValidateSet
attribute. Supply an argument that is in the set and then try the command again.
    + CategoryInfo          : InvalidData: (:) [Get-CimInstance],
    + ParameterBindingValidationException
    + FullyQualifiedErrorId : ParameterArgumentValidationError,Get-CimInstance

[chi-fp02]: PS>get-smbshare public | get-sharesize

Path             FileSize FileCount
----             -------- ---------
E:\shared\Public  8932930       121


[chi-fp02]: PS>get-childitem e:\shared\public
The term 'Get-ChildItem' is not recognized as the name of a cmdlet, function,
script file, or operable program. Check
the spelling of the name, or if a path was included, verify that the path
is correct and try again.
    + CategoryInfo          : ObjectNotFound: (Get-ChildItem:String) [],
    + CommandNotFoundException
    + FullyQualifiedErrorId : CommandNotFoundException
```

Should you need to update the role or module, you can make the changes to those files and copy them again to the server. They should take affect the next time a JEA session is opened. If you want to wipe the entire configuration and start all-over, unregister the configuration.

```
Invoke-command {
 Unregister-PSSessionConfiguration -Name shareadmins
} -computername chi-fp02
```

Because this is such a specialized topic, and we've only provide some basic guidance we'll save you the frustration of creating a JEA based tool. And it can be frustrating because nothing is available for the user to run, unless it is specified as part of the endpoint.

Let's Review

Let's review and see what you picked up in the chapter.

1. What type of PowerShell file contains the role definitions?
2. What type of PowerShell file contains the session configuration?
3. Where does the role capability file need to be stored?
4. Are your custom module functions limited in scope or execution?

Review Answers

Did you come up with answers like these?

1. A .psrc file.
2. A .pssc file.
3. The .psrc file must be copied to the RoleCapabilities file of a module which is then copied to the remote server.
4. Generally not. These functions can use commands even if not explicitly granted in the .psrc file. Although we recommend using the fully cmdlet name and including the function in the VisibleFunctions setting.

PowerShell in ASP.NET: A Primer

One interesting fact about PowerShell's construction is that PowerShell *itself* - the engine that runs commands - is a .NET Framework class. The PowerShell you're used to - either the console or the ISE, perhaps - is actually a *hosting application*. These applications give you a way of feeding stuff to the actual engine, and a way for the engine's output to be shown to you. Technically, any .NET Framework application can "host" the PowerShell engine - including ASP.NET.

Hosting the engine in an ASP.NET web application is a cool way to create web-based self-service tools - a different kind of GUI than WPF or WinForms, basically. PowerShell would run on the web server, under whatever identity you've configured IIS to run ASP.NET as. This opens up a ton of useful possibilities.

Caveats

There are a few things we need to make clear:

- This chapter isn't going to teach you ASP.NET. There are entire series of books that will do that; we're assuming that you know ASP.NET already.
- The content in this chapter is written for "full" ASP.NET. As of this writing, ASP.NET Core 1.0 can't host PowerShell all that well or easily. Therefore, this chapter applies only to Microsoft Windows, not other operating systems which may support ASP.NET Core.
- This chapter isn't going to teach you IIS, either. We expect that you know how to configure IIS to run ASP.NET, including dealing with credentials, identities, and so on.

You also need to know that using the "raw" engine is a little different from what you're used to in the ISE or PowerShell console. The *runspaces* created by the engine aren't populated with all the global variables that you're used to, for example - it's the *console* (or ISE) which creates those, not PowerShell itself. So you may find that you need to do a little more work to set up some commands to run properly.

The Basics

You'll need to start by making sure you have the PowerShell Reference Assemblies in your IDE (e.g., Visual Studio). Specifically, you need the `System.Management.Automation` reference assembly. Beware of unofficial NuGet packages - these can be outdated or even contain malware. The official one[52] is owned by "PowerShellTeam," which is what you want to look for.

[52] https://www.nuget.org/packages/System.Management.Automation

You'll then need to add, in your ASP.NET code, a `Using` reference for `System.Management.Automation`.

Then you need to think about what you'll do with the eventual command output. PowerShell returns everything as collections of objects; you'll need to plan for a way to display that information. You could, for example, pipe your command to `Out-String`, which will cause PowerShell to render the objects as text using its own formatting subsystem - more or less what the console host application does when you run a command. Or, you could construct some big graphical display, like the Exchange Management Console, complete with icons and whatever information you want. It's up to you.

When you're ready, it's pretty easy to run a command:

```
var shell = PowerShell.Create();
shell.Commands.AddScript("Get-Service | Out-String");
// this also works and is equivalent:
// shell.AddCommand("Get-Service");
// shell.AddCommand("Out-String");
var results - shell.Invoke();
foreach (var psObject in results) {
    // use psObject
}
```

The `AddCommand` technique is a bit harder to use, as each command is added individually. You chain an `AddParameter()` call to specify parameters:

```
shell.AddCommand("Get-Service").
      AddParameter("Name","WinRM")
```

The above also assumes you want to use the default runspace, which loads most of the core PowerShell command automatically. But you can also instantiate runspaces that contain only a custom set of commands - the official docs have examples on doing so.

Simply enumerate the results and you're done. Here's a good walk-through[53] if you'd like to explore more, and the Microsoft documentation[54] is definitely worth a read.

Beyond ASP.NET

There are third-party products that allow PowerShell scripts to be run by IIS, enabling those scripts to create web pages which are transmitted to requesting clients. In other words, you basically use PowerShell instead of ASP.NET on the web server. PowerShell Server[55] is one such third-party tool, Posh Server[56] is another.

[53] http://jeffmurr.com/blog/?p=142
[54] https://dotnet.microsoft.com/learn/aspnet/what-is-aspnet
[55] http://powershellserver.com
[56] http://poshserver.net

Part 4: The Data Connection

In this Part, we'll look at various kinds of structured data that you may need to work with from within PowerShell. It might mean grabbing something from some type of data source, or perhaps putting something into a particular data type. We'll try and use some realistic examples that illustrate how to use these different structured data constructs, and give you some tips for keeping out of trouble.

Working with SQL Server Data

We often see people struggle - really hard, in some cases - to store data in Microsoft Excel, using PowerShell to automate the process. This makes us sad. Programmatically, Excel is kind of a hunk of junk. Sure, it can make charts and graphs - but only with significant effort and a lot of delicacy. But, people say, "I already have it!" This also makes us sad, because for the very reasonable price of $FREE, you can have SQL Server (Express), and in fact you probably have some flavor of SQL Server on your network that you could use. But why?

- SQL Server is easy to use from PowerShell code. Literally a handful of lines, and you're done.
- SQL Server Reporting Services (also free in the Express edition) can turn SQL Server data into *gorgeous* reports with charts and graphs - and can automate the production and delivery of those reports with zero effort from you.
- SQL Server is something that many computers can connect to at once, meaning you can write scripts that run on servers, letting those servers update their own data in SQL Server. This is faster than a script which reaches out to query many servers in order to update a spreadsheet.

We don't know how to better evangelize using SQL Server for data storage over Microsoft Excel.

SQL Server Terminology and Facts

Let's quickly get some terminology and basic facts out of the way.

- SQL Server is a service that runs on a *server*. Part of what you'll need to know, to use it, is the server's name. A single machine (physical or VM) can run multiple *instances* of SQL Server, so if you need to connect to an instance other than the default, you'll need the instance name also. The naming pattern is `SERVER\INSTANCE`.
- A SQL Server instance can host one or more *databases*. You will usually want a database for each major data storage purpose. For example, you might ask a DBA to create an "Administration Data" database on one of your SQL Server computers, giving you a place to store stuff.
- Databases have a *recovery mode* option. Without getting into a lot of details, you can use the "Simple" recovery mode (configurable in SQL Server Management Studio by right-clicking the database and accessing its Options page) if your data isn't irreplaceable and you don't want to mess with maintaining the database's log files. For anything more complex, either take a DBA to lunch, or read Don's *Learn SQL Server Administration in a Month of Lunches*.
- Databases contain *tables*, each of which is analogous to an Excel worksheet.

- Tables consist of *rows* (entities) and *columns* (fields), which correspond to the rows and columns of an Excel sheet.
- Columns have a *data type*, which determines the kind of data they can store, like text (NVARCHAR), dates (DATETIME), or numbers (INT). The data type also determines the data ranges. For example, NVARCHAR(10) can hold 10 characters; NVARCHAR(MAX) has no limit. INT can store smaller values than BIGINT, and bigger values than TINYINT.
- SQL Server defaults to Windows Authentication mode, which means the domain user account running your scripts must have permission to connect to the server (a *login*), and permission to use your database (a *database user*). This is the safest means of authentication as it doesn't require passwords to be kept in your script. If running a script as a scheduled task, the task can be set to "run as" a domain user with the necessary permissions.

Just seven little things to know, and you're good to go.

Even if you are the only person who will ever interact with stored data, you are still better off installing SQL Server Express (did we mention it is free?) instead of relying on Excel.

Connecting to the Server and Database

You'll need a *connection string* to connect to a SQL Server computer/instance, and to a specific database. If you're not using Windows Authentication, the connection string can also contain a clear-text username and password, which is a really horrible practice. We use ConnectionStrings.com[57] to look up connection string syntax, but here's the one you'll use a lot:

Use `Server=SERVER\INSTANCE;Database=DATABASE;Trusted_Connected=True;` to connect to a given server and instance (omit the `\INSTANCE` if you're connecting to the default instance) and database. Note that SQL Server Express usually installs, by default, as an instance named `SQLEXPRESS`. You can run `Get-Service` in PowerShell to see any running instances on a computer, and the service name will include the instance name (or just `MSSQLSERVER` if it's the default).

With that in mind, it's simple to code up a connection:

```
$conn = New-Object -Type System.Data.SqlClient.SqlConnection
$conn.ConnectionString = 'Server=SQL1;Database=MyDB;Trusted_Connection=True;'
$conn.Open()
```

You can leave the connection open for your entire script; be sure to run `$conn.Close()` when you're done, though. It's not a tragedy to not close the connection; when your script ends, the connection object will vanish, and SQL Server will automatically close the connection a bit later. But if you're using a server that's pretty busy, the DBA is going to get in your face about leaving the connection

[57] http://connectionstrings.com

open. And, if you run your script multiple times in a short period of time, you'll create a new connection each time rather than re-using the same one. The DBAs will definitely notice this and get agitated.

You do not need to have any SQL Server software installed locally for these steps as they are relying on out-of-the-box bits from the .NET Framework. And even if you are working with a local SQL installation, you should still follow SQL Server best practices.

Writing a Query

The next thing you need to do is retrieve, insert, update, or remove some data. This is done by writing queries in the Transact-SQL (T-SQL) language, which corresponds with the ANSI SQL standard, meaning most queries look basically the same on most database servers. There's a great free online SQL tutorial[58] if you need one, but we'll get you started with the basics.

To do this, you'll need to know the *table* and *column names* from your database. SQL Server Management Studio is a good way to discover these.

For the following sections, we're going to focus on *query syntax*, and then give you an example of how we might build that query in PowerShell. Once your query is in a variable, it's easy enough to run it - and we'll cover how to do that in a bit. Also, we're not going to be providing exhaustive coverage of SQL syntax; we're covering the basics. There are plenty of resources, including the aforementioned online tutorial, if you need to dig deeper.

Adding Data

Adding data is done by using an `INSERT` query. The basic syntax looks like this:

```
INSERT INTO <tablename>
    (Column1, Column2, Column3)
    VALUES (Value1, Value2, Value3)
```

So you'll need to know the name of the table you're adding data to, and you'll need to know the column names. You also need to know a bit about how the table was defined. For example, if a "Name" column is marked as mandatory (or "NOT NULL") in the table design, then you *must* list that column and provide a value for it. Sometimes, a table may define a default value for a column, in which case you can leave the column out if you're okay with that default value. Similarly, a table can permit a given column to be empty (NULL), and you can omit that column from your list if you don't want to provide a value.

[58] http://www.w3schools.com/sql/

Whatever order you list the columns in, your values must be in the same order. You're not forced to use the column order that the table defines; you can list them in any order.

Numeric values aren't delimited in T-SQL. String values are delimited in single quotes; any single quotes *within* a string value (like "O'Leary") must be doubled ("O''Leary") or your query will fail. Dates are treated as strings, and are delimited with single quotes.

It's dangerous to build queries from user-entered data. Doing so opens your code to a kind of attack called *SQL Injection*. We're assuming that you plan to retrieve things like system data, which shouldn't be nefarious, rather than accepting input from users. The safer way to deal with user-entered data is to create a stored procedure to enter the data, but that's well beyond the scope of this book.

We might build a query in PowerShell like this:

```
$ComputerName = "SERVER2"
$OSVersion = "Win2012R2"
$query = "INSERT INTO OSVersion (ComputerName,OS) VALUES('$ComputerName','$OSVersion\
')"
```

This assumes a table named OSVersion, with columns named ComputerName and OS. Notice that we've put the entire query into double quotes, allowing us to just drop variables into the `VALUES` list.

We always put our query in a variable, because that makes it easy to output the query text by using `Write-Verbose`. That's a great way to debug queries that aren't working, since you get to see the actual query text with all the variables "filled-in."

Removing Data

A `DELETE` query is used to delete rows from a table, and it is almost always accompanied by a `WHERE` clause so that you don't delete *all the rows*. Be really careful, as there's no such thing as an "UNDO" query!

```
DELETE FROM <tablename> WHERE <criteria>
```

So, suppose we're getting ready to insert a new row into our table, which will list the OS version of a given computer. We don't know if that computer is already listed in the table, so we're going to just delete any existing rows before adding our new one. Our `DELETE` query might look like this:

```
$query = "DELETE FROM OSVersions WHERE ComputerName = '$ComputerName'"
```

There's no error generated if you attempt to delete rows that don't exist.

Changing Data

An `UPDATE` query is used to change an existing row, and is accompanied by a `SET` clause with the changes, and a `WHERE` clause to identify the rows you want to change.

```
UPDATE <tablename>
   SET <column> = <value>, <column> = <value>
   WHERE <criteria>
```

For example:

```
$query = "UPDATE DiskSpaceTracking `
          SET FreeSpaceOnSysDrive = $freespace `
          WHERE ComputerName = '$ComputerName'"
```

We'd ordinarily do that all on one line; we've broken it up here just to make it fit more easily in the book. This assumes that $freespace contains a numeric figure, and that $ComputerName contains a computer name.

In SQL Server, column names aren't case-sensitive.

Retrieving Data

Finally, the big daddy of queries, the `SELECT` query. This is the only one that returns data (although the other three will return the number of rows they affected). This is also the most complex query in the language, so we're really only tackling the basics.

```
SELECT <column>,<column>
       FROM <tablename>
       WHERE <criteria>
       ORDER BY <column>
```

The `WHERE` and `ORDER BY` clauses are optional, and we'll come to them in a moment.

Beginning with the core `SELECT`, you follow with a list of columns you want to retrieve. While the language permits you to use `*` to return all columns, this is a poor practice. For one, it performs slower than a column list. For another, it makes your code harder to read. So stick with listing the columns you want.

The `FROM` clause lists the table name. This can get a ton more complex if you start doing multi-table joins, but we're not getting into that in this book.

A `WHERE` clause can be used to limit the number of rows returned, and an `ORDER BY` clause can be used to sort the results on a given column. Sorting is ascending by default, or you can specify descending. For example:

```
$query = "SELECT DiskSpace,DateChecked `
          FROM DiskSpaceTracking `
          WHERE ComputerName = '$ComputerName' `
          ORDER BY DateChecked DESC"
```

Creating Tables Programmatically

It's also possible to write a *data definition language* (DDL) query that creates tables. The four queries we've covered up to this point are *data manipulation language* (DML) queries. The ANSI specification doesn't cover DDL as much as DML, meaning DDL queries differ a lot between server brands. We'll continue to focus on T-SQL for SQL Server; we just wanted you to be aware that you won't be able to re-use this syntax on other products without some tweaking.

```
CREATE TABLE <tablename> (
    <column> <type>,
    <column> <type>
)
```

You list each column name, and for each, provide a datatype. In SQL Server, you'll commonly use:

- Int or BigInt for integers
- VarChar(x) or VarChar(MAX) for string data; "x" determines the maximum length of the field while "MAX" indicates a binary large object (BLOB) field that can contain any amount of text.
- DateTime

You want to use the smallest data type possible to store the data you anticipate putting into the table, because oversized columns can cause a lot of wasted disk space.

Running a Query

You've got two potential types of queries: ones that return data (`SELECT`) and ones that don't (pretty much everything else). Running them starts the same:

```
$command = New-Object -Type System.Data.SqlClient.SqlCommand
$command.Connection = $conn
$command.CommandText = $query
```

This assumes `$conn` is an open connection object, and that `$query` has your T-SQL query. How you run the command depends on your query. For queries that don't return results:

```
$command.ExecuteNonQuery()
```

That *can* produce a return object, which you can pipe to `Out-Null` if you don't want to see it. For queries that produce results:

```
$reader = $command.ExecuteReader()
```

This generates a *DataReader* object, which gives you access to your queried data. The trick with these is that they're *forward-only*, meaning you can read a row, and then move on to the next row - but you can't go back to read a previous row. Think of it as an Excel spreadsheet, in a way. Your cursor starts on the first row of data, and you can see all the columns. When you press the down arrow, your cursor moves down a row, and you can only see *that* row. You can't ever press up arrow, though - you can only keep going down the rows.

You'll usually read through the rows using a `While` loop:

```
while ($reader.read()) {
  #do something with the data
}
```

The `Read()` method will advance to the next row (you actually start "above" the first row, so executing `Read()` the first time doesn't "skip" any data), and return True if there's a row after that.

To retrieve a column, inside the `While` loop, you run `GetValue()`, and provide the *column ordinal number* of the column you want. This is why it's such a good idea to explicitly list your columns in your `SELECT` query; you'll know which column is in what position. The first column you listed in your query will be `0`, the one after that `1`, and so on.

So here's a full-fledged example:

```
$conn = New-Object -Type System.Data.SqlClient.SqlConnection
$conn.ConnectionString = 'Server=SQL1;Database=MyDB;Trusted_Connection=True;'
$conn.Open()

$query = "SELECT ComputerName,DiskSpace,DateTaken FROM DiskTracking"

$command = New-Object -Type System.Data.SqlClient.SqlCommand
$command.Connection = $conn
$command.CommandText = $query
$reader = $command.ExecuteReader()

while ($reader.read()) {
    [pscustomobject]@{'ComputerName' = $reader.GetValue(0)
                      'DiskSpace' = $reader.GetValue(1)
                      'DateTaken' = $reader.GetValue(2)
                     }
}

$conn.Close()
```

This snippet will produce objects, one object for each row in the table, and with each object having three properties that correspond to three of the table columns.

If by chance you don't remember your column positions, you can use something like this to auto-discover the column number.

```
while ($reader.read()) {
    [pscustomobject]@{
    'ComputerName' = $reader.GetValue($reader.getordinal("computername"))
    'DiskSpace' = $reader.GetValue($reader.getordinal("diskspace"))
    'DateTaken' = $reader.GetValue($reader.getordinal("datetaken"))
    }
}
```

Regardless of the approach we'd usually wrap this in a `Get-` function, so that we could just run the function and get objects as output. Or a corresponding `Set-`, `Update-` or `Remove-` function depending on your SQL query.

Invoke-Sqlcmd

If you by chance have installed a local instance of SQL Server Express, you will also have a set of SQL-related PowerShell commands and a SQLSERVER PSDrive. We aren't going to cover them as this isn't a SQL Server book. But you will want to take advantage of `Invoke-Sqlcmd`.

Instead of dealing with the .NET Framework to create a connection, command and query, you can simply invoke the query.

```
Invoke-Sqlcmd "Select Computername,Diskspace,DateTaken from DiskTracking" `
-Database MyDB
```

You can use any of the query types we've shown you in this chapter. One potential downside to this approach in your toolmaking is that obviously this will only work locally, or where the SQL Server modules have been installed. And there is a bit of a lag while the module is initially loaded.

Thinking About Tool Design Patterns

If you've written a tool that retrieves or creates some data that you intend to put into SQL Server, then you're on the right track. A next step would be a tool that inserts the data into SQL Server (Export-Something), and perhaps a tool to read the data back out (Import-Something). This approach maintains a good design pattern of each tool doing one thing, and doing it well, and lets you create tools that can be composed in a pipeline to perform complex tasks. You can read a bit more about that approach, and even get a "generic" module for piping data in and out of SQL Server databases in Ditch Excel: Making Historical & Trend Reports in PowerShell[59], a free ebook.

Let's Review

Because we don't want to assume that you have access to a SQL Server computer, we aren't going to present a hands-on experience in this chapter. However, we do encourage you to try and answer these questions:

1. How to you prevent `DELETE` from wiping out a table?
2. What method do you use to execute an `INSERT` query?
3. What method reads a single database row from a reader object?

Review Answers

Here are our answers:

1. Specify a `WHERE` clause to limit the deleted rows.
2. The `ExecuteNonQuery()` method.
3. The `Read()` method.

[59] https://www.gitbook.com/book/devopscollective/ditch-excel-making-historical-trend-reports-in-po

Working with XML Data

As a PowerShell toolmaker you may have a need to work with a variety of file types and sources. One such type, which can appear daunting at first, is XML. Perhaps you need to get data from XML to use it with your tool. Or perhaps your tool needs to create an XML document. In this chapter, we'll explore a variety of ways you might interact with XML in your toolmaking.

Simple: CliXML

If you need to store results from a PowerShell command in a rich format that you intend to use again in a future PowerShell session, this is easily managed with the Clixml cmdlets `Export-Clixml` and `Import-Clixml`.

Remember that PowerShell tools should do one thing and typically they write objects to the pipeline. You shouldn't really need to incorporate these cmdlets in your functions except for rare exceptions. Although you might include them in a controller script.

A great benefit of using `Export-Clixml` is that it also stores type information along with the data. When you import the file PowerShell recreates the objects. Note that these files can only be used in PowerShell.

You might decide to export drive information.

```
Get-CimInstance win32_logicaldisk -filter "drivetype=3" |
Export-Clixml .\disks.xml
```

You can view the file in Notepad but you shouldn't need to modify it. Later, perhaps as part of another scripted process, you may want to work with the results. Easy. Import the file.

```
$d = import-clixml .\disks.xml
```

The variable $d now holds the same values as the original command and you can work with it the same way. If you need to work with data between PowerShell sessions, these are the best cmdlets.

Importing Native XML

Of course XML is a long-standing industry format and you may need to consume or work with native XML files, perhaps something created outside of PowerShell. Since the XML data is irrelevant for our purposes, we'll have some fun with an XML file called BandData.xml which you can find in the corresponding code chapter.

To bring this data into PowerShell all we need to do is get the content and tell PowerShell to treat it as an XML document.

```
[xml]$data = Get-content .\BandData.xml
```

The variable $data is now an XML document which we can navigate like any rich object. We recommend using tab completion to help properly format names with special characters or spaces.

```
PS C:\> $data

xml                             #comment                            Bands
---                             --------                            -----
version="1.0" encoding="UTF-8"  ...                                 Bands


PS C:\> $data.'#comment'

 This is a demonstration XML file

PS C:\> $data.Bands

Band
----
{Name, Name, Name, Name...}

PS C:\> $data.bands.band

Name Lead            Members
---- ----            -------
Name Steven Tyler    Members
Name Geddy Lee       Members
Name Ozzie Osbourne  Members
Name Joe Elliott     Members
Name Bret Michaels   Members
Name Vince Neil      Members
```

```
Name Jim Morrison    Members
Name Kurt Cobain     Members
Name Ian Gillan      Members
...
PS C:\> $data.bands.band[0]

Name Lead         Members
---- ----         -------
Name Steven Tyler Members
PS C:\> $data.bands.band[0].name

Year City       #text
---- ----       -----
1970 Boston, MA Aerosmith


PS C:\> $data.bands.band[0].members

Member
------
{Tom Hamilton, Joey Kramer, Joe Perry, Brad Whitford}
```

In this file the core data is a `Band` object. In XML-speak this is a `node`. The tricky part is turning this data back into a meaningful object you can use in PowerShell.

```
$data.bands.band |
Select-Object -property @{Name="Name";Expression = {$_.name.'#text'}},
@{Name="Founded";Expression={$_.name.Year}},
@{Name="Lead";Expression={$_.lead}},
@{Name="Members";Expression={$_.members.member}}
```

This should give you output like this:

```
Name           Founded Lead            Members
----           ------- ----            -------
Aerosmith      1970    Steven Tyler    {Tom Hamilton, Joey Kramer, Joe Perry,
                                        Brad Whitford}
Rush           1968    Geddy Lee       {Alex Lifeson, Neil Peart}
Black Sabbath  1968    Ozzie Osbourne  {Tony Iommi, Geezer Butler, Bill Ward}
Def Leppard            Joe Elliott     {Rick Allen, Phil Collen, Tony Kenning,
                                        Rick Savage}
Poison                 Bret Michaels   {Rikki Rockett, C.C. DeVille,
```

```
                                        Bobby Dall}
...
```

Modify XML Data

Let's say you are a big Poison fan and need to update missing information. First, you need to select the specific node. You can use `Where-Object` to filter the nodes:

```
PS C:\> $p = $data.bands.band | where {$_.Name.'#text' -eq 'Poison'}
PS C:\> $p

Name Lead          Members
---- ----          -------
Name Bret Michaels Members
```

Or if you have experience with InfoPath and XML queries (which is outside the scope of this book) you can use `Select-XML`:

```
PS C:\> $data.SelectNodes("//Bands/Band[Name='Poison']")

Name Lead          Members
---- ----          -------
Name Bret Michaels Members
```

The Year property is a child of the Name property.

```
PS C:\> $p.name

Year City #text
---- ---- -----
          Poison
```

The band was founded in 1983 in Mechanicsburg, Pennsylvania so let's update.

```
PS C:\> $p.name.year = '1983'
PS C:\> $p.name.city = 'Mechanicsburg, PA'
PS C:\> $p.name

Year City             #text
---- ----             -----
1983 Mechanicsburg, PA Poison
```

Add XML Data

Or let's say you need to add something to this XML file. To do this, you need to have some understanding of how the XML file is laid out. In looking at the file in a text editor, we can determine that if we want to add an band object for the group Cream, we will need to eventually have XML that looks like this:

```
<Band>
    <Name Year="1966" City="London, England">Cream</Name>
    <Lead>Eric Clapton</Lead>
    <Members>
        <Member>Ginger Baker</Member>
        <Member>Jack Bruce</Member>
    </Members>
</Band>
```

The first step is to create an empty XML element called 'Band'.

Don't forget that XML is case-sensitive.

```
$band = $data.CreateNode("element","Band","")
```

This node has child elements of *Name*, *Lead* and *Members*. The *Name* element has additional properties called *attributes* for the founding year and location. We'll have to accommodate those as well. In fact, let's create the *Name* element.

```
$name = $data.CreateElement("Name")
```

The band name will be the text value of this node so that is easily set:

```
$name.InnerText = "Cream"
```

The attributes are a bit trickier. You create them as distinct elements:

```
$y = $data.CreateAttribute("Year")
$y.InnerText = "1966"
$c = $data.CreateAttribute("City")
$c.InnerText = "London, England"
```

And then add them to their parent element, in this case the *Name* element.

```
$name.Attributes.Append($y)
$name.Attributes.Append($c)
```

You can verify by checking the OuterXML property.

```
PS C:\> $name.OuterXml
<Name Year="1966" City="London, England">Cream</Name>
```

If this looks good you can append this to the *Band* element.

```
$band.AppendChild($name)
```

Follow the same steps to add the *Lead* element.

```
$LeadMember =  $data.CreateElement("Lead")
$LeadMember.InnerText = "Eric Clapton"
$band.AppendChild($LeadMember)
```

The *Members* node is a bit more complicated since it has child objects of *Member* but hopefully by now you've recognized the pattern.

```
$members = $data.CreateNode("element","Members","")
$people = "Ginger Baker", "Jack Bruce"

foreach ($item in $people) {
    $m = $data.CreateElement("Member")
    $m.InnerText = $item
    $members.AppendChild($m)
}

#add members to the band node
$band.AppendChild($members)
```

Finally, add the new band object to the collection.

```
$data.Bands.AppendChild($band)
```

Saving XML

All we've done to this point is update the object. To update the file itself, we need to save it by specifying a path.

```
$data.Save('c:\work\banddata.xml')
```

You specify the original file if you want to update it. We recommend using complete and absolute path names. Relative paths and shortcut PSDrives may not work.

ConvertTo-Xml

We mentioned at the beginning of this chapter that the `-Clixml` cmdlets are an easy way to convert PowerShell data to XML. But those files are intended primarily for use within PowerShell. What if you need to create XML files to be used outside of PowerShell? That's where `ConvertTo-Xml` comes into play.

You can convert any output but we'll keep it simple and limit ourselves to data from the local computer. Pipe any cmdlet to `ConvertTo-Xml` to create an XML document.

```
Get-CimInstance win32_service | ConvertTo-Xml
```

If you look through the document you'll realize the cmdlet converted all properties, not just what you see by default. Most likely you will want to be a little more selective.

```
$s = Get-CimInstance win32_service -ComputerName $env:computername |
Select-Object * -ExcludeProperty CimClass,Cim*Properties |
ConvertTo-Xml
```

This cmdlet will create generic *Object* nodes.

```
PS C:\> $s.objects.object[0]

Type                                          Property
----                                          --------
System.Management.Automation.PSCustomObject {PSShowComputerName, Name, S...}
```

The cmdlet also does a decent job of capturing each property type.

```
PS C:\> $s.objects.object[0].Property

Name                    Type           #text
----                    ----           -----
PSShowComputerName      System.Boolean True
Name                    System.String  AdobeFlashPlayerUpdateSvc
Status                  System.String  OK
ExitCode                System.UInt32  0
DesktopInteract         System.Boolean False
ErrorControl            System.String  Normal
PathName                System.String  C:\windows\SysWOW64\Macromed\Flash
                                        \FlashPlayerUpdateService.exe
ServiceType             System.String  Own Process
StartMode               System.String  Manual
Caption                 System.String  Adobe Flash Player Update Service
Description             System.String  This service keeps your Adobe Flash
                                         Player installation up to...
InstallDate             System.Object
CreationClassName       System.String  Win32_Service
Started                 System.Boolean False
SystemCreationClassName System.String  Win32_ComputerSystem
SystemName              System.String  WIN81-ENT-01
AcceptPause             System.Boolean False
AcceptStop              System.Boolean False
DisplayName             System.String  Adobe Flash Player Update Service
ServiceSpecificExitCode System.UInt32  0
StartName               System.String  LocalSystem
State                   System.String  Stopped
```

```
TagId                 System.UInt32  0
CheckPoint            System.UInt32  0
ProcessId             System.UInt32  0
WaitHint              System.UInt32  0
PSComputerName        System.String  WIN81-ENT-01
```

While the XML document is still stored as a variable you could modify using the steps we showed earlier. When you are ready to save the data to a file use the Save() method.

```
$s.Save("c:\work\services.xml")
```

Creating native XML from scratch

The `ConvertTo-Xml` cmdlet is handy although it is a bit generic. If you truly need to create more meaningful XML you can build your own from scratch. Let's say you need to create an XML file for an external application with update (or hotfix) information. The application is expecting a per computer node with this hotfix information:

- update-id
- update-type
- install-date
- installed-by
- caption

First, we need data.

```
$data = Get-HotFix -ComputerName $env:computername |
Select-Object -Property Caption,InstalledOn,InstalledBy,HotfixID,Description
```

Because the XML node names won't always align with the PowerShell property names, and we want avoid a lot of hard coding, we'll create a "mapping" hashtable.

```
$map = [ordered]@{
    'update-id' = 'HotFixID'
    'update-type' = 'Description'
    'install-date' = 'InstalledOn'
    'install-by' = 'InstalledBy'
    caption = 'Caption'
}
```

You'll see how we use this in a bit. But we need an XML document.

```
[xml]$Doc = New-Object System.Xml.XmlDocument
```

While it is not an absolute requirement, you should create an XML declaration for the version and encoding and append it to the document.

```
$dec = $Doc.CreateXmlDeclaration("1.0","UTF-8",$null)
$doc.AppendChild($dec) | Out-Null
```

You'll see us starting to use `Out-Null` to suppress the XML output since we don't really want to see it.

Optionally, you might want to include a comment in your XML document. And for the sake of variety, we'll even create and append it all in one command.

```
$text = @"

Hotfix Inventory
$(Get-Date)

"@

$doc.AppendChild($doc.CreateComment($text)) | Out-Null
```

Now to the heart of the document. We want to have a computer node to show the computername. The process should be looking familiar by now.

```
$root = $doc.CreateNode("element","Computer",$null)
$name = $doc.CreateElement("Name")
$name.InnerText = $env:computername
$root.AppendChild($name) | Out-Null
```

We're creating, defining and appending to the parent. Where this gets interesting is where you have to create multiple, nested entries. You don't want to have to manually create everything one item at a time. Let PowerShell do the work for you.

We know we're going to need an outer node for Updates.

```
$hf = $doc.CreateNode("element","Updates",$null)
```

Within this node, we need to create an entry for each update. This is where the mapping table comes into play. We can loop through each update from $data and create an update entry. Then we can use a nested loop to go through the mapping hashtable to create the corresponding entries.

```
foreach ($item in $data) {
    $h = $doc.CreateNode("element","Update",$null)
    #create the entry values from the mapping hash table
    $map.GetEnumerator() | foreach {
      $e = $doc.CreateElement($_.Name)
      $e.innerText = $item.$($_.value)
      #append to Update
      $h.AppendChild($e) | Out-Null
    }
    #append the element
    $hf.AppendChild($h) | Out-Null
}
```

This performs the bulk of the work. All that remains is to append and save the file.

```
$root.AppendChild($hf) | Out-Null
$doc.AppendChild($root) | Out-Null
$doc.Save("c:\work\hotfix.xml")
```

The end result is something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Hotfix Inventory
06/04/2020 11:56:15
-->
<Computer>
  <Name>DESK01</Name>
  <Updates>
    <Update>
      <update-id>KB4552925</update-id>
      <update-type>Update</update-type>
      <install-date>06/02/2020 00:00:00</install-date>
      <install-by>NT AUTHORITY\SYSTEM</install-by>
      <caption>http://support.microsoft.com/?kbid=4552925</caption>
    </Update>
```

```
    <Update>
      <update-id>KB4537759</update-id>
      <update-type>Security Update</update-type>
      <install-date>05/30/2020 00:00:00</install-date>
      <install-by>NT AUTHORITY\SYSTEM</install-by>
      <caption>http://support.microsoft.com/?kbid=4537759</caption>
    </Update>
    <Update>
      <update-id>KB4557968</update-id>
      <update-type>Security Update</update-type>
      <install-date>05/30/2020 00:00:00</install-date>
      <install-by>NT AUTHORITY\SYSTEM</install-by>
      <caption>http://support.microsoft.com/?kbid=4557968</caption>
...
  </Updates>
</Computer>
```

You can find the complete demo script in the code downloads for this chapter.

Your Turn

We'd like to see what you can do with creating an XML-oriented tool. Let's say that your boss has decided she would like to start tracking disk usage using XML. Her plan is to take a usage snapshot on a weekly basis. She wants to maintain data for all reports and computers in a single XML file. (You might need to educate her about the use of a database!). On one hand you could easily accomplish this using `Export-Clixml` except that there is no parameter for appending.

Start Here

We'll help you out and give you a PowerShell expression that provides the necessary information using the local computer. You'll need to adjust to handle remote computers.

```
Get-CimInstance win32_logicaldisk -Filter "drivetype=3" |
Select-Object -property @{Name="Date";
Expression={(Get-Date).ToShortDateString()}},
PSComputername,DeviceID,Size,Freespace,@{Name="PercentFree";
Expression = {($_.freespace/$_.size)*100 -as [int]}}
```

Your Task

You will need to create a function to update the an xml file for multiple computers with the required information. You might consider creating several functions to handle the different aspects of this task.

Our Take

Given the manager's requirements we thought it might be best to create an XML file from scratch. We could have probably achieved similar results with `ConvertTo-Xml`. Our solution consists of several functions which could be packaged together as a module. You can find our functions in the chapter's code downloads.

The first function, `Get-DiskUsage`, uses `Get-CimInstance` to retrieve the disk information and writes an object to the pipeline. This function could be reused to send information to a CSV file, or anything else. The core function is `Update-DiskXml`. The function needs a parameter for the XML file and a group of computernames. If the XML file doesn't exist, we wrote another function, `New-DiskXML`, to create an empty XML document that meets our requirements.

The `Update-DiskXml` command calls `Get-DiskUsage` to get drive information and then creates snapshot information which it appends to the main document. Usage syntax looks like this:

```
Get-Content servers.txt | Update-DiskXml -path c:\work\diskhistory.xml
```

From a design perspective we *could* have written the update command to take pipeline input from `Get-DiskUsage`.

```
Get-Content servers.txt | Get-DiskUsage |
Update-DiskXml -path c:\work\diskhistory.xml
```

Ultimately the correct choice depends on how you think the consumer of your PowerShell tool will use it. But notice that we didn't build one function that did everything. This is a good reminder that in PowerShell toolmaking you build single purpose tools that do one thing but can work together in the PowerShell pipeline.

The other missing element is a command to process the XML and presenbt a formatted report. Or maybe this would be a controller script. It also might make sense to turn all of this into a module. We'll leave these tasks to you as additional learning opportunities.

Let's Review

Before you go, how about a quick test to see what you learned?

1. What is one of the major benefits of using XML?
2. If you only need to work with serialized data between PowerShell sessions, what are the best set of commands to use?
3. What command would you use to create native XML?
4. What is the easiest way to import a native XML document?

Review Answers

Here' how we would have answered the questions.

1. It is a great vehicle for storing hierarchical data.
2. `Import-Clixml` and `Export-Clixml`
3. `ConvertTo-Xml`
4. `[xml]$doc = Get-Content data.xml`

Working with JSON Data

As you build your PowerShell tools, you might have a need to store stuff in separate files. This might be configuration data for your command. Or perhaps you need to store the results in a file that will be used by another process or program. Perhaps even outside of PowerShell. This use to mean using things like INI or XML files. But over the last few years a new format has entered the world of PowerShell, JSON. Processing JSON instead of XML is often faster and typically can result in smaller file sizes. JSON files also tend to be a lot easier for humans to read compared to XML.

Now, JSON has been around for quite a while in the developer world as a data storage mechanism. You can learn all the gritty details at http://json.org[60]. But we'll keep this simple. A JSON file is a text file that serializes an object, much like XML. The object is wrapped in a set of curly braces and contains one or more sets of name/value pairs.

```
{
    "Name":  "bits",
    "DisplayName":  "Background Intelligent Transfer Service",
    "Status":  4
}
```

The name is essentially the property name and the value is self-evident. Because this is a text file, all of the values are strings. This will become important when you attempt to bring a JSON file into PowerShell.

JSON files can include multiple objects separated by commas and enclosed in a set of square brackets to indicate an array.

```
[
    {
        "Name":  "BITS",
        "DisplayName":  "Background Intelligent Transfer Service",
        "Status":  4
    },
    {
        "Name":  "Bluetooth Device Monitor",
        "DisplayName":  "Bluetooth Device Monitor",
        "Status":  4
    },
    {
```

[60] http://json.org

```
        "Name":  "Bluetooth OBEX Service",
        "DisplayName":  "Bluetooth OBEX Service",
        "Status":  4
    },
    {
        "Name":  "BrokerInfrastructure",
        "DisplayName":  "Background Tasks Infrastructure Service",
        "Status":  4
    },
    {
        "Name":  "Browser",
        "DisplayName":  "Computer Browser",
        "Status":  4
    },
    {
        "Name":  "BthHFSrv",
        "DisplayName":  "Bluetooth Handsfree Service",
        "Status":  1
    },
    {
        "Name":  "bthserv",
        "DisplayName":  "Bluetooth Support Service",
        "Status":  4
    }
]
```

Finally, the JSON format supports nested objects.

```
{
    "Name":  "bits",
    "DisplayName":  "Background Intelligent Transfer Service",
    "Status":  4,
    "RequiredServices":  [
                    {
                        "CanPauseAndContinue":  false,
                        "CanShutdown":  false,
                        "CanStop":  false,
                        "DisplayName":  "Remote Procedure Call (RPC)",
                        "DependentServices":  null,
                        "MachineName":  ".",
                        "ServiceName":  "RpcSs",
                        "ServicesDependedOn":  "DcomLaunch RpcEptMapper",
                        "ServiceHandle":  null,
```

```
                    "Status":  4,
                    "ServiceType":  32,
                    "StartType":  2,
                    "Site":  null,
                    "Container":  null
                },
                {
                    "CanPauseAndContinue":  false,
                    "CanShutdown":  false,
                    "CanStop":  true,
                    "DisplayName":  "COM+ Event System",
                    "DependentServices":  "igfxCUIService1.0.0.0
                     COMSysApp SENS BITS",
                    "MachineName":  ".",
                    "ServiceName":  "EventSystem",
                    "ServicesDependedOn":  "rpcss",
                    "ServiceHandle":  null,
                    "Status":  4,
                    "ServiceType":  32,
                    "StartType":  2,
                    "Site":  null,
                    "Container":  null
                }
                    ]
}
```


We showed you these examples so you would know what a JSON file looks like, but you should never have to create one by hand. Instead you can call upon the PowerShell JSON cmdlets, `ConvertTo-Json` and `ConvertFrom-Json`.

Converting to JSON

You can take the output from any PowerShell expression that writes to the pipeline and turn it into JSON.

```
Get-CimInstance win32_computersystem | ConvertTo-Json
```

If you try that command you'll notice right away that you don't get a file. The cmdlet is doing exactly what it is designed to do, convert objects to a json format. To save the results to a file you can pipe to `Out-File` or `Set-Content`.

```
Get-CimInstance win32_computersystem | ConvertTo-Json |
Out-File wmics.json
Get-CimInstance win32_computersystem | ConvertTo-Json |
Set-Content .\wmics2.json
```

By default, you'll end up with easy to read and formatted JSON. However, there is also an option to compress the converted json.

```
Get-CimInstance win32_computersystem | ConvertTo-Json -compress
```

You'll notice that this removes all the spaces and indentations. The resulting file will be smaller but still valid JSON. We can't think of any reason to compress unless you are creating very large files that you intend to copy between systems. Of course, you can also keep things manageable by only converting what you actually need.

```
Get-CimInstance win32_computersystem -computername $env:computername |
Select-Object -property PSComputername,Manufacturer,
@{Name="MemoryGB";Expression={$_.totalPhysicalmemory/1GB -as [int]}},
Number* | ConvertTo-Json
```

This should create output like this:

```
{
    "PSComputerName":  "CLIENT01",
    "Manufacturer":  "LENOVO",
    "MemoryGB":  8,
    "NumberOfLogicalProcessors":  4,
    "NumberOfProcessors":  1
}
```

We're assuming you know what you'll do with the final file and will plan accordingly.

One tip we'll point out is that if you want to create a json file, perhaps to hold configuration data or something similar, don't try to manually create the file. Instead, "objectify" your data in PowerShell and then convert to JSON.

```
[pscustomobject]@{
  Path = "C:\Scripts"
  LastModified = 6/1/2020"
  Count = 20
  Types = @(".ps1","psm1","psd1","json","xml")
} | ConvertTo-Json
```

You don't have to muck about trying to get the formatting right. Let the cmdlet do the work for you.

```
{
    "Path":  "C:\\Scripts",
    "LastModified":  "6/1/2020",
    "Count":  20,
    "Types":  [
                  ".ps1",
                  "psm1",
                  "psd1",
                  "json",
                  "xml"
              ]
}
```

We've already mentioned that the JSON format is essentially one long string. The format does not have any mechanism for comments or metadata like you can include in an XML file. That's not to say you can't incorporate such a feature but you'll have to design your own implementation. Using our example above, you could try something like this:

```
[pscustomobject]@{
  Created = (Get-Date)
  Comment = "config data for script tool"
},
[pscustomobject]@{
  Path = "C:\Scripts"
  LastModified = "6/1/2020"
  Count = 20
  Types = @(".ps1","psm1","psd1","json","xml")
} | ConvertTo-Json
```

You'll end up with this JSON:

```
[
    {
        "Created":  {
                        "value":  "\/Date(1591287561241)\/",
                        "DisplayHint":  2,
                        "DateTime":  "Thursday, June 4, 2020 12:19:21 PM"
                    },
        "Comment":  "config data for script tool"
    },
    {
        "Path":  "C:\\Scripts",
        "LastModified":  "6/1/2020",
        "Count":  20,
        "Types":  [
                      ".ps1",
                      "psm1",
                      "psd1",
                      "json",
                      "xml"
                  ]
    }
]
```

As you can see, PowerShell transforms the date object into something a bit more complicated in JSON. Knowing that everything is going to be a string you might modify the first part:

```
[pscustomobject]@{
  Created = (Get-Date).ToString()
  Comment = "config data for script tool"
},
```

Now the JSON is a bit easier to read.

```
{
    "Created": "6/4/2020 12:20:33 PM",
    "Comment":  "config data for script tool"
}
```

Converting from JSON

By now you can probably guess the name of the cmdlet that turns JSON content into something you can use in PowerShell: `ConvertFrom-Json`. If you read the help for the cmdlet, which you should by

the way, you'll recognize that the cmdlet doesn't use a file. Rather, you have to get the json content and then convert it.

We have a json file (which we've included in the code samples for this chapter) with entries like this:

```
{
    "Name":  "wuauserv",
    "DisplayName":  "Windows Update",
    "Status":  1,
    "MachineName":  "chi-dc04",
    "Audit":  "06/04/20"
},
```

To bring this into PowerShell we'll run this command:

```
PS C:\> $in = Get-Content c:\work\audit.json | ConvertFrom-Json
```

Nothing too surprising here. You have to get the content before you can convert it. The conversion will create a custom object.

```
PS C:\> $in | Get-Member


   TypeName: System.Management.Automation.PSCustomObject

Name        MemberType   Definition
----        ----------   ----------
Equals      Method       bool Equals(System.Object obj)
GetHashCode Method       int GetHashCode()
GetType     Method       type GetType()
ToString    Method       string ToString()
Audit       NoteProperty string Audit=06/01/20
DisplayName NoteProperty string DisplayName=Microsoft Monitoring Agent
                                Audit Forwarding
MachineName NoteProperty string MachineName=chi-dc04
Name        NoteProperty string Name=AdtAgent
Status      NoteProperty int Status=1
```

As you can see everything is treated as a string which may not be an issue for you.

```
PS C:\> $in[0..2]


Name        : AdtAgent
DisplayName : Microsoft Monitoring Agent Audit Forwarding
Status      : 1
MachineName : chi-dc04
Audit       : 06/04/20

Name        : ADWS
DisplayName : Active Directory Web Services
Status      : 4
MachineName : chi-dc04
Audit       : 06/04/20

Name        : AeLookupSvc
DisplayName : Application Experience
Status      : 1
MachineName : chi-dc04
Audit       : 06/04/20
```

One option to make this more accurate might be to "reformat" using `Select-Object`.

```
PS C:\> $in[0..2] | Select-Object -property Name,Displayname,
@{Name="Status";
Expression = { $_.Status -as [System.ServiceProcess.ServiceControllerStatus]}},
@{Name="Audit";Expression= { $_.Audit -as [datetime]}},
@{Name="Computername";Expression = {$_.Machinename}}

Name         : AdtAgent
DisplayName  : Microsoft Monitoring Agent Audit Forwarding
Status       : Stopped
Audit        : 6/4/2020 12:00:00 AM
Computername : chi-dc04

Name         : ADWS
DisplayName  : Active Directory Web Services
Status       : Running
Audit        : 6/4/2020 12:00:00 AM
Computername : chi-dc04

Name         : AeLookupSvc
DisplayName  : Application Experience
```

```
Status       : Stopped
Audit        : 6/4/2020 12:00:00 AM
Computername : chi-dc04
```

With this approach the objects are richer and can be properly sorted, filtered or whatever.

Now, you may be thinking, "why not do this reformatting during the conversion?" That's an excellent an idea. Your initial idea is to use an expression like this:

```
Get-Content audit.json |
ConvertFrom-Json | Select Name,Displayname,
@{Name="Status";Expression = { $_.Status -as [System.ServiceProcess.ServiceControlle\
rStatus]}},
@{Name="Audit";Expression= { $_.Audit -as [datetime]}},
@{Name="Computername";Expression = {$_.Machinename}}
```

Only to realize you don't get the expected results.

```
Name         :
Displayname  :
Status       : ContinuePending
Audit        :
Computername : {chi-dc04, chi-dc04, chi-dc04, chi-dc04...}
```

This is because the `ConvertFrom-Json` cmdlet writes a single object to the pipeline. If you use `Get-Member` you'll see that it is a `System.Object[]`. The workaround is to use `ForEach-Object`.

```
Get-Content .\audit.json |
ConvertFrom-Json |
ForEach-Object { $_ | Select-Object -Property Name,Displayname,
@{Name="Status";
Expression = { $_.Status -as [System.ServiceProcess.ServiceControllerStatus]}},
@{Name="Audit";Expression= { $_.Audit -as [datetime]}},
@{Name="Computername";Expression = {$_.Machinename}}
}
```

For experienced and advanced readers, you could also insert a custom type name into the converted objects and then use custom type and format extensions. The bottom line when it comes to working with JSON is you still want to think about working with objects in the pipeline but you have to know what that data will look like and how you will use it.

In PowerShell 7, this annoying problem has been fixed and you don't need to use this `ForEach-Object` hack. In fact, the Json cmdlets have a few more features in PowerShell 7. We are always harping about reading the help, even for commands you think you know. PowerShell is still a growing technology. Perhaps even more so now with PowerShell 7. You absolutely need to look at help and examples.

Your Turn

Let's see what you've picked up in this chapter and see if you can build a simple PowerShell tool that utilizes JSON files.

Start Here

One of the great benefits of PowerShell is that you can use it with just about anything. So we are going to put you in a company that has an external process which processes widgets and generates a JSON summary report. Each file will have JSON like this:

```
{
    "JobID":  214699,
    "Items processed":  78,
    "Errors":  2,
    "Warnings":  0,
    "RunDate":  "6/2/2020 9:53:45 PM"
}
```

All of the files are created in a single folder. We've provided some files in the code folder under \codeChapters\working-with-jsonSampleData. Your manager has asked you to create a PowerShell tool which will process these files and generate a summary report.

Your Task

Knowing how fickle your manager is, your tool should just write a summary object to the pipeline. This way you can run your command and then pipe it to anything else to generate a specific type of report such as a text file or HTML, although that's not the real goal. Instead, you should create a stand-alone PowerShell script that will process all of the JSON files in the SampleData directory and write a summary object to the pipeline with this information.

- Number of files processed
- Total number of items processed
- Average number of items processed

- Total number of Errors
- Average number of Errors
- Total number of Warnings
- Average number of Warnings
- StartDate (the earliest run date value)
- EndDate (the last run date value)

Our Take

You can find our complete solution in the downloadable code samples, under this book's folder in the Chapters subfolder. Below is a stripped down version.

Summary Report

```
[cmdletbinding()]
Param(
    [Parameter(
        Position = 0,
        Mandatory,
        HelpMessage = "Enter the path with the json test data"
    )]
    [ValidateNotNullorEmpty()]
    [string]$Path
)

Write-Verbose "Starting $($MyInvocation.MyCommand)"

Write-Verbose "Processing files from $Path"

$files = Get-ChildItem -Path $path -Filter *.dat

Write-Verbose "Found $($files.count) files."
$data = foreach ($file in $files) {
    Write-Verbose "Converting $($file.name)"
    Get-Content -Path $file.fullname |
    ConvertFrom-Json |
    Select-Object -Property Errors,Warnings,
    @{Name = "Date"; Expression = {$_.RunDate -as [datetime]}},
    @{Name = "ItemCount"; Expression = {$_.'Items processed'}}
}

#sort the data to get the first and last dates
$sorted = $data | Sort-Object Date
```

```
$first = $sorted[0].Date
$last = $sorted[-1].Date

Write-Verbose "Measuring data"
# The $stats variable will be an array of measurements for each property
$stats = $data | Measure-Object errors, warnings, ItemCount -sum -average

Write-Verbose "Creating summary result"
[PSCustomObject]@{
    NumberFiles           = $data.count
    TotalItemsProcessed   = $stats[2].sum
    AverageItemsProcessed = $stats[2].Average
    TotalErrors           = $stats[0].sum
    AverageErrors         = $stats[0].average
    TotalWarnings         = $stats[1].sum
    AverageWarnings       = $stats[1].Average
    StartDate             = $first
    EndDate               = $last
}
```

Naturally the trickiest part is converting the JSON files. Each file has a single entry which means each file has to be converted separately. And we also need to converted objects to be "good" PowerShell which means no spaces in property names and they should be properly typed. That's why we use `Select-Object` and custom hash tables to rename the JSON property that contains a space and treat the RunDate as a `[DateTime]` object.

The rest of the script is merely using `Measure-Object` to calculate the necessary values and write a custom object to the pipeline.

```
PS C:\> c:\work\SampleReport.ps1 -path c:\work\SampleData

NumberFiles           : 20
TotalItemsProcessed   : 1151
AverageItemsProcessed : 57.55
TotalErrors           : 21
AverageErrors         : 1.05
TotalWarnings         : 50
AverageWarnings       : 2.5
StartDate             : 5/3/2020 1:55:45 AM
EndDate               : 6/3/2020 6:06:45 AM
```

Let's Review

We readily admit that working with JSON files will be limited to some very specific use cases. But let's make sure you still understand the basics.

1. The JSON format is similar to what other serialization format?
2. The `ConvertTo-Json` cmdlet will also create a file for you. True or False?
3. PowerShell will automatically determine property types when converting from JSON. True or False.
4. What are some of the benefits of using JSON instead of XML?

Review Answers

Here are some likely answers.

1. XML
2. False.
3. False. If you need properties to be other than `[String]` you will have to add relevant PowerShell code.
4. JSON tends to produce smaller files, can be faster and is easier to read.

Working With CSV Data

For the sake of completeness, it only made sense to do a chapter on working with CSV data. If there was one data format that comes up all the time, it is this. On one hand it is a pretty simple format. But looks can be deceiving which is probably why we see a lot of questions about it.

I Want to Script Microsoft Excel

No you don't. Probably not really. What you really want to do is work with CSV data. Sure, use Microsoft Excel to get you started by exporting a CSV file. Or get your CSV file from PowerShell and open it in Excel for anything else you want to do. But for the love of Jeffrey Snover don't try to write PowerShell to update cells in a worksheet. Yes, you can but it is interminable. After spending a few minutes counseling a wayward sole, we usually find that they really just need to work with a CSV file.

Know Your Data

When it comes to working with any data type, you have to know what you are exporting or importing. A CSV file is a plain text file with values separated by commas. Typically, the first line in the file is a header.

```
Name,Size,Color
john,12,green
george,3,yellow
paul,1,purple
ringo,7,white
```

The most important thing to know about a CSV file is that it is considered a "flat" file. All you can have is collection of values. You can't have a column with a service object. That object has its own hierarchy of property values. It would be impossible to add it to the CSV sample above. For rich object types, you need to use a hierarchical format such as XML or JSON.

If you try this:

```
Get-Service win* | Export-CSV win.csv
Import-CSV win.csv
```

You won't get the result you expect. You'll see values like `"System.ServiceProcess.ServiceController[]"` which indicates a nested object. You can use the PowerShell CSV cmdlets, but you have to know what you are exporting.

```
Get-Service win* -ComputerName SRV1 |
Select-Object -property Name,Displayname,Status,StartType,
@{Name="Computername";Expression={$_.machinename}} |
Export-CSV win-srv1.csv
```

We're going to trust that you'll read the full help and examples for all the cmdlets we cover in this chapter.

Looking at the raw CSV file, we now have a simple collection of flat values.

```
#TYPE Selected.System.ServiceProcess.ServiceController
"Name","DisplayName","Status","StartType","Computername"
"WinDefend","Windows Defender Service","Running","Automatic","SRV1"
"WinHttpAutoProxySvc","WinHTTP Web Proxy Auto-Discovery Service","Running","Manual",\
"SRV1"
"Winmgmt","Windows Management Instrumentation","Running","Automatic","SRV1"
"WinRM","Windows Remote Management (WS-Management)","Running","Automatic","SRV1"
```

Knowing your data also applies to your export. Assuming you've sent it to a file with `Export-CSV` how are you going to use it? Are you going to import back into PowerShell? You don't have to do anything. But if you plan on using the data in a non-PowerShell application, such as importing it into a SQL table, then you might want to get rid of that `#TYPE` header.

```
... | Export-CSV file.csv -NoTypeInformation
```

Custom Headers

Here's a little trick that can make your life easier. Often we get CSV files from external sources. We may have no control over how they are created. But we still want to incorporate them into PowerShell. Here's a sample CSV file that lacks a header line.

```
Dom1,DC,DB9097,3/16/18
Srv1,Member,CC2365,1/22/19
Srv2,Member,CC2369,1/22/19
Win10,Client,WX3487,12/4/18
```

Remember our comment that you need to know your data? We know that the first column is the computername, the second is type, the third is an asset tag and the last column is the acquisition date. We can tell PowerShell to import the data using a custom header.

```
$head = "Computername","Type","AssetTag","Acquired"
Import-Csv .\data-nohead.csv -Header $head
```

Which gives us this output:

```
Computername Type   AssetTag Acquired
------------ ----   -------- --------
Dom1         DC     DB9097   3/16/18
Srv1         Member CC2365   1/22/19
Srv2         Member CC2369   1/22/19
Win10        Client WX3487   12/4/18
```

Now we have objects in the pipeline with the right property names.

If your CSV file does have a header but you don't want to use it, you have a few options. We have a variation of the above file that has a header of `Server,Class,Asset,Inventory`. We want to use the data as a source of computer names that we can pipe to `Get-Service`.

When you read the help and look at the `Computername` parameter, you'll see that it accepts pipeline input by property name.

```
    Required?                    false
    Position?                    named
    Default value                None
    Accept pipeline input?       True (ByPropertyName)
    Accept wildcard characters?  false
```

We need to make sure that the objects coming from `Import-CSV` include a property called `Computername`. But the CSV heading is `Server`. No problem. We'll rename it on the fly.

```
Import-Csv .\data.csv |
Select-Object @{Name="computername";Expression={$_.server}} |
Get-Service Bits | Select-Object Machinename,Name,Status
```

Or you can use the custom header but you need to take an extra step to strip it off. In this situation you have to parse the CSV file and then convert the results from CSV.

```
Get-Content .\data.csv | Select-Object -Skip 1 |
ConvertFrom-CSV -Header $head |
Get-Service Bits |
Select-Object Machinename,Name,Status
```

Importing Gotchas

One huge gotcha when working with CSV files is that everything is treated as a string. You might try this:

```
PS C:\> Import-Csv .\data.csv | Sort-Object Inventory

Server Class  Asset  Inventory
------ -----  -----  ---------
Srv1   Member CC2365 1/22/19
Srv2   Member CC2369 1/22/19
Win10  Client WX3487 12/4/18
Dom1   DC     DB9097 3/16/18
```

But that didn't work. If you use `Get-Member` you'll see what we're talking about.

```
PS C:\> import-csv .\data.csv | Get-member


   TypeName: System.Management.Automation.PSCustomObject

Name        MemberType   Definition
----        ----------   ----------
Equals      Method       bool Equals(System.Object obj)
GetHashCode Method       int GetHashCode()
GetType     Method       type GetType()
ToString    Method       string ToString()
Asset       NoteProperty string Asset=DB9097
Class       NoteProperty string Class=DC
Inventory   NoteProperty string Inventory=3/16/18
Server      NoteProperty string Server=Dom1
```

This comes back to the mantra "know your data". One solution is to re-define your properties using `Select-Object`.

```
Import-Csv .\data.csv |
Select-Object @{Name="ComputerName";Expression={$_.server}},
Asset,Class,
@{Name="InventoryDate";Expression = {$_.Inventory -as [datetime]}} |
Sort-Object InventoryDate
```

Now we get the results we're expecting.

```
ComputerName Asset  Class  InventoryDate
------------ -----  -----  -------------
Dom1         DB9097 DC     3/16/2018 12:00:00 AM
Win10        WX3487 Client 12/4/2018 12:00:00 AM
Srv1         CC2365 Member 1/22/2019 12:00:00 AM
Srv2         CC2369 Member 1/22/2019 12:00:00 AM
```

Although, for sorting purposes we could have done this:

```
Import-Csv .\data.csv | Sort-Object {$_.Inventory -as [datetime]}
```

Regardless, we had to take steps to turn the string value into the proper type.

Your Turn

You should spend a little time with CSV files to make sure you understand the concepts and techniques.

Start Here

Export a group of files from a directory of your choice to a CSV file selecting these properties:

- name
- extension
- fullname (exported as Path)
- length (exported as Size)
- CreationDate (exported as Created)
- LastWriteTime (exported as Modified)

Your Task

Write a PowerShell expression to export the files to a CSV file. Create a file that you could use outside of PowerShell. Then write code to import the CSV file and sort by length in descending order. The timestamp properties should also be treated as [datetime] values.

Our Take

We used a relatively simple one-line command to create the CSV file.

Exporting Data

```
Get-ChildItem c:\work -file |
Select-Object Name,Extension,
@{Name="Path";Expression = {$_.fullname}},
@{Name="Size";Expression = {$_.length}},
@{Name="Created";Expression = {$_.Creationtime}},
@{Name="Modified";Expression = {$_.LastWriteTime}} |
Export-CSV -Path .\files.csv -NoTypeInformation
```

Our results file should be in the downloads.

To restore the data, we did something a little bit different. We wrote a function to create a new type of object from the CSV file. Again, "know your data".

New-FileData

```
Function New-FileData {
[cmdletbinding()]
Param(
[Parameter(Mandatory,ValueFromPipeline)]
[object[]]$InputObject
)

Begin {}
Process {
    [PSCustomObject]@{
        PSTypeName = "FileData"
        Name = $_.Name
        Extension = $_.Extension
        Path = $_.Path
        Size = $_.Size -as [int]
        CreationTime = $_.Created -as [datetime]
        LastWritetime = $_.Modified -as [datetime]
    }
```

```
}
End {}
}
```

Now we can import the data back into PowerShell.

```
$filedata = Import-CSV .\files.csv | New-FileData
```

We can verify that properties are of the correct type.

```
PS C:\> $filedata | Get-Member


   TypeName: FileData

Name          MemberType   Definition
----          ----------   ----------
Equals        Method       bool Equals(System.Object obj)
GetHashCode   Method       int GetHashCode()
GetType       Method       type GetType()
ToString      Method       string ToString()
CreationTime  NoteProperty datetime CreationTime=10/10/2019 10:21:17 AM
Extension     NoteProperty string Extension=.vhdx
LastWritetime NoteProperty datetime LastWritetime=10/10/2019 10:21:17 AM
Name          NoteProperty string Name=dummy.vhdx
Path          NoteProperty string Path=C:\work\dummy.vhdx
Size          NoteProperty int Size=4194304
```

And the sort works.

```
PS C:\> $fileData | Sort-Object size -Descending | Select-Object Name,Size

Name                              Size
----                              ----
dummy.vhdx                     4194304
handbrake_2019-04-03-1336.txt   780500
psconfcover.jpg                 368578
AZUser.xml                      206358
handbrake_2019-04-03-1337.txt   116054
get-psreleaseasset-macos.png    105876
...
```

Let's Review

1. What is best way to describe data stored in a CSV file?
2. How is data treated in a CSV file?
3. What parameter would you use with `Export-CSV` to suppress the type information?
4. We didn't explicitly cover it, but since we know you read the help, what parameter would you use to import a CSV file that used a semi-colon instead of a comma?

Review Answers

1. Flat.
2. Everything is treated as a string
3. `-NoTypeInformation`
4. `-Delimiter ';'`

Part 5: Seriously Advanced Toolmaking

In this Part of the book, we'll dive into some deep, "extra" topics. These are all things we're pretty sure you should know, but that you might not use right away, especially if you are an apprentice toolmaker. This Part isn't constructed in a storyline, so you can just pick and choose the bits you think you'll need or you find interesting.

Tools for Toolmaking

If you look at any master craftsman from carpenter to chef and one thing they have in common is their toolkit. A carpenter is going to invest in high quality tools that help them do their job such as hammers and power tools. A chef will often invest hundreds of dollars for a single knife, but it is a knife they will use every day and it makes them more productive. As a PowerShell toolmaker, you need to take the same approach. Sure, you could build all the PowerShell tools you need with nothing more than Notepad but you will be **far** from efficient and you'll dread your work. One feature of high quality tools is that they often make the job easier and more fun.

So, as the commercial line goes, "What's in your toolbox?" In this chapter we want to share some suggestions for items you might consider adding. Don't take any of these as absolute recommendations. Just because something works for us doesn't mean it works for you. And obviously we don't know about every tool in the PowerShell universe, but we need to start somewhere.

Some of the items we're going to discuss are free and others are commercial products. Don't assume anything. A free tool might be a buggy piece of junk while a commercial tool might save you hundreds of hours and pay for itself in short order. Most commercial tools have a trial period which we encourage you to take advantage of. Only then can you determine if the expense is justified.

Editors

The first tool you will need is an editor. You want something that accelerates your toolmaking. For us, these are the critical elements we look for in an editor:

- Intellisense or some sort of command/parameter completion
- At least basic debugging features
- Color coded syntax
- line numbering
- Support for some type of snippet
- The ability to execute or evaluate code within the editor

You may also have to evaluate what other tasks your editor might need to accomplish based on your job duties. Do you need to write Python scripts? Do you often need to create graphical tools with WPF or WinForms? Do you need to create C# utilities? Do you need to support Windows only or are you a cross-platform kind of person?

Let's look at a few options you might want to consider.

PowerShell ISE

The PowerShell ISE is an option you don't really have to think about. On client operating systems you just have it. One of the reasons the ISE was developed was so that you wouldn't have to use Notepad to write your PowerShell scripts and tools.

The PowerShell ISE offers all of the features we listed above. The layout and display are customizable. You can run the entire script or selected pieces of code directly in the editor and its integrated shell. You can run multiple and distinct PowerShell sessions through its tab interface as well as create sessions to remote computers. We also like that the PowerShell ISE has its own object model which means you can create your own ISE-tools and shortcuts. You will also find a number of plugins and extensions that have been developed over the years such as ISE Steroids.

At the very least, you should learn how to take advantage of the PowerShell ISE to create better code faster. The ISE is already installed and comes with a free copy of Windows!

Now for the buzz kill. With the release of PowerShell 7, the PowerShell ISE should be considered deprecated. It isn't going away anytime soon. But it also isn't getting any updates or fixes, aside from critical security problems that might arise. What you see is what you get. If you've already invested in mastering the ISE you don't have to give it up. Just know that there's nothing new on the horizon for it. If you are just getting started and need to learn how to use an editor, you might as well jump in with VS Code.

Visual Studio Code

As useful as the PowerShell ISE might be, it only runs on Windows and PowerShell is extending in the enterprise to Linux and Mac. To meet this need, Microsoft has been working on a concept of editor services. We're not interested in APIs and architecture drawings. What can we use? The answer is a free download from Microsoft called Visual Studio Code, also known as VS Code.

You can download the latest version from https://code.visualstudio.com/download[61] for Windows, a few Linux distros and MacOS. After it is installed, the application can auto-update as needed.

VS Code is **the** Microsoft source editor going forward. Microsoft Technical Fellow Jeffrey Snover has publicly stated that their investment is in VS Code. VS Code is designed to be a cross-platform product and supports multiple languages. For your purposes the first thing you'll need to do is install the PowerShell extension. In VS Code type `Ctrl+Shift+P` and start typing "extensions". Select "Install Extensions" and search for "PowerShell". After you install the extension, you'll get syntax highlighting and Intellisense-like completion in PowerShell script files. You can also run selected lines of code with F8 just like the ISE.

[61] https://code.visualstudio.com/download

VS Code can be customized, has good debugging features and a nice git integration. For many IT Pros comfortably familiar with the ISE, VS Code still has a few shortcomings but Microsoft is aware of them and is actively working on them. Right now they are releasing monthly updates.

Expect a steeper learning curve than with the PowerShell ISE, but that is because VS Code is a much richer and feature-loaded application.

Visual Studio

So far we've been looking at editors with an eye towards scripting. If you need something more developer oriented clearly Visual Studio is the way to go. This is a very rich product that should cover just about any development or need you might have. As such, the installation can be hefty. Your organization may already have licenses for Visual Studio or you can download the free Visual Studio Community Edition.

You can find information about downloading all the Visual Studio products at https://www.visualstudio.com/downloads/[62].

One reason you might want some flavor of Visual Studio is support for WPF. Visual Studio makes it much easier to design the graphical interface for your PowerShell tool. You can take the XAML and build your tool around it. We covered this in the WPF chapter.

There is an excellent PowerShell extension for Visual Studio from fellow MVP Adam Driscoll. According to the extension's description it offers these features:

- Edit, run and debug PowerShell scripts locally and remotely using the Visual Studio debugger
- Create projects for PowerShell scripts and modules
- Leverage Visual Studioâ€™s locals, watch, call stack for your scripts and modules
- Use the PowerShell interactive REPL window to execute PowerShell scripts and command right from Visual Studio
- Automated testing support using Pester

Finally, if you are thinking about gliding from PowerShell into C#, you'll want something like Visual Studio. This is certainly the most complex tool we've mentioned but if it fulfills a need it is worth your investment.

3rd Party

To properly fill out your toolbox you might need to spend a few bucks, or maybe more than a few bucks, for something that meets a need or increases your productivity. If you have to spend some money for something but it cuts your development time that might be a worthwhile investment.

[62] https://www.visualstudio.com/downloads/

ISESteroids

One of the more popular add-ons for the PowerShell ISE is ISESteroids developed by MVP Dr. Tobias Weltner. The product ships as a PowerShell module. You can find it in the PowerShell gallery.

```
Find-Module ISESteroids
```

The module adds a number of features and tools to the ISE that make it easier to develop, troubleshoot and debug your PowerShell projects. You can use the product free for 10 days. After that you will need to acquire a license. There are a variety of license options and for many IT Pros the cost has been more than offset by increased proficiency.

You can learn more at http://www.powertheshell.com/isesteroids/[63] including licensing details.

PowerShell Studio

Another popular commercial tool is PowerShell Studio from SAPIEN. They have been making scripting tools for long time going all the way back to the days of PrimalScript. PowerShell Studio is a very feature rich application that makes it easier to create PowerShell scripts, tools and modules.

One of the more compelling features is the ability to easily create WinForms-based PowerShell tools through a graphical drag-and-drop editor. This greatly simplifies the tedious process of developing the PowerShell code to display the graphical elements.

Another popular feature is the ability to "package" your PowerShell script as an executable.

PowerShell Studio is available in both 32 and 64 bit flavors and has a 45 day trial period. Learn more at https://www.sapien.com/software/powershell_studio[64].

PowerShell Community Modules

You might also find a number of useful tools from the PowerShell community. Many of these have been published to the PowerShell Gallery.

```
Find-Module -tag scripting
```

Jeff has published a number of modules to the PowerShell Gallery. Many of them designed to make scripting easier. Take a look at his GitHub repository[65] for more information.

And of course, you should consider using the PowerShell Script Analyzer if you aren't using VS Code, where it is baked in.

[63] http://www.powertheshell.com/isesteroids/
[64] https://www.sapien.com/software/powershell_studio
[65] https://github.com/jdhitsolutions

```
Install-module PSScriptAnalyzer
```

This module includes commands that will analyze your code and alert you to potential problems or those bits of code that run counter to currently accepted best practices. This functionality is built into Visual Studio Code where the analysis runs in real-time as you're writing your code.

Books, Blogs and Buzz

Lastly, we are always asked about books and other learning material. Educational material should definitely be considered tools and you should be constantly adding to your bookshelf.

We've authored many, many books over our career. You can find current books from http://manning.com[66] and http://leanpub.com[67]. On LeanPub you can also find a number of ebooks you can get for free or a very modest price.

Both of us blog, http://donjones.com[68] and https://jdhitsolutions.com[69]. Jeff has also contributed for years to the Petri IT Knowledge base. Checkout https://petri.com/category/PowerShell[70] for the latest. You should also check out https://mcpmag.com/pages/topic-pages/powershell.aspx[71] which has a good PowerShell section authored by other MVPs such as Adam Bertram and Boe Prox.

The other web source we recommend is Powershell.org[72]. Yes, we are actively involved in the organization behind it, but don't let that sway you. The forums on the site are very lively and actively monitored. If you need help getting over a roadblock in your tool development, this is the place to go for answers.

Lastly, you may not be into this sort of thing, but social media is a fantastic source of PowerShell information. We're both active on Twitter (@concentrateddon and @jeffhicks). We strongly recommend you setup a filter in your Twitter client for #PowerShell. This is the best way to keep up with what is new in the PowerShell world. There are also PowerShell-related groups in Facebook.

Recommendations

Where does all of this leave you? First off, if you want to consider yourself a professional PowerShell toolmaker you will need to invest time and perhaps money in tools for your toolbox. This might mean software. It might mean coughing up a few bucks for a couple books or some training videos. PowerShell and its related technologies like DSC are constantly evolving and you need to stay with the curve.

[66] http://manning.com
[67] http://leanpub.com
[68] http://donjones.com
[69] https://jdhitsolutions.com
[70] https://petri.com/category/PowerShell
[71] https://mcpmag.com/pages/topic-pages/powershell.aspx
[72] https://PowerShell.org

You should be using at least the PowerShell ISE for your development efforts with an eye towards moving to VS Code. That's where Microsoft is making the investment so you probably should as well.

Finally, your most valuable tool is curiosity. You have to be willing and interested to read about PowerShell, how people are using it and what they are bringing to the party. The more you learn, the better toolmaker you become and the more rewarding your career will become.

Measuring Tool Performance

We PowerShell geeks will often get into late-night, at-the-pub arguments about which bits of PowerShell perform best under certain circumstances. You'll hear arguments like, "the `ForEach-Object` cmdlet is slower because its script block has to be parsed each time" or, "storing all those objects in a variable will make everything take longer because of how arrays are managed." At the end of the day, if performance is *important* to you, this is the chapter for you.

Is Performance Important

Well, maybe. *Why* is performance important to you? Look, if you've written a command that will have to reboot a dozen computers, then we're going to be splitting hairs all night about which way is faster or slower. It won't matter. But if you're writing code that needs to manipulate *thousands* of objects, or *tens of thousands* or more, then a minute performance gain per-object will add up quickly. The point is, before you sweat this stuff, know that tweaking PowerShell for millisecond performance gains isn't useful unless there are a *lot* of milliseconds to be saved.

Measure What's Important

But if performance *is* important, then you need to *measure it*. Forget every possible argument for or against any given technique, and *measure it*. And, as you measure, make sure you're measuring to the scale that your command will eventually run. That is, don't test a command with five objects when the plan is to run against five hundred thousand. Pressures like memory, disk I/O, network, and CPU won't interact in meaningful ways at small scale, and so small-scale measurements won't prove out as you scale up your workload.

Think of it this way: just because a one-lane road can carry 100 cars an hour, doesn't mean a 4-lane road can carry 400 an hour. It's a different situation, with different dynamics. So *measure* against the workload you plan to run.

You'll perform that measurement using the `Measure-Command` cmdlet. Feed it your command, script, pipeline, or whatever, and it'll run it - and spit out how long it took it to complete. Take this short script as an example (this is test.ps1 in the sample files):

Test.ps1

```
Write-Host "Round 1" -ForegroundColor Green
Measure-Command -Expression {
    Get-Service |
    ForEach-Object { $_.Name }
}

Write-Host "Round 2" -ForegroundColor Yellow
Measure-Command -Expression {
    Get-Service |
    Select-Object Name
}

Write-Host "Round 3" -ForegroundColor Cyan
Measure-Command -Expression {
    ForEach ($service in (Get-Service)) {
        $service.name
    }
}
```

This basically does the same thing in different ways. Let's run that to see what happens:

```
Round 1

Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 148
Ticks             : 1486572
TotalDays         : 1.72056944444444E-06
TotalHours        : 4.12936666666667E-05
TotalMinutes      : 0.00247762
TotalSeconds      : 0.1486572
TotalMilliseconds : 148.6572

Round 2
Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 37
```

```
Ticks             : 379826
TotalDays         : 4.39613425925926E-07
TotalHours        : 1.05507222222222E-05
TotalMinutes      : 0.000633043333333333
TotalSeconds      : 0.0379826
TotalMilliseconds : 37.9826

Round 3
Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 38
Ticks             : 389199
TotalDays         : 4.50461805555556E-07
TotalHours        : 1.08110833333333E-05
TotalMinutes      : 0.000648665
TotalSeconds      : 0.0389199
TotalMilliseconds : 38.9199
```

There's a significant penalty, time-wise, for the first method, while the second two are almost tied. Neat, right?

Be Careful!

The thing to remember is that whatever you're measuring *will actually run* and *will actually do stuff*. This isn't a "safe test mode" or something. So you may need to modify your script a bit, so that you can test it without actually performing the task at hand. Of course, that can backfire, too. You can imagine that a tool designed to modify Active Directory might run a *lot* faster if it wasn't actually communicating with Active Directory, and so your measurement wouldn't really be real-world or useful.

One thing to watch for when running `Measure-Command` is that a single test isn't necessarily absolute proof. There could be any number of factors that might influence the result. Sometimes it helps to run the test several times. Jeff wrote a command `Test-Expression` in the PSScriptTools module that allows you to run a test multiple times, giving you (hopefully) a more meaningful result. There's even a GUI version.

Factors Affecting Performance

There are a bunch of things that can impact a tool's performance.

Collections and arrays can get really slow if they get really big and you keep adding objects to them one at a time. This slowdown has to do with how .NET allocates and manages memory for these things.

Anything storing a lot of data in memory can get a slowdown if .NET has to stop and garbage-collect variables that are no longer referenced. Generally, you want to try and manage reasonable amounts of data in-memory, not great huge wodges of 60GB text files.

Compiling script blocks - as `ForEach-Object` requires - can incur a performance penalty. It's not always avoidable, but it isn't the fastest operation on the planet in some cases.

Wasting memory can result in disk paging, which can slow things down. For example, in the below fragment, we're still storing a potentially and unnecessarily huge list of users in $users long past the point where we're done with it.

```
$users = Get-ADUser -filter *
$filtered = $users | Where { $_.Department -like '*IT*' }
$final = $filtered | Select Name,Cn
$final | Out-File names.txt
```

It'd be better do to this entirely without variables, and getting the filtering happening on the domain controller:

```
Get-ADUser -filter "Department -like '*IT*'" |
Select Name,Cn |
Out-File names.txt
```

Now we're getting massively less data back from Active Directory, and storing none of it in persistent variables. Or to put it more precisely, this is an example of the benefits filtering early.

Here's the problem - We often see beginners write a command like this:

```
Get-CimInstance win32_service -computername server01 |
where state -eq 'running'
```

This may not seem like a big deal but imagine the CIM command was going to return 1000 objects. With the approach we just showed, the first command has to complete and send all 1000 objects, in this case across the wire and *then* the results are figured. Compared to letting `Get-CimInstance` do the filtering in place - on the server - and then only sending the filtered results back.

```
Get-CimInstance win32_service -computername server01 -filter "state = 'running'"
```

There's one other feature you should take advantage of when using `Get-CimInstance` and not many people do. Let's say you are using code like this:

```
Get-CimInstance win32_service -computername $computers-filter "state = 'running'" |
Select-Object -property Name,StartMode,StartName,ProcessID,SystemName
```

The `$Computers` variable is a list of computernames. This pattern is pretty common. Get something and then select the things that matter to you. However, the remote server is assembling a complete Win32_Server instance with all the properties. But you are throwing most of them away. The better approach is to limit what `Get-CimInstance` will send back.

```
$cimParams=@{
  Classname = 'win32_service'
  ComputerName = $computers
  Filter = "state = 'running'"
  Property = 'Name','StartMode','StartName','ProcessID','SystemName'
}
Get-CimInstance  @cimParams | Select-Object -property Name,StartMode,StartName,
ProcessID,@{Name="Computername";Expression={$_.SystemName}}
```

PowerShell will want to display the results using its default formatting. You'll most likely use `Select-Object` or create your own custom object. Regardless, this approach *should* run slightly faster. It may be small, but it can add up. You'll really appreciate this when you are querying 500 servers.

Always look for ways to limit or filter as early in your command as possible. Take advantage of parameters like Filter, Include, Exclude, ID and Name.

Key Take-Away

You should get used to using `Measure-Command` to testing your code, especially if there are several ways you could go. We'll look at other performance related concepts in the Scripting at Scale chapter. But for now your key take-away should be that good coding practices can go a long way toward avoiding performance problems!

PowerShell Workflows: A Primer

Introduced in PowerShell v3, *Workflows* are at attempt to make "scripts" that can take a long time to run, and might need to be interrupted and resumed where they left off. The idea of workflow has been around for a while but it required serious developer skills and Visual Studio. PowerShell workflow was intended as a way for IT Pros to leverage their scripting skills to create a a workflow, which most likely was going to run on remote servers. We have mixed feelings about workflows. Generally speaking, while we appreciate the sentiment, we think the execution is a little lacking, and very confusing.

Note that the feature we're discussing here is *not* the same as workflows in Azure Automation, which look and behave the same (making this chapter totally relevant), but which use a different underlying engine. Our commentary below applies to the "on-prem" workflows bundled in PowerShell itself.

Workflow *seems* like PowerShell scripting. It isn't. You're using a PowerShell-like language to code for Windows Workflow Foundation, or WWF. That is, when you "run" your workflow, it's literally being translated into WWF, which then runs it. PowerShell does not run workflows. This fact is, without a doubt, where almost all pain, confusion, and panic comes from when dealing with workflows. It is easy to think you are writing PowerShell but you aren't.

Terminology

Let's start with some terminology.

- A *workflow* is a special script, nominally written in the PowerShell language, which is compiled into XAML when run. The XAML is handed off to WWF, which actually executes it. WWF is a core part of the .NET Framework.
- A workflow consists of one or more *activities*. These are special little chunks of executable code, written in .NET, and designed to work with WWF (much like cmdlets are written in .NET and designed to work with PowerShell).
- A workflow can also include *logical constructs*, like `If` and `Switch` blocks - although some WWF constructs work differently from their PowerShell counterparts.

To make things interesting (or confusing), many native (that is, "core") PowerShell cmdlets have an *equivalent activity*. So a workflow can contain the command `Get-ChildItem`, because an activity of that name exists. There is not an easy way to see which activities there are, outside of Visual Studio's workflow designer surface.

Theory of Execution

It's important to remember that your workflow script will be compiled *into something else* and handed off to WWF for execution. WWF may, in turn, need to run instances of PowerShell in order to run certain commands within your workflow, but WWF is in fact the one "in charge" of execution. This means the contents of your workflow are governed by WWF rules, which are a bit different from a PowerShell script.

Logical constructs in workflow work the way you'd expect them to, with a couple of differences:

- The `switch` construct doesn't support PowerShell's fancier variations; you need to stick with a simple, basic `switch`.
- The `foreach` construct supports a `-parallel` switch, which enables the contents of the loop to run across multiple threads, effectively processing multiple items at once. You need to ensure that the contents of the loop *can* run simultaneously, and won't get into resource contention or something (like all attempting to append to a given file at the same time).

Activities are a bit odder. First, you cannot use positional parameters. Can. Not. You also need to spell out cmdlet names (aliases are also supported) and parameter names. That's actually all good practice regardless. The oddity comes up when you try to run a command *that doesn't have a corresponding workflow activity*. In that case, WWF starts up an instance of PowerShell to run your command. This is notable, because once your command finishes, that PowerShell instance is shut down - meaning any state changes that your command created, like new variables, *will be gone*.

Variables are the big confusing point. Variables inside the workflow do persist, and they are available to each activity within the workflow. Variables created by a distinct PowerShell instance - as in the above case of running a non-activity command - will *not* persist back into the workflow scope. So if you have a bunch of non-activity commands that need to share information, you end up wrapping them all in an `InlineScript{}` block (which will discuss in a bit) so they can "see" each other. That partially defeats the point of a workflow, because if your code is interrupted, you can only resume to an *activity* - not midway into an `InlineScript{}` block.

So you can see why workflows can be confusing, and tend to not behave as you might initially expect them to. A workflow can persist data, shut down, and resume later with all the data intact - but *only* if that data existed *in the workflow itself* and not in some separate PowerShell process that got spawned.

We suggest reviewing the official reference docs[73] before you dive in as well as the workflow about topics.

[73] https://technet.microsoft.com/en-us/library/jj574142(v=ws.11).aspx

A Quick Illustration

Let's look at a quick example and discuss some important features. We're providing this as Example1.ps1 in the code samples, *but this is not intended to run.* It's just a compilation of several points, so we can discuss them all at once.

Example

```
workflow Example {
    Param(
        [string]$Value
    )

    $procs = Get-Process
    $total_ram = 0
    $services = $null
    $events = $null
    $result = ""

    foreach -parallel ($proc in $procs) {
        $workflow:total_ram += $proc.ws
    } #foreach
    Write-Output "Total RAM used $total_ram"

    sequence {

        $folders = Get-ChildItem -Path $value -Directory

        parallel {
            $workflow:services = Get-Service
            $workflow:events = Get-EventLog -LogName Security
        } #parallel

        $workflow:result = InLineScript {
            "Hello it is $(Get-Date)"
        } #inline script
        Write-Output "Check $workflow:result"

        $nics = Get-NetAdapter

    } #sequence

    Write-Output $result
```

```
    Write-Output "$($folders.count) folders"
    Write-Output "$($services.count) services"
    Write-Output "$($events.count) events"
    Write-Output "$($nics.count) NICs"

} #workflow

Example -Value "c:\"
```

You *can* run this, by the way - it just doesn't do anything useful. It's instructional to see how long it takes to run, though - showing that workflows aren't necessarily about speed.

You cannot run a workflow from a PSDrive that is not a logical disk. Suppose you have a PSDrive called Scripts which is pointing to C:Scripts. You can't execute the workflow command form Scripts:. You would need to change to some folder on the C:\ drive, or any other logical disk.

Here's the output:

```
0 folders
218 services
13293 events
0 NICs
```

And here's what to note:

- Workflows can have input parameters, in addition to a slew of automagically-created common parameters, which we'll get into in a bit.
- `Get-Process` is available as an activity, and so it runs "as-is." The resulting process objects go into `$procs`. Notice that, by declaring `$total_ram` in the workflow, we define it as a workflow-level variable. That's why you later see us referring to it as `$workflow:total_ram`. Doing so allows us to "feed" the variable into what becomes an inline script.
- Our mathematics in adding up the `ws` (working set) property can't be done in workflow, and so it becomes an implicit inline script. By using the `$workflow:` variable modifier, we "inject" the workflow-level variable into the inline script, enabling us to capture those results.
- The `Write-Output` command is still the correct way to output from a workflow.
- The `sequence` block encloses a list of activities that must be run in exactly the order shown. Sequential execution is actually the default, but you could nest a block of `sequence` inside of a `parallel`. By contrast, a `parallel` block will execute its contents *in any order*, with no way to know up front what that order will be.
- Notice our use of `$workflow:` for `$services` and `$events` but not for `$folders` and `$nics`, and how that affects the output.

- We have one explicit `InlineScript`, which again is a new PowerShell process. `Get-NetAdapter`, which is not an activity, is an implicit inline script.

Variable scope gets tricky.

- Inside a loop - such as our run-in-parallel `foreach` - workflow variables are visible automatically, and the loop does not constitute its own scope. We only had to use `$workflow:` inside the loop because our math statement is an implicit inline script, spawning a new instance of PowerShell. Using the modifier enables us to persist the data from those separate processes.
- Inside a `sequence` or `parallel` block, you can *see* workflow-level variables, but you must use the `$workflow:` modifier in order to *change* a workflow-level variable. This is why `$services` and `$events` can be used in our output, but `$nic` and `$folders` and `$result` don't.

The variable stuff - which is how you pass information from place to place, of course - is what makes workflows especially challenging. For example, we'll see folks try to make this change:

```
$result = ""
InLineScript {
    $result = "Hello it is $(Get-Date)"
} #inline script
Write-Output "Check $result"
```

This will simply result in "Check" being output - `$result` is still empty at that point. It was changed *inside an inline script*, and as soon as the inline script's PowerShell process ended, whatever was inside the `InlineScript` block ceased to exist. We can't use the `$workflow:` modifier because `$result` isn't defined at the workflow level; it's defined in a `sequence` block, which is its own scope. Here's how to do what we're attempting (this is Example2.ps1):

Example2

```
workflow Example {
    Param(
        [string]$Value
    )

    $procs = Get-Process
    $total_ram = 0
    $services = $null
    $events = $null
    $result = ""

    foreach -parallel ($proc in $procs) {
        $workflow:total_ram += $proc.ws
```

```
    } #foreach
    Write-Output "Total RAM used $total_ram"

    sequence {

        $folders = Get-ChildItem -Path $value -Directory

        parallel {
            $workflow:services = Get-Service
            $workflow:events = Get-EventLog -LogName Security
        } #parallel

        $workflow:result = InLineScript {
            "Hello it is $(Get-Date)"
        } #inline script
        Write-Output "Check $workflow:result"

        $nics = Get-NetAdapter

    } #sequence

    Write-Output $result
    Write-Output "$($folders.count) folders"
    Write-Output "$($services.count) services"
    Write-Output "$($events.count) events"
    Write-Output "$($nics.count) NICs"

} #workflow

Example -Value "c:\"
```

Now, we've defined `$result` as a workflow-level script, right at the top. We _assign the results of the `InlineScript` to `$result`, using the `$workflow:` modifier because we're inside a `sequence` block. Again - it's tricky stuff, and can require a lot of experimentation.

Within an `InlineScript`, you cannot use the `$workflow:` modifier, just to keep things fun. Instead, use the `$using:` modifier. Here's another revision1:

Example3

```
workflow Example {
    Param(
        [string]$Value
    )

    $procs = Get-Process
    $total_ram = 0
    $services = $null
    $events = $null
    $result = ""

    foreach -parallel ($proc in $procs) {
        $workflow:total_ram += $proc.ws
    } #foreach
    Write-Output "Total RAM used $total_ram"

    sequence {

        $folders = Get-ChildItem -Path $value -Directory

        parallel {
            $workflow:services = Get-Service
            $workflow:events = Get-EventLog -LogName Security
        } #parallel

        $workflow:result = InLineScript {
            "Hello it is $(Get-Date)"
            "There are $($using:procs.count) Processes"
        } #inline script

        $nics = Get-NetAdapter

    } #sequence

    Write-Output $result
    Write-Output "$($folders.count) folders"
    Write-Output "$($services.count) services"
    Write-Output "$($events.count) events"
    Write-Output "$($nics.count) NICs"

} #workflow
```

```
Example -Value "c:\"
```

Go ahead and run this, if you like. Here's the output on our machine:

```
Total RAM used 1620455424
Hello it is 06/21/2020 09:06:37
There are 62 Processes
0 folders
218 services
13310 events
0 NICs
```

As you can see, the `$using:` modifier inside the `InlineScript` block enabled us to "pass in" a workflow-level variable. Because we captured the `InlineScript` block output into another workflow-level variable, we were able to display that result at the end.

So in the end, you wind up declaring a lot of variables up-front, and then referencing them using the appropriate modifiers:

- Use `$workflow:` to reference a top-level variable in a sub-block, excepting an `InlineScript`.
- Use `$using:` to reference a top-level variable in an `InlineScript`.

When to Workflow

We differ a bit from the official documentation in when you might want to use a workflow. We feel they're best used when:

- You have some long-running process that may need to be interrupted and resumed. This *isn't* necessarily something like computer provisioning, though, which we do feel is better accomplished by DSC.
- You have a large amount of data to process and want to parallelize it. However, workflow isn't the only means of doing so, and depending on the exact task, workflow may not be the least complicated. Workflow also isn't specifically intended to be fast - WWF imposes some performance overhead that other means (which we discuss in "Scripting at Scale") might not.
- You're using Azure Automation.

We don't necessarily feel that workflow is a go-to simply when you need some task run on multiple remote machines. To do that, PowerShell just uses Remoting, and you could accomplish more or less the same thing using `Invoke-Command` and its `-AsJob` parameter. So the above criteria are really the points where we start *considering* workflow.

Sequences and Parallels are Standalone Scopes

Another element that sets workflows apart from the type of scripting you are used to is a `Sequence` and a `Parallel` block. The whole point of a workflow is to orchestrate some set of steps; sometimes these steps need to be in a specific order, and other times they can be run in parallel. That's what the `Sequence` and `Parallel` blocks, which we introduced in the illustration above, are for. And, as we pointed out, tricky part is that you need to think about each sequence or parallel as its own standalone variable scope. The only way to pass variables *between* scopes is to *first* define the variable in the workflow (outside of a block), and then reference them *inside* the block by using the `$workflow:` prefix.

Workflow Example

Now let's do something a bit more functional. We want to add up the amount of space a given set of user home folders take up on disk. This is one of the better examples we've come up with, because it *is* realistic, and it *does* take a long time to run if you do them all sequentially. This is DirSizer.ps1 in the sample code. The presumption is that you can provide a root path, whose immediate child directories are each a user's home folder.

Get-UserFolderSizes

```
workflow Get-UserFolderSizes {
    Param(
        [string[]]$RootPath
    )

    foreach -parallel ($path in $RootPath) {
        Write-Verbose "Scanning $path"

        # Get subdirectories
        $subs = Get-ChildItem -Path $path -Directory
        Write-Verbose "$($subs.count) user folders"

        foreach -parallel ($sub in $subs) {
            Write-Verbose "Scanning $($sub.FullName)"

            $size = Get-ChildItem -recurse -Path ($sub.FullName) -File |
                    Measure-Object -Property Length -Sum |
                    Select-Object -ExpandProperty Sum

            Write-Verbose "Size of $($sub.FullName) is $size"
```

```
            $props = @{Path=$sub.FullName
                       Size=$size}
            $obj = New-Object -TypeName PSObject -Property $props
            Write-Output $obj

        } #foreach subdirectory

    } #foreach path
}

Get-UserFolderSizes -RootPath c:\Users
```

You'll notice that `Write-Verbose` works quite well, and in fact prefixes each line of output with the name of the computer it came from:

```
VERBOSE: [localhost]:Size of C:\Users\User is 3160
```

That behavior recognizes the fact that workflows are intended to be pushed out to multiple computers, so most output gets tagged with the name of the computer that produced it. For example:

```
Path                  : C:\Users\User
Size                  : 3160
PSComputerName        : localhost
PSSourceJobInstanceId : 0448746e-a2c8-4e44-b1ff-92aa32851062
```

We only created two of those four properties; the other two were magically added by the workflow engine.

Notice that we didn't have a lot of fussing with variable scope? If you're careful with your design, you can avoid having to persist data across scopes. Here, because we only use the scope-less `foreach` loop, everything is basically a workflow-level variable. We're pretty sure all the commands we used exist as activities, too, which eliminates implicit inline scripts. Although `New-Object` may be running in an inline script. We're not sure, but at least our usage assigns the result of the inline script to a variable (`$obj`), getting that result into the workflow's scope.

Workflow Common Parameters

Every workflow picks up a whole slew of common parameters, reflecting some of the built-in functionality that the workflow engine provides. A big one is `-PSComputerName`, which accepts a list of computer names. Each computer named will be sent a copy of the workflow, and asked to execute it. The local computer will not execute the workflow unless it's in the list of names. Communication happens via PowerShell Remoting, which must be enabled and working.

The `-PSCredential` parameter specifies an alternate credential to run the workflow under, and is only valid along with `-PSComputerName`. You can also use `-PSParameterCollection`, which is a hashtable, to provide different input arguments to each named computer, allowing workflow to be customized on each.

This means we could define the dirsizer workflow locally and invoke it remotely specifying a path that is relative to the remote machine.

```
Get-UserFolderSizes -RootPath c:\Users `
-PSComputername server01 `
-PSCredential company\administrator
```

Another gotcha is that if you are running PowerShell v5 or later locally but the remote server is running v3 or v4, you'll most likely get an error.

Find more common parameters in the docs[74].

One of the reasons workflows held any interest was for it's ability to run tasks in parallel. PowerShell 7 now supports this feature with ForEach-Object‘.

Checkpointing Workflows

This is kinda the whole magic point about workflows: they can be interrupted, and can pick up later where they left off. A checkpoint saves the "state" of the workflow, which includes workflow-level variables, any output created to that point, and so on. Checkpoints actually get saved to disk (within the user profile directory), meaning they can survive a reboot of the computer the workflow is running on.

Writing a checkpoint incurs processing overhead, so you don't want to just drop these things in every other line of your code. Focus on checkpointing after major areas of work are done, especially long-running ones you'd hate to have to repeat. Also place them after sections which would be impractical to repeat, such as joining a machine to a domain.

If you run a workflow with the `-PSPersist:$true` common parameter, you'll get automatic checkpoints at the beginning and end of the workflow, plus whichever ones you specify manually. To specify one manually, either add the `-PSPersist:$true` common parameter *to any activity*, and you'll get a checkpoint after that activity completes. You can't use that on `InlineScript` blocks, though. You can also run `Checkpoint-Workflow` anywhere within a workflow (but not in an `InlineScript`) to manually create a checkpoint at that spot.

Checkpoints that are part of a pipeline won't be taken until the entire pipeline completes. Checkpoints in a `Parallel` block are taken when the entire block completes. Checkpoints in a `Sequence` are taken immediately.

[74] https://docs.microsoft.com/en-us/powershell/module/psworkflow/about/about_activitycommonparameters?view=powershell-5.1

Workflows and Output

PowerShell workflows were intended to be executed across multiple remote servers, sight unseen. Workflows don't need to write anything back to the pipeline, and a good argument could be made that they shouldn't. Their task is to get a bunch of steps accomplished in the most efficient matter possible. They are intended to *do* stuff, not get stuff. Using `Write-Verbose` statements is still a good practice for troubleshooting but don't try to use `Write-Host` or feel you need something written to the pipeline.

This is a really important concept. If a workflow *does* need to create some output, you need to think about *where it will go*. You won't personally be there to see it run, in most cases, and so you need to consider writing output to disk, to a central database, or some other location, so that you can get to the output later.

Your Turn

If you are up for a challenge we have one for you. We haven't gone into tremendous detail about workflow because they still are a special case in our opinion, but we've given you pointers to additional resources. We're also going to assume that you can run the workflow on a computer where you can make changes and manually reverse them.

Start Here

Let's say that your company is deploying a special management application to your servers. In order to prepare for deployment, you need to create a workflow that accomplishes these tasks.

- Create a local user account called ITApp with a default password.
- Create a folder called C:ITApp and share it as ITApp giving Everyone ReadAccess and the ITApp user full control.
- Under C:ITApp create folders Test_1 to Test_10
- Set the Remote Registry service to auto start
- Log each step to a text file called C:ITAppWF.txt

Your Task

You will need to think about what order these activities need to run and what variables you might need to pass through the workflow. Remember, it may look like you are using cmdlet names, but they are actually activities. And we'll give you a tip that the common `-PSCredential` parameter is used to authenticate to the remote server. As you work on this exercise you will almost certainly get errors. Read the error message as it will often explain what you need to do to fix the problem.

Our Take

You can find our solution in the chapter downloads.

Solution.ps1

```
Workflow ITAppSetup {

Param(
[Parameter(Mandatory)]
[string]$Password,
[string]$Log = "c:\ITAppWF.txt"
)

Set-Content -Value "[$((Get-Date).timeofDay)] Starting Setup" -Path $log

Add-Content -Value "[$((Get-Date).timeofDay)] Configuring Remote Registry service" -\
Path $log
Set-Service -Name RemoteRegistry -StartupType Automatic

Sequence {
    Add-Content -Value "[$((Get-Date).timeofDay)] Creating local user" -Path $log
    net user ITApp $Password /add
}

Sequence {
    Add-Content -Value "[$((Get-Date).timeofDay)] Testing for C:\ITApp folder" -Path\
 $log
    if (Test-Path -Path "C:\ITApp") {
        Add-Content -Value "[$((Get-Date).timeofDay)] Folder already exists." -Path \
$log
        $folder = Get-Item -Path "C:\ITApp"
    }
    else {
        Add-Content -Value "[$((Get-Date).timeofDay)] Creating C:\ITApp folder" -Pat\
h $log
        $folder = New-Item -Path C:\ -Name ITApp -ItemType Directory
        Add-Content -Value "[$((Get-Date).timeofDay)] Created $($folder.fullname)" -\
Path $log
    }

    Add-Content -Value "[$((Get-Date).timeofDay)] Testing for ITApp share" -Path $log
    if (Get-SmbShare ITApp -ErrorAction SilentlyContinue) {
        Add-Content -Value "[$((Get-Date).timeofDay)] File share already exists" -Pa\
```

```
th $log
    }
    else {
      Add-Content -Value "[$((Get-Date).timeofDay)] Creating file share" -Path $log
      New-SmbShare -Name ITApp -Path $folder.FullName -Description "ITApp data" -Ful\
lAccess "$($env:computername)\ITApp" -ReadAccess Everyone
    }

    Add-Content -Value "[$((Get-Date).timeofDay)] Creating subfolders" -Path $log
    foreach -parallel ($i in (1..10)) {
        $path = Join-Path -Path $folder.FullName -ChildPath "Test_$i"
        #add a random offset to avoid contention for the log file
        $offset = Get-Random -Minimum 500 -Maximum 2000
        Start-Sleep -Milliseconds $offset
        Add-Content -Value "[$((Get-Date).timeofDay)] Creating $path" -Path $log
        $out = New-Item -Path $folder.FullName -Name "Test_$i" -ItemType Directory
    }
}
 Add-Content -Value "[$((Get-Date).timeofDay)] Setup complete" -Path $log

} #close workflow
```

If you came up with something like this you probably ran into some issues such as contention for the log file or trying to use the `-Not` operator. While a command like this works as you expect it:

```
if (-Not (Test-Path -path C:\foo)) {
 ...
}
```

When executed in a Workflow the operator doesn't appear to be evaluated. We ended up writing our solution to avoid using `-Not`.

We want to point out that there's no reason you couldn't have written a traditional PowerShell script to accomplish these same tasks. Or used DSC. That's why we think many IT Pros mis-use workflow. You do get to take advantage of built-in parameters like `-PSComputername` and parallelism but you'll have to decide if it is worth the trade-off.

Let's Review

Ok. Quick review time.

1. In a workflow, most cmdlets are treated as what type of workflow element?
2. What are some of the benefits of using a workflow?
3. What feature saves the state of a workflow so that it can be resumed later?

Review Answers

We answered like this:

1. Activity.
2. The most attractive feature is the use of parallelism. We also like that you built-in support for remoting and jobs.
3. Checkpoints.

Globalizing Your Tools

This is a bit of a specialized chapter, and we realize up front that a lot of folks won't ever need it. With that in mind, we'll also try to keep it concise.

Globalization is the process of writing your tools in a way that makes them easier to *localize*. Localization is translating parts of your tool to reflect a specific *culture*. A culture is more than a language; it can also incorporate widely understood colors, iconography, and other communication elements. Because *most* PowerShell scripts are text-only, localization does tend to come down to language, which means you translate the text strings - error messages, verbose messages, and the like - into another language.

Neither of us are fluent in anything but English, and even that fluency is debatable sometimes. For our examples, we're relying on machine translation, so please forgive us if anything is horribly amiss.

Starting Point

We're going to go back a few chapters and start with a script that we've used previous. For this chapter's downloadable sample code, we'll call it StartingPoint.ps1.

StartingPoint.ps1

```
Function Get-MachineInfo {

[CmdletBinding()]
Param(
    [Parameter(ValueFromPipeline,
        Mandatory)]
    [Alias('CN', 'MachineName', 'Name')]
    [string[]]$ComputerName
)

BEGIN {}

PROCESS {
    foreach ($computer in $computername) {
        Try {
            Write-Verbose "Connecting to $computer"
```

```
$params = @{
    ComputerName  = $Computer
    ErrorAction   = 'Stop'
}

$session = New-CimSession @params

Write-Verbose "Querying $computer"
#define a hashtable of parameters to splat
#to Get-CimInstance
$cimparams = @{
    ClassName  = 'Win32_OperatingSystem'
    CimSession = $session
    ErrorAction = 'stop'
}
$os = Get-CimInstance @cimparams

$cimparams.Classname = 'Win32_ComputerSystem'
$cs = Get-CimInstance @cimparams

$cimparams.ClassName  = 'Win32_Processor'
$proc = Get-CimInstance @cimparams | Select-Object -first 1

$sysdrive = $os.SystemDrive
$cimparams.Classname = 'Win32_LogicalDisk'
$cimparams.Filter = "DeviceId='$sysdrive'"
$drive = Get-CimInstance @cimparams

Write-Verbose "Outputting for $($session.computername)"
$obj = [pscustomobject]@{
    ComputerName      = $session.computername.ToUpper()
    OSVersion         = $os.version
    OSBuild           = $os.buildnumber
    Manufacturer      = $cs.manufacturer
    Model             = $cs.model
    Processors        = $cs.numberofprocessors
    Cores             = $cs.numberoflogicalprocessors
    RAM               = $cs.totalphysicalmemory
    Architecture      = $proc.addresswidth
    SystemFreeSpace   = $drive.freespace
}
Write-Output $obj
```

```
            Write-Verbose "Closing session to $computer"
            $session | Remove-CimSession
        }
        Catch {
            Write-Warning "FAILED to query $computer. $($_.exception.message)"
        }
    } #foreach

} #PROCESS

END {}

} #end function

Get-MachineInfo -ComputerName $env:COMPUTERNAME
```

Notice that we've explicitly removed the comment-based help from this script? That's on purpose, as comment-based help isn't really globalized. Instead, we'd rely on "full" help files, created with Platyps, which we've discussed in an earlier chapter.

Make a Data File

Our first step will be to create a separate file containing our English-language (specifically, US English) text strings. We use a separate file because that makes it easier to hand just that file off to a professional translator. They can create equivalents for whatever other languages we might need.

PowerShell's *data language* provides a really minimal set of instructions, meaning these files aren't scripts per se. That helps prevent malicious code from sneaking in. Add anything illegal, and the module won't load.

File naming is important with data files. But, in our downloadable sample code, we aren't really able to arrange the files into a proper module form. So here's what we're going to do: in the folder for this chapter, we'll have a `Modules` folder that represents a normal module location, like `Program Files\WindowsPowershell\Modules`. Within it, we'll create a folder named GloboTools, which represents a module named GloboTools.

In that folder, we'll obviously have GloboTools.psm1 and GloboTools.psd1, which are the module file and the module manifest. You could load this module to try it out by running `Import-Module` and providing the full path to the .psd1 file.

The following, then, is en/GloboTools.psd1. The *en* part is a partial culture code, as in *en-US*. There's a full list on MSDN[75], and it's the first, lowercase two-letter part we're using for the folder name.

[75] https://msdn.microsoft.com/en-us/library/ee825488(v=cs.20).aspx

```
ConvertFrom-StringData @'
    connectingTo = Connecting to
    queryingFrom = Querying
    closingSessionTo = Closing session to
    outputFor = Output for
    failed = FAILED to query
'@
```

Basically, we created a hash table of sorts. Each pair consists of a *message identifier* (that's our word for it, not an official term), which has no spaces. Each identifier is followed by the appropriate words for this culture.

Don't confuse the language .psd1 files for the module manifest .psd1 file. They're both the same file extension, but they have different purposes. Language .psd1 files are stored under a culture-specific subfolder in the module folder.

Use the Data File

Of course, we need to actually use that data file. So here's our GloboTools.psm1, and you can see that we've replaced our static messages with references to `$msgTable`, along with a message identifier.

Notice in the below how we use a subexpression to access specific messages from `$msgTable`.

GloboTools.psm1

```
Import-LocalizedData -BindingVariable msgTable

Function Get-MachineInfo {

[CmdletBinding()]
[alias("gmi")]
Param(
    [Parameter(ValueFromPipeline, Mandatory)]
    [Alias('CN', 'MachineName', 'Name')]
    [string[]]$ComputerName
)

BEGIN {}

PROCESS {
```

```
    foreach ($computer in $computername) {
        Try {
            Write-Verbose "$($msgTable.connectingTo) $computer"
            $params = @{
                ComputerName  = $Computer
                ErrorAction   = 'Stop'
            }

            $session = New-CimSession @params

            Write-Verbose "$($msgTable.queryingFrom) $computer"
            #define a hashtable of parameters to splat
            #to Get-CimInstance
            $cimparams = @{
                ClassName  = 'Win32_OperatingSystem'
                CimSession = $session
                ErrorAction = 'stop'
            }
            $os = Get-CimInstance @cimparams

            $cimparams.Classname = 'Win32_ComputerSystem'
            $cs = Get-CimInstance @cimparams

            $cimparams.ClassName  = 'Win32_Processor'
            $proc = Get-CimInstance @cimparams | Select-Object -first 1

            $sysdrive = $os.SystemDrive
            $cimparams.Classname = 'Win32_LogicalDisk'
            $cimparams.Filter = "DeviceId='$sysdrive'"
            $drive = Get-CimInstance @cimparams

            Write-Verbose "$($msgTable.outputFor) $($session.computername)"
            $obj = [pscustomobject]@{
                ComputerName      = $os.CSName
                OSVersion         = $os.version
                OSBuild           = $os.buildnumber
                Manufacturer      = $cs.manufacturer
                Model             = $cs.model
                Processors        = $cs.numberofprocessors
                Cores             = $cs.numberoflogicalprocessors
                RAM               = $cs.totalphysicalmemory
                Architecture      = $proc.addresswidth
                SystemFreeSpace   = $drive.freespace
```

```
            }
            Write-Output $obj

            Write-Verbose "$($msgTable.ClosingSessionTo) $computer"
            $session | Remove-CimSession
        }
        Catch {
            Write-Warning "$($msgTable.failed) $computer. $($_.exception.message)"
        }
    } #foreach

} #PROCESS

END {}

} #end function
```

Notice the first line in our module, which is *outside of any function*. The `Import-LocalizedData` command *magically* checks our system's configured culture, which happens to be `en-US`. It then looks in the `\en` subfolder for a .psd1 file having the module's name, imports it, and stores the results in `$msgTable`, which is the `-BindingVariable` we specified (note that the variable name, when used with the parameter, doesn't include the dollar sign). In our script, `$msgTable` now represents all of our localized strings, and we can access each one as a property of that variable.

If you `Import-Module GloboTools.psd1`, and then run `Get-GloboMachineInfo -ComputerName localhost -Verbose`, you'll see the localized strings in action.

Adding Languages

We'll save the following as `de\GloboTools.psd1` and again apologize for machine translations. We have several German friends and we hope they're giggling.

```
ConvertFrom-StringData @'
    connectingTo = Verbinden mit
    queryingFrom = Abfrage von
    closingSessionTo = Abschlusssitzung zu
    outputFor = Ausgabe f□r
    failed = gescheitert
'@
```

Testing this is a little tricky; you basically have to add a `-UICulture` parameter to `Import-LocalizedData` to force it to import something other than what your system is configured to do.

Another option is to use the `Test-WithCulture` command from the Jeff's PSScriptTools module. Adjust paths in the following as needed.

```
Test-WithCulture -Culture de-de -Scriptblock { import-module .\GloboTools.psd1
-force get-machineinfo localhost -Verbose }
```

Defaults

Now, there's a downside to this approach we've shown you, which is that `Import-LocalizedData` will simply *not do anything* if the culture it needs isn't present. So you can take another step to provide a default - say, in English. Just add this to the top of the module script file, before the call to `Import-LocalizedData`:

```
$msgTable = Data {
# culture-en-US
ConvertFrom-StringData @'
    connectingTo = Connecting to
    queryingFrom = Querying
    closingSessionTo = Closing session to
    outputFor = Output for
    failed = FAILED to query
'@
}
```

This pre-populated `$msgTable` with English strings, and allows `Import-LocalizedData` to *overwrite* those with another culture, if needed and if that other culture has a file present.

Let's Review

We don't really have an exercise for you with this chapter. But let's at least ask a few review questions.

1. What type of file do you use for text translations?
2. What cmdlet imports the localized data?
3. Where do you put your localized data files?

Review Answers

And some answers are:

1. A PowerShell data file or .psd1 file.
2. `Import-LocalizedData`. That was easy.
3. In culture-specific subfolders like de-DE or vi-VN

Using “Raw” .NET Framework

This topic comes up a *lot*. So let’s break it down a bit: our overwhelming preference is to use “native” PowerShell whenever possible. That means running commands - cmdlets, functions, and so on - versus external applications, or using .NET Framework classes. We have three core reasons for this preference:

1. Commands are easier to read in a script, can be more admin-focused, support discoverability and help, and are usually more consistently designed.
2. Commands can be mocked in Pester tests, which is hugely useful.
3. Commands can consistently use a set of common parameters that enable verbose output, error handling, pipeline capturing, and much more.

But you’ll run into times when there just isn’t a command for what you need to do - and that’s what this chapter is all about.

Understanding .NET Framework

The .NET Framework consists of a set of classes that perform an enormous variety of tasks. There are simple classes for manipulating strings, complex classes for working with Active Directory, and super-complex classes for dealing with databases and data structures. The Framework is *huge*. The Framework is accessible from any language that can run in the .NET Common Language Runtime (or CLR) or Dynamic Language Runtime (DLR), which means PowerShell is “in.”

However: just because you’re using .NET *in* PowerShell doesn’t mean you’re “scripting in PowerShell” or that you’re “.NET scripting.” The Framework is a hugely different beast from PowerShell. It’s more complex, it’s very developer-centric, it’s documented differently, and so on. You may find that your favorite Q&A forums for PowerShell can’t help as much when you start doing .NET, and that you have to take your questions to a developer-centric site like StackOverflow.com.

Let’s be clear about something: if .NET was a good administrator tool, PowerShell would literally not exist, and we’d all be “scripting” in C# instead. In fact, we regularly see people struggling to make complex .NET stuff work in PowerShell, and wonder why they don’t just fire up Visual Studio and start a new C# project, because we know what they’re doing would be faster and easier that way. PowerShell is not a “first class citizen” in the .NET world. It lacks many crucial .NET language features that much of the Framework takes for granted - things like proper event management, asynchronous callbacks, generics, and more. It is *not* accurate to say that “PowerShell can do anything in .NET.” It can’t, always, and you’ll get a bloody forehead banging your head against that wall.

But there *are* times when something you need exists within .NET, doesn’t exist in PowerShell command, and will work fine in PowerShell. For *those* instances, this chapter exists.

Let’s start with some terminology:

- A *type* is a blueprint for the way a piece of software can be used. A type typically defines an *interface*, the means by which you tell the software what to do.
- A *class* is an actual implementation of a type, including all the code that makes the interface actually work.
- A class, through its type definition, usually has *members*. These members are the individual elements of the interface. A class is meant to be a bit of a black box. The members are the buttons you push and dials you read, while what goes on inside - the code - is a mystery. The main kinds of members include:
  - *Properties*, which expose information about the class.
  - *Methods*, which ask the class to perform a task.
  - *Events*, which enable you to respond to things that happen to the class.
- Some properties and methods are *static*, which means the class can operate these without additional information, and without having to create an *instance* of the class. On the other hand, most members are *instance*, which means they can only be used once you’ve instantiated the class. Think of a television: you can’t just stand up and announce “turn on the TV.” First, you have to *go get a TV*. That is, you have to get a concrete *instance* of the abstract “TV” type. Once you have a particular TV, you can turn it on.

Interpreting .NET Framework Docs

Always run `$PSVersionTable` in PowerShell to see what version of the .NET Framework you’re using. Then, make sure whatever docs you read are for the same version. We can’t tell you how much time we’ve wasted trying to get something to work, only to realize we were reading instructions from a different version!

MSDN.Microsoft.com is the base point for .NET Framework’s documentation. We find, however, that it’s often easier to start with a search engine, using a .NET type name if possible. That way, you can jump straight to what you need.

Let’s use the documentation for System.DateTime[76] as an example.

- Notice the “Other Versions” dropdown, where you can select documentation for a specific version of .NET.
- You’ll first see several *constructors*. These are basically static methods of the class, which can be used to create a new instance of the class. Constructors often take input arguments, which usually control how the new instance is created.

[76] https://msdn.microsoft.com/en-us/library/system.datetime.aspx

- Next you'll see *properties*. Ones with a big red "S" icon are static, and can be used without running a constructor to create an instance. For example, the `Now` property is static - you don't need a particular date or time to get the *current* date or time. However, `TimeOfDay` is an instance property - until you *have* a date, you can't find out what time of day that date is.
- Next are *methods*, which can again be instance or static.
- You may also see *operators*, which are tiny bit like methods in that they ask the class to perform something, although in this case they only perform comparisons.
- There are sometimes *fields*, which usually contain static information about the class' capabilities or features.

You can click through on any member to read more about it. Go ahead and take a second to look up `System.DateTime` and make sure you can identify the above items.

Coding .NET Framework in PowerShell

Now let's talk about using these things!

Static Members

Remember, a *static* member (which appears in the docs with a big red "S" icon) doesn't require you to instantiate the class. That is, you don't need to *create an object*. You simply use the class name, in square brackets, followed by two colons, and then the member:

```
[System.DateTime]::Now
```

Or a method:

```
[System.DateTime]::DaysInMonth(2020,2)
```

We looked that up in the documentation, by the way, to figure out how to use it. It says `DaysInMonth()` takes on integer for the year, and another for the month, and then tells you how many days that month has.

Instance Members

These require you to instantiate the class - or, in PowerShell terms, to create an object. To do so, you'll use `New-Object` along with the class' type name. *You have to pick a constructor* in order to create the new object! Some types will allow you to create a new instance using zero input arguments; other classes can't create a new instance of themselves unless you provide some kind of input. Perusing the constructors for `System.DateTime`, for example, they all appear to require one or more arguments, which means we'll have to provide them:

```
$dt = New-Object -TypeName System.DateTime -ArgumentList 1
```

Now, here's how arguments work: if you look at the docs, you'll see that arguments are simply a comma-separated list inside parentheses. From PowerShell's perspective, you just provide a comma-separated list of arguments to the `-ArgumentList` parameter. .NET magically figures out which constructor you're using, based on the data types of your arguments, and the number of arguments you provide. There's no other way to specify a constructor, so you have to get the arguments spot-on.

For example, `System.DateTime` has five constructors:

- One accepts a number
- One accepts a number and a "DateTimeKind" enumeration
- One accepts three numbers
- One accepts six numbers
- One accepts seven numbers

You'll never, ever see two constructors that accept the same number of arguments all of the same data type, in the same order.

Wait - "enumerations?" Yeah. These are basically like a `ValidateSet()` parameter attribute, in that the enumeration is a list of acceptable values. Under the hood, they're always numbers, but to you, they're friendly-looking names. They just look a bit funky in PowerShell code. We had to look up the DateTimeKind enumeration[77] by clicking through from the constructor's help page.

```
$dt = New-Object -Type System.DateTime -Arg (500, [System.DateTimeKind]::Utc)
```

Once you've got your new instance, you can use its members:

```
$dt.DayOfWeek
$dt.ToLocalTime()
```

And that's about it. It's a lot harder to find stuff in .NET than it is to use it!

Loading Assemblies

PowerShell can access most of the "core" .NET stuff (like, in the System namespace) without needing to load anything. But other times, you'll first need to load the .NET *assembly* into memory, so PowerShell knows what you're trying to use. If you know the path and filename of your DLL, it's easy:

[77] https://msdn.microsoft.com/en-us/library/shx7s921.aspx

```
[System.Reflection.Assembly]::LoadFile("Mydll.dll")
```

If you don’t - say, if you’re trying to use something from the Global Assembly Cache (GAC) - it’s sometimes a bit tougher. Ideally, you should be able to use the `Add-Type` command. In theory, the GAC knows where the DLL files are for every type in the GAC.

```
Add-Type -Assembly My.Big.Crazy.Framework
```

Notice you’re just providing the type name, not a filename. But if this doesn’t work - and sometimes, it doesn’t - try:

```
[System.Reflection.Assembly]::LoadWithPartialName('My.Big.Crazy.Framework')
```

The `LoadWithPartialName()` method can be a little bit of a bad practice (it’s actually deprecated), and if you have a lot of side-by-side versioning going on, it can potentially load a version you didn’t mean to load. So you want to try and avoid it and stick with `Add-Type` instead, which actually lets you be very specific about what to load:

```
Add-Type -AssemblyName "Microsoft.SqlServer.Smo, `
Version=12.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91"
```

What’s fun is that `Add-Type` has a `-Path` parameter, meaning it can also replace the `LoadFile()` method, too! And it makes for easier reading in your scripts!

Wrap It

OK, let’s say you’ve found a magical .NET type that’ll do everything you’ve always dreamed of, and more. You’ve found its docs, you figured out how to use it, and you’ve got some working code. You’re done!

Not so fast.

A true Toolmaker *isn’t* done. Not until that thing has been wrapped into a PowerShell command, so that future generations don’t have to go through all that figuring-out pain, ever again. Let’s run through a quick example.

PowerShell 5 (and later) has an `[enum]` type accelerator that makes it easier to work with enumerations. It has two static methods, `GetValues()` and `GetNames()`. Given an enumeration type, it can get you the names (that is, the possible choices) in the enumeration, or the underlying values. For example:

```
[enum]::GetNames([System.Environment+SpecialFolder])
```

So this `[enum]` bit works like a normal type; it's essentially a shortcut to a .NET Framework type that has a longer and less-convenient name. Two colons indicates we're using a static member, and we've used the `GetNames()` method. According to the docs[78], the method needs a type as its input argument, and it'll get the enumerations for that type. In PowerShell, type names go inside [square brackets], like `[System.DateTime]`. In our case, we wanted the `SpecialFolder` enumeration from the `System.Environment` class.

We told you, the toughest part was figuring out .NET, not using it.

Anyway, the docs for `System.Environment`[79] links to the `SpecialFolder` enumeration[80], and so we constructed `[System.Environment` from the type name, and `+SpecialFolder]` from the name of the enumeration itself. That is, `System.Environment` contains the `SpecialFolder` enumeration.

So we came up with this (which is Example.ps1 in the sample code):

Example.ps1

```
function Get-SpecialFolders {
    [CmdletBinding()]
    Param()

    $folders = [enum]::GetNames([System.Environment+SpecialFolder])
    Write-Verbose "Got $($folders.count) folders"
    foreach ($folder in $folders) {
     [pscustomobject]@{
        Name = $folder
        Path = [environment]::GetFolderPath($folder)
     }
    }
}
Get-SpecialFolders
```

Notice the empty `Param()` block? That's so we could still have `[CmdletBinding()]` even though we don't need any input parameters for this function. Notice that we've turned this obscure .NET-ish code into a simple PowerShell function that returns familiar-looking objects as its output. *This* is the goal of a Toolmaker!

Your Turn

This is great toolmaking practice, so prepare to dive in and make something cool!

[78] https://msdn.microsoft.com/en-us/library/system.enum.getnames(v=vs.110).aspx
[79] https://msdn.microsoft.com/en-us/library/system.environment(v=vs.110).aspx
[80] https://msdn.microsoft.com/en-us/library/system.environment.specialfolder.aspx

Start Here

The `System.Math` class (Google it!) has a ton of static members. In fact, that’s all it has - you can’t actually instantiate the type, because it doesn’t have any constructors. Cool, right? It’s not hard to use already:

```
System.Math::Abs(-5)
```

That’ll return the absolute value of -5, which is 5.

Your Task

We want you to figure out how to use the `Round()` method from `System.Math`. When you do, build a function around it. We’ll suggest a command name of `ConvertTo-RoundNumber`. Your command should:

- Accept a number (of type `[double]`) to be rounded
- Optionally, accept the number (as an `[int]`) of decimal places to round to

Good luck!

Our Take

Here’s what we did (it’s in Solution.ps1 in the downloadable sample code for this chapter):

Solution.ps1

```
function ConvertTo-RoundNumber {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory)]
        [double]$Number,

        [int]$DecimalPlaces
    )

    if ($PSBoundParameters.ContainsKey('DecimalPlaces')) {
        [System.Math]::Round($Number, $DecimalPlaces)
    }
    else {
        [System.Math]::Round($Number)
    }
```

```
}

ConvertTo-RoundNumber -Number 5.55345 -DecimalPlaces 2
ConvertTo-RoundNumber -Number 5.6748
```

Let’s Review

Answer these questions, and you’ll know you picked up the main point of the chapter:

1. What differentiates a static member and an instance member of a type?
2. What does a constructor do?
3. How to you pass arguments to a constructor?
4. How do you determine which constructor will run, when a type has multiples?
5. What one command can be used to load .NET assemblies from any location?

Review Answers

Here are the answers:

1. Static members (shown with a red “S” icon in the docs) don’t require you to instantiate the class; instance members do.
2. Creates an instance of a type.
3. By using the `-ArgumentList` parameter of `New-Object`.
4. .NET figures this out based on the number and data types of your arguments.
5. The `Add-Type` command.

Scripting at Scale

We've always felt that one of PowerShell's greatest strengths was that if you could do something with one thing, be it a file, event log, computer or user account, you could do it for 10, 100 or 1000. In most cases, your PowerShell code would be essentially the same. This notion should also be influencing the way you do your work as an IT Pro.

For the longest time we usually approached our work on a singular basis. Say you had to check free disk space on 10 servers. In the last century, you'd go through your list one at a time and get the data you needed. But today, you should be thinking about ***managing at scale***. Don't think about getting disk space for 1 server at a time, think about how to do it for all 10 *at the same time*. Don't check individual event logs on 100 servers, check them all at once. Once you start looking at your work from this perspective, you'll realize you need to change your tool set or how you are using it. Fortunately, PowerShell makes it relatively easy to take this approach.

However, even with all of that we're also going to offer up this potentially heresy, "**PowerShell isn't always the answer**." If you need to manage 10,000 servers in near real-time, PowerShell is probably not going to be the best tool. We're not saying it won't work, but the scripting effort may be beyond your abilities or performance won't be what you need. PowerShell is always going have overhead, which is not necessarily a bad thing. That "overhead" makes it easy to find and use well-defined commands and parameters in a meaningful pipelined expression. We want you to realize that at some point you may need to move beyond PowerShell to compiled C# applications or full-blown software management solutions.

So what's the point of this chapter? Well, this is for the majority of you that want to use PowerShell to manage more than a few things at a time. When you are building your PowerShell tools, you may want to take large-scale operations into account. To that end we wanted to provide some advice and techniques for scripting at scale but with a few caveats.

Depending on your tool, performance at scale may be influenced by factors outside of your control such as network or server loads, limitations with cmdlets you are using within your PowerShell tool and how a user, maybe even you, are expecting to use it. The best we can say is keep the following things in mind and test.

In this chapter you'll see us use `Measure-Command` quite a bit. But you shouldn't rely on a single test as an absolute metric. Any number of factors could influence the value. Plus, there is often a caching effect which can throw off consecutive test results. You might consider testing in new PowerShell sessions. You can also use Jeff's Test-Expression module which you can find in the PowerShell Gallery.

To Pipeline or not

Without a doubt the pipeline makes PowerShell easy to use. It is pretty easy to run a command like this:

```
Get-Content services-to-test.txt | Get-Service
```

However, you could also have written the expression like this:

```
Get-Service -name (Get-Content services-to-test.txt)
```

For a small list the differences are irrelevant. But once you start scaling, this type of performance difference begins to add up. Using the pipeline will always involve some degree of overhead which is the price we pay for the convenience. Let's look at the cost.

Here's a bunch of service names.

```
$names = Get-Service | Select-Object -expandproperty name
```

Now let's see the difference in how we use it:

```
Measure-Command {
 $names | Get-Service
}

Measure-Command {
 Get-Service $names
}
```

In our test, the first command took 57ms to complete and the latter only 40ms. Not that much really. So let's make a bigger list of names.

```
$big = $names+$names+$names+$names+$names
```

Now we'll re-run the tests with $big instead of $names. Now we're at 253ms vs 169ms which is beginning become noticeable. And we're going to assume that as the amount of data to process goes up the differences will become more noticeable.

All of our talk about scripting at scale assumes you are running your toolset interactively and efficiency is paramount. If the typical usage will be to run your command in background job where you'll get the data when you get around to it, then everything we're covering may not really matter.

Or let's look at a larger pipelined process.

```
$data = Get-ChildItem -path $env:temp -file -Recurse |
Group-Object -property extension |
Sort-Object -property Count |
Select-Object -property Count,Name,
@{Name="Size";Expression = {($_.group | Measure-Object -Property length -sum).sum}}
```

There's nothing inherently wrong with this approach. It works and on our test system with a very cluttered temp folder it took about 860ms. Compare the previous command to this:

```
$files = Get-ChildItem $env:temp -file -Recurse
$grouped = $files | Group-Object -Property extension
$sorted = $grouped | Sort-Object -Property Count
$data = $sorted | Select-Object -Property Count,Name,
@{Name="Size";Expression = {($_.group | Measure-Object -Property length -sum).sum}}
```

The end result is the same, but in this case the commands ran in about 780ms. This version might even be easier to understand.

Let's look at this from a toolmaking perspective. We have a function, which you can find in the chapter's code downloads, to calculate the square root of a number.

SquareRoot.ps1

```
Function SquareRoot {
    [cmdletbinding()]
    Param(
        [Parameter(Position = 0, Mandatory, ValueFromPipeline)]
        [int[]]$Value
    )

    Begin {
        Write-Verbose "[BEGIN  ] Starting: $($MyInvocation.Mycommand)"
    } #begin

    Process {
        foreach ($item in $value) {
            [pscustomobject]@{
                Value      = $item
                SquareRoot = [math]::Sqrt($item)
            }
        }
    } #process

    End {
```

```
        Write-Verbose "[END    ] Ending: $($MyInvocation.Mycommand)"
    } #end
}
```

The command is written so that the user can pipe in a list of numbers or pass the list with the parameter.

```
$n = 1..1000
Measure-Command {$n | squareroot}
Measure-Command {squareroot $n }
```

In this simple comparison we scored 52ms vs 38ms respectively. Let's see the difference with varying sets of numbers between using the pipeline and using the parameter.

```
10,100,500,1000,5000,10000 | ForEach-Object {
 $n = 1..$_
 $pipe = (Measure-Command {$n | squareroot}).totalMilliseconds
 $param = (Measure-Command {squareroot $n}).TotalMilliseconds

 [pscustomobject]@{
  ItemCount = $_
  PipelineMS = $pipe
  ParameterMS = $param
  PctDiff = 100 - (($param/$pipe) * 100 -as [int])
 }
}
```

The results speak for themselves:

```
ItemCount PipelineMS ParameterMS PctDiff
--------- ---------- ----------- -------
       10     1.1204      0.7462      33
      100     3.1137      1.5254      51
      500     16.268       9.499      42
     1000    48.8335      32.456      34
     5000   207.8208     119.046      43
    10000   395.9118    290.6119      27
```

In this case, we may want to consider revising the tool and removing the option of accepting pipelined input, thus forcing the use to pass values with the parameter. Although this might make the tool more difficult for the user who might be expecting to pipe in a set of numbers.

This is not to say you should never use the pipeline in your toolmaking, only that you might want to consider if you are using it wisely.

Foreach vs ForEach-Object

Another scaling factor might be whether you rely on the `Foreach` enumerator or the `ForEach-Object` cmdlet. Let's demonstrate with a large number of items to process.

```
$n = 1..10000
```

Let's do something with each item and measure:

```
Measure-Command {
 $a = 0
 foreach ($i in $n) {
   $a+=$i
 }
}
```

This took about 50ms to complete compared to the alternative:

```
Measure-Command {
 $n | ForEach-Object -Begin { $a = 0 } -process {
 $a+=$_
 }

}
```

Which took about 227ms to get the same result. Again there may be other reasons you might want to use one technique over the other but once you start scaling, there are differences to consider.

Write-Progress

Another design consideration for tools that need to work at scale is user feedback. If you have a long running command, it is often helpful to let the user know what is happening. You could sprinkle a bunch of `Write-Host` commands throughout your function but that'll get ugly pretty quickly. Instead, you should use a cmdlet that doesn't get a lot of love, `Write-Progress`.

You've probably seen a text-style progress bar when running some commands. That comes from `Write-Progress`. The tricky part is that you have to write your code to include it from the beginning. The cmdlet requires at least an *activity* description. Here's a quick one-liner that demonstrates it:

```
1..5 | foreach {
  Write-Progress "Counting"
  Start-Sleep -Seconds 1
}
```

You can also include a *Status* which will display just below it.

```
1..5 | foreach {
 Write-Progress -activity "Counting" -status "Processing"
 Start-Sleep -Seconds 1
}
```

And if you want to get very granular there is a provision to display the *current operation*.

```
1..5 | foreach {
 Write-Progress -activity "Counting" -status "Processing" -currentOperation $_
 Start-Sleep -Seconds 1
}
```

You'll find many of our demos in the chapter download.

Here's more complete example. We've thrown in a `Start-Sleep` command to make it easier to see the progress display.

```
$out = Get-Process | Where-Object starttime | ForEach-Object {

Write-Progress -Activity "Get-Process" `
               -Status "Processing" `
              -CurrentOperation "process: $($_.name) [$($_.id)]"

 $_ | Select-Object ID, Name,StartTime,
@{Name="Runtime";Expression = {(Get-Date) - $_.starttime}}
Start-Sleep -Milliseconds 50
}
```

Now, if you know in advance how much data you need to process, you can provide a time remaining value or a percent complete.

```
$i = 20
1..$i | foreach -Begin {
 [int]$seconds = 21
 } -process {

 Write-Progress -Activity "My main activity" -Status "Calculating square roots" `
 -CurrentOperation "processing:  $_" -SecondsRemaining $seconds
 [math]::Sqrt($_)
 Start-Sleep -Seconds 1
 $seconds-= 1
}
```

It is up to you to come up with code to figure out the seconds remaining value. You don't have to be 100% accurate but "close enough". When the loop finishes, the `Write-Progress` display will automatically dismiss.

There is a `-Completed` switch parameter for the cmdlet which will force the display to disappear, but we've never had to use it.

Maybe it's because of the type of commands we write, but when we use `Write-Progress` we tend to show a percent complete value.

```
$i = 20
1..$i | ForEach-ObjectNotFound -Begin {
  [int]$count = 0
 } -process {
 #calculate percent complete
 $count++
 $pct = ($count/$i) * 100
 Write-Progress -Activity "My main activity"
                -Status "Calculating square roots" `
                -CurrentOperation "processing:  $_" -PercentComplete $pct

 [math]::Sqrt($_)
 Start-Sleep -Milliseconds 200

}
```

Usually when using `Write-Progress` most values like activity and status will remain unchanged or rarely change This is a good use case for splatting a hashtable of parameters. We've included a sample function in the GetFolderSize.ps1 file.

GetFolderSize.ps1

```
Function Get-FolderSize {
    [cmdletbinding()]
    Param(
        [Parameter(
            Position = 0,
            ValueFromPipeline,
            ValueFromPipelineByPropertyName
        )]
        [ValidateNotNullorEmpty()]
        [string]$Path = $env:temp
    )

    Begin {
        Write-Verbose "[BEGIN  ] Starting: $($MyInvocation.Mycommand)"
    } #begin

    Process {
        Write-Verbose "[PROCESS] Analyzing: $path"

        #define hash table of parameter values for Write-Progress
        $progParam = @{
            Activity         = $MyInvocation.MyCommand
            Status           = "Querying top level folders"
            CurrentOperation = $path
            PercentComplete  = 0
        }

        Write-Progress @progParam

        Write-Verbose "[PROCESS] Get top level folders"
        $top = Get-ChildItem -Path $path -Directory

        #sleeping enough to see the first part of Write-Progress
        Start-Sleep -Milliseconds 300

        #initialize a counter
        $i = 0

        #get the number of files and their total size for each
        #top level folder
        foreach ($folder in $top) {
```

```
            #calculate percentage complete
            $i++
            [int]$pct = ($i/$top.count)*100

            #update the param hashtable
            $progParam.CurrentOperation = "Measuring folder size: $($folder.Name)"
            $progParam.Status = "Analyzing"
            $progParam.PercentComplete = $pct

            Write-Progress @progParam

            Write-Verbose "[PROCESS] Calculating folder statistics for $($folder.name)\
."
            $stats = Get-ChildItem -path $folder.fullname -Recurse -File |
            Measure-Object -Property Length -Sum -Average
            if ($stats.count) {
                 $fileCount = $stats.count
                 $size = $stats.sum
             }
            else {
                 $fileCount = 0
                 $size = 0
             }
            #write a custom object result to the pipeline
            [pscustomobject]@{
                  Path     = $folder.fullName
                  Modified = $folder.LastWriteTime
                  Files    = $fileCount
                  Size     = $Size
                  SizeKB   = [math]::Round($size/1KB, 2)
                  SizeMB   = [math]::Round($size/1MB, 2)
                  Avg      = [math]::Round($stats.average, 2)
            }
        } #foreach
    } #process

    End {
        Write-Verbose "[END    ] Ending: $($MyInvocation.Mycommand)"
    } #end
}
```

In the beginning you can see where we defined a hashtable of values for `Write-Progress`.

```
$progParam=@{
        Activity = $MyInvocation.MyCommand
        Status = "Querying top level folders"
        CurrentOperation = $path
        PercentComplete = 0
    }
```

Later, as the script processes folders we can update the hashtable on the fly.

```
#calculate percentage complete
$i++
[int]$pct = ($i/$top.count)*100

#update the param hashtable
$progParam.CurrentOperation = "Measuring folder size: $($folder.Name)"
$progParam.Status = "Analyzing"
$progParam.PercentComplete = $pct

Write-Progress @progParam
```

You can change the progress bar color scheme in the console by modifying `$host.privatedata.progressforegroundcolor` and `$host.privatedata.progressbackgroundcolor`. Use the same values you'd use with `Write-Host`.

For your commands that need to process a lot of data, or might take a bit longer to run, using `Write-Progress` will make you look like a real professional.

Leverage Remoting

Perhaps the idea of scripting at scale is most important when your tool needs to process hundreds or thousands of computers. PowerShell cmdlets make it easy to pass in multiple computer names, but there are some things you might want to consider.

When you see a cmdlet with a `-Computername` parameter, more than likely that command is connecting to each computer sequentially over legacy protocols like RPC and DCOM. If one of the computers is slow to respond or offline, you can't get to the rest of the list until it responds or errors out. We did a quick test with 5 computers we knew to be online.

```
Measure-Command {
Get-Service bits,wuauserv -ComputerName $computers
}
```

This took about 917ms. We're working with the assumption that as the number of computers increases the time required will increase proportionally. Compare this to running the same `Get-Service` command but this time using `Invoke-Command` which means it runs essentially simultaneously on all the computers.

```
Measure-Command {
Invoke-Command {Get-Service bits,wuauserv} -ComputerName $computers
}
```

This version took a bit longer at 1348ms, primarily because of the overhead in setting up and tearing down a PSSession. But let's say you already had a PSSession created.

```
$ps = New-PSSession -ComputerName $computers
Measure-Command {
Invoke-Command {Get-Service bits,wuauserv} -session $ps
}
```

Now the command completes in 161ms! And this improves if your list of computers contains items that are offline or otherwise might error out. We added an offline computer to the list and re-ran the first test. That took over 6 seconds to complete. But using `Invoke-Command` and the computername took 581ms AND we got an error we could have handled. Or course, if you use a PSSession you know that `Invoke-Command` will run without error.

Our tests with 5 computers hardly posed any sort of impact on the network. But what if we were querying 50 or 500 computers? By default `Invoke-Command` will throttle connections to 32 at a time. That means if we gave it 50 computers, it would make a connection to the first 32, then as servers responded, the remaining list would be processed up to a max of 32 at a time. You can raise or lower this limit with the `-ThrottleLimit` parameter.

So what does this mean with your toolmaking?

Assuming your underlying code relies on remoting anyway, you might consider running that code through `Invoke-Command`. For example, look at this simple function that gets drive info:

GetDiskSpace.ps1

```
Function Get-DiskSpace {
[cmdletbinding()]
Param(
[Parameter(Position = 0, Mandatory)]
[string[]]$Computername
)

Invoke-Command -scriptblock {
 Get-CimInstance -ClassName win32_logicaldisk -filter "deviceid='c:'" |
 Select-Object -property @{Name="Computername";Expression={$_.SystemName}},
 DeviceID,Size,Freespace,
 @{Name="PctFree";Expression={ "{0:p2}" -f $($_.freespace/$_.size)}}
} -ComputerName $computername -HideComputerName |
Select-Object -Property * -ExcludeProperty RunspaceID
```

Or if you need to handle errors with offline computers or access issues you could use something like this version:

With Error Handling

```
Function Get-DiskSpace {
[cmdletbinding()]
Param(
[Parameter(Position = 0, Mandatory)]
[string[]]$Computername
)

foreach ($computer in $computername) {
  Write-Verbose "Querying $computer"
  Try {
    Invoke-Command -scriptblock {
     Get-CimInstance -ClassName win32_logicaldisk -filter "deviceid='c:'" |
     Select-Object -Property @{Name="Computername";Expression={$_.SystemName}},
     DeviceID,Size,Freespace,
     @{Name="PctFree";Expression={ "{0:p2}" -f $($_.freespace/$_.size)}}
    } -ComputerName $computer -HideComputerName -ErrorAction stop  |
     Select-Object -Property * -ExcludeProperty RunspaceID
   }
  Catch {
     Write-Warning "[$($computer.toupper())] $($_.exception.message)"
  }
```

```
} #foreach
}
```

Need to support pipelining a bunch of computer names? You might consider this variation.

From the pipeline

```
Function Get-DiskSpace {
[cmdletbinding()]
Param(
[Parameter(Position = 0, Mandatory, ValueFromPipeline)]
[string[]]$Computername
)

Begin {
    #initialize an array
    $computers=@()
}

Process {
    #add each computer to the array
    $computers+=$Computername
}

End {
    #run the actual command here for all computers
    Invoke-Command -scriptblock {
     Get-CimInstance -ClassName win32_logicaldisk -filter "deviceid='c:'" |
     Select-Object -Property @{Name="Computername";Expression={$_.SystemName}},
     DeviceID,Size,Freespace,
     @{Name="PctFree";Expression={ "{0:p2}" -f $($_.freespace/$_.size)}}
    } -ComputerName $computers -HideComputerName  |
    Select-Object -Property * -ExcludeProperty RunspaceID
}
}
```

All of the work is done at once in the `End` scriptblock.

Notice that in all cases, we're doing as much processing, such as selecting properties, *on the remote computer* to take advantage of its processing resources and to limit what has to come back across the wire.

If your function is running a single command in a remoting session, there's no advantage to creating a session, running `Invoke-Command` and then removing the session. But if you are doing something that requires multiple commands on a remote server, then we recommend creating a PSSession and re-using that as necessary. Just remember to clean it up at then end.

Leverage Jobs

Another option for scaling your commands might be to take advantage of PowerShell's background job infrastructure. Certainly anyone should be able to run your tool with `Start-Job` but perhaps you'd like to make this easier or you know your commands will take a long, long time to complete and jobs make sense.

Here's a version of the diskspace function that simply passes the `-AsJob` parameter to the underlying `Invoke-Command`.

AsJob

```
Function Get-DiskSpace {
[cmdletbinding()]
Param(
[Parameter(Position = 0, Mandatory, ValueFromPipeline)]
[string[]]$Computername,
[switch]$AsJob
)

Begin {
    #initialize an array
    $computers=@()
}

Process {
    #add each computer to the array
    $computers+=$Computername
}

End {
    #add a parameter
    $psboundParameters.Add("HideComputername",$True)

    #run the actual command here for all computers
    Invoke-Command -scriptblock {
     Get-CimInstance -ClassName win32_logicaldisk -filter "deviceid='c:'" |
```

```
     Select-Object -Property @{Name="Computername";Expression={$_.SystemName}},
     DeviceID,Size,Freespace,
     @{Name="PctFree";Expression={ "{0:p2}" -f $($_.freespace/$_.size)}}
    } @psboundParameters |
     Select-Object -Property * -ExcludeProperty RunspaceID
 }
}
```

One potential "gotcha", that you'd have to document or train, is that if there are errors, the job might show as failed but there will be results from computers where it was successful.

Or you might want to internally spin off a bunch of jobs in order scale. Here's a template of what such a function might look like:

Job Template

```
Function Get-Foo {
    [cmdletbinding()]
    Param(
        [Parameter(Position = 0, ValueFromPipeline)]
        $This,
        $That,
        $TheOtherThing
    )

    Begin {
      #initialize an array to hold job objects
      $jobs = @()

      $mycode = {
         #define your code to run with parameters if necessary
         #parameters will need to be passed positionally
         Param($this, $that)
         #awesome PowerShell code goes here
        }
    }

    Process {
      #add the job to the array
      $jobs += Start-Job -ScriptBlock $mycode -ArgumentList $this, $that
    }

    End {
     #wait for all jobs to complete
```

```
    Write-Host "Waiting for background jobs to complete" -ForegroundColor Yellow
    $jobs | Wait-Job

    #receive job results
    #or bring job results back in to the function and do
    #something with them
    $jobs |
    Get-Job -ChildJobState Completed -HasMoreData $True |
    Receive-Job -keep
   }

}
```

Consider this nothing more than a starting point and we've included this in the chapter downloads.

Leverage Runspaces

Background jobs are convenient but there is a price to pay. Although by this point in the book we hope you realize everything in PowerShell toolmaking is a trade-off. One final option for scripting at scale is the use of a PowerShell *runspace*. Frankly, we were a little hesitant in covering this topic as it is advanced stuff and borders on .NET systems programming. But the concept comes up often enough that we figured we'd at least get you started, with the caveat that using runspaces should be for exceptional situations and *not* the norm.

Before we dive into the gnarly details let's get some context. We built a list of 85 computernames and ran `Test-WSMan` through a few different approaches and used `Measure-Command`.

```
$all = foreach ($item in $computers) {
    Test-WSMan $item
}
```

Testing sequentially took 4 minutes and 45 seconds.

```
Measure-Command {
$all=@()
$all+= foreach ($item in $computers) {
 Start-Job {Test-WSMan $item}
}
$all | Wait-job | Receive-Job -Keep
}
```

Using background jobs was a bit faster at 4 minutes 32 seconds. Using runspaces we were able to test in about 18 seconds. That probably got your attention. Here's what we did.

First, you need to create a runspace object.

```
$run = [powershell]::Create()
```

The runspace is basically an empty PowerShell session that you fill with commands and scripts. We added the `Test-WSMan` cmdlet and the computername parameter.

```
$run.AddCommand("Test-WSMan").addparameter("computername",$env:computername)
```

The main reason to use runspaces is the ability to run commands asynchronously. In other words, we can very quickly spin off a runspace, even faster than a background job. This will require using the `BeginInvoke()` method.

```
$handle = $run.beginInvoke()
```

You can test this handle to see if the task is completed with `$handle.IsCompleted`. If so, stop the asynchronous process by invoking `EndInvoke()` with the handle object.

```
$results = $run.EndInvoke($handle)
```

This will give you the command results. The last step you should do is clean up after yourself.

```
$run.Dispose()
```

As you can tell, there's a lot of .NET stuff here. But if your comfortable with that, you can come up with code like we did to test all 85 computers.

```
#initialize an array to hold runspaces
$rspace = @()

#create a runspace for each computer
foreach ($item in $computers) {
    $run = [powershell]::Create()
    $run.AddCommand("Test-WSMan").addparameter("computername",$item)
    $handle = $run.beginInvoke()
    #add the handle as a property to make it easier to reference later
    $run | Add-Member -MemberType NoteProperty -Name Handle -Value $handle
    $rspace+=$run
}
```

```
While (-Not $rspace.handle.isCompleted) {
    #an empty loop waiting for everything to complete
}

#get results
$results=@()
for ($i = 0;$i -lt $rspace.count;$i++) {
    $results+= $rspace[$i].EndInvoke($rspace[$i].handle)
}

#cleanup
$rspace.ForEach({$_.dispose()})
```

For an interesting take on working with runspaces take a look at https://smsagent.wordpress.com/2017/02/17/powershell-tip-create-background-jobs-with-a-custom-class/[81]. You'll find some great tutorials at https://devblogs.microsoft.com/scripting/beginning-use-of-powershell-runspaces-part-1/[82]. There is also the popular PoshRSJob module in the PowerShell Gallery which might help you out.

With this in mind here's a version of the Get-DiskSpace function that uses runspace.

Using Runspaces

```
Function Get-DiskSpace {
[cmdletbinding()]
Param(
[Parameter(Position = 0, Mandatory)]
[string[]]$Computername
)

$rspace = @()

foreach ($computer in $computername) {
Write-Verbose "Creating runspace for $Computer"
  $run = [powershell]::Create()
  [void]$run.AddCommand("Get-CimInstance").addparameter("computername",$computer)
  [void]$run.Commands[0].AddParameter("classname","win32_logicaldisk")
  [void]$run.commands[0].addParameter("filter","deviceid='c:'")
  $handle = $run.beginInvoke()
```

[81] https://smsagent.wordpress.com/2017/02/17/powershell-tip-create-background-jobs-with-a-custom-class/
[82] https://devblogs.microsoft.com/scripting/beginning-use-of-powershell-runspaces-part-1/

```
  #add the handle as a property to make it easier to reference later
  $run | Add-member -MemberType NoteProperty -Name Handle -Value $handle
  $rspace+=$run
} #foreach

#wait for everything to complete
While (-Not $rspace.handle.isCompleted) {
  #an empty loop waiting for everything to complete
}

Write-Verbose "Getting results"
$results=@()
for ($i = 0;$i -lt $rspace.count;$i++) {
  #stop each runspace
  $results+= $rspace[$i].EndInvoke($rspace[$i].handle)
}
Write-Verbose "Cleaning up runspaces"
$rspace.ForEach({$_.dispose()})

Write-Verbose "Process the results"
$Results |
Select-Object -property @{Name="Computername";Expression={$_.SystemName}},
DeviceID,Size,Freespace,
@{Name="PctFree";Expression={ "{0:p2}" -f $($_.freespace/$_.size)}}

}
```

There's no guarantee that this approach is any faster. We ran the very first version that used `Invoke-Command` with our list of 85 computer names in just a bit over 26 seconds. Using the runspace version took 1 minute 19 seconds so perhaps it isn't the right choice for this particular task.

If you are using PowerShell 7 you can use `ForEach-Object` with it's `-Parallel` parameter. But don't get carried away. As we've pointed out there is overhead in setting up and tearing down the runspaces. You want to make sure that the commands you want to run warrant the overhead.

Design Considerations

Is your head spinning yet? There's a lot to digest here and no absolute answers. But we can give you some design guidelines.

- Do you have to even worry about scale?
- How will people use your tool? Will they expect to pipe stuff in or pass values through parameters?
- What is your scripting skill level?
- What version of PowerShell is running or available to you?
- Do you need to handle other requirements such as credentials?
- What is an acceptable performance window? Is it really that big a deal if something takes 1 minute versus 45 seconds?

Everything in toolmaking is a balancing act and trade-off. Only you can decide what approach will work best in your environment. Using tools like `Measure-Command` can help. Or take advantage of the expertise in the PowerShell.org forum and solicit feedback on your project.

Your Turn

Let's see how much you've picked up in this chapter by re-visiting a PowerShell tool from an earlier chapter.

Start Here

In the chapter on creating Basic Controller Scripts and Menus we looked at a process script that checked for recent eventlog entries on remote servers and created an HTML report. That version ran through the list of computers sequentially which could potentially take a long time to run. We've included a copy in this chapter's downloads called `GetEventlogs-Start.ps1`.

Your Task

Modify it to perform better at scale using content from this chapter as inspiration. If you have a bunch of computers you can test with, measure how long each version takes to complete.

Our Take

We ran the starting version against a list of 10 computers, some of which we knew would fail and script took 2 minutes and 22 seconds to complete. Then we tested with this version.

GetEventLogs.ps1

```
#Requires -version 5.0

[cmdletbinding()]
Param(
    [Parameter(Position = 0, Mandatory)]
    [ValidateNotNullorEmpty()]
    [string[]]$Computername,

    [ValidateSet("Error", "Warning", "Information", "SuccessAudit",
    "FailureAudit")]
    [string[]]$EntryType = @("Error", "Warning"),

    [ValidateSet("System", "Application", "Security",
        "Active Directory Web Services", "DNS Server")]
    [string]$Logname = "System",

    [datetime]$After = (Get-Date).AddHours(-24),

    [Alias("path")]
    [ValidateScript({Test-Path $_})]
    [string]$OutputPath = "."
)

#define a hashtable of parameters for Write-Progress

$progParam = @{
 Activity         = $MyInvocation.MyCommand
 Status           = "Gathering $($EntryType -join ",") entries from $logname after $\
after."
 CurrentOperation = $null
}

Write-Progress @progParam

#invoke the command remotely as a job
$jobs = @()
foreach ($computer in $computername) {
    $progParam.CurrentOperation = "Querying: $computer"
    Write-Progress @progParam
    $jobs += Invoke-Command {
      $logParams = @{
       LogName = $using:logname
```

```
        After = $using:after
        EntryType = $using:entrytype
       }
         Get-EventLog @logParams
    } -ComputerName $Computer -AsJob
} #foreach

do {
    $count = ($jobs | Get-Job | Where-Object state -eq 'Running').count
    $progParam.CurrentOperation = "Waiting for $count remote commands to complete"
    Write-Progress @progParam
} while ($count -gt 0)

$progParam.CurrentOperation = "Receiving job results"
Write-Progress @progParam
$data = $jobs | Receive-Job

if ($data) {
    $progParam.CurrentOperation = "Creating HTML report"
    Write-Progress @progparam

    #create html report
    $fragments = @()
    $fragments += "<H1>Summary from $After</H1>"
    $fragments += "<H2>Count by server</H2>"
    $fragments += $data | Group-Object -Property Machinename |
    Sort-Object -property Count -Descending |
    Select-Object -property Count, Name |
    ConvertTo-Html -As table -Fragment

    $fragments += "<H2>Count by source</H2>"
    $fragments += $data | Group-Object -Property source |
    Sort-Object Count -Descending |
    Select-Object -property Count, Name |
    ConvertTo-Html -As table -Fragment

    $fragments += "<H2>Detail</H2>"
    $fragments += $data |
    Select-Object -property Machinename, TimeGenerated, Source, EntryType,
    Message | ConvertTo-Html -as Table -Fragment

    # the here string needs to be left justified
    $head = @"
```

```
<Title>Event Log Summary</Title>
<style>
h2 {
width:95%;
background-color:#7BA7C7;
font-family:Tahoma;
font-size:10pt;
font-color:Black;
}
body { background-color:#FFFFFF;
       font-family:Tahoma;
       font-size:10pt; }
td, th { border:1px solid black;
         border-collapse:collapse; }
th { color:white;
     background-color:black; }
table, tr, td, th { padding: 2px; margin: 0px }
tr:nth-child(odd) {background-color: lightgray}
table { width:95%;margin-left:5px; margin-bottom:20px;}
</style>
"@
    $convert = @{
     Body = $fragments
     PostContent = "<h6>$(Get-Date)</h6>"
     Head = $head
    }
    $html = ConvertTo-Html @convert

    #save results to a file
    $file = "$(Get-Date -UFormat '%Y%m%d_%H%M')_EventlogReport.htm"
    $filename = Join-Path -Path $OutputPath -ChildPath $file

    $progparam.CurrentOperation = "Saving file to $filename"
    Write-Progress @progParam
    Start-Sleep -Seconds 1

    Set-Content -Path $filename -Value $html -Encoding Ascii
    #write the result file to the pipeline
    Get-Item -Path $filename

} #if data
else {
    Write-Host "No matching event entries found." -ForegroundColor Magenta
```

```
}

#clean up jobs if any
if ($jobs) {
    $jobs | Remove-Job
}
```

You'll see that we took advantage of `Write-Progress` to keep the user informed and `Invoke-Command` to run the event log queries remotely. We decided to run the remote commands as background jobs so that we could keep track of how many computers we were waiting on. Using the same list of computers this script completed in under 25 seconds!

Let's Review

Did you learn anything in this chapter?

1. What is a good alternative to `Write-Host` for providing feedback to a user of your tool?
2. What cmdlet should you use to evaluate performance?
3. Is using a pipeline expression always the best solution?
4. You should always use the Foreach enumerator instead of `ForEach-Object`. True or False?
5. What are some of the potential disadvantages of using commands with a `-Computername` parameter?

Review Answers

Did you come up with answers like these?

1. `Write-Progress`
2. `Measure-Command`
3. Ok. This was kind of a trick question. Performance wise, there is always overhead when using the pipeline, especially with large number of objects. But it may make the most logical sense in using your tool.
4. Sorry, another tricky one. This also depends. The cmdlet makes it easier to write to the pipeline and you get the Begin, Process and End scriptblocks but the enumerator often performs faster. I guess that answer then is False.
5. These cmdlets tend to connect with legacy protocols such as RPC and DCOM which aren't necessarily fast or firewall friendly. Plus, parameter values are processed sequentially which may means you can only get one result at a time.

Scaffolding a Project with Plaster

By this point, you shouldn't be thinking about your PowerShell code so much in terms of "scripts" as you are in terms of "projects." Your code is going to need a lot more than just a .psm1 file - there'll be a manifest, automated unit tests, Visual Studio Code configuration files, and lots more. We find that a lot of people skip some of these professional-grade "extras" in their eagerness to "just get scripting," which, if we're being honest, we do too, a lot of the time. Plaster is an open-source PowerShell tool that's designed to help you do things the right way, without slowing you down. It helps *scaffold* a project. That is, it's designed to create a complete, pro-grade "structure" for a project, so that all the right things are in all the right places and you can focus on writing your code quickly. It'll create the right folders for help files (PlatyPS!), unit tests (Pester!), and more, all based on customizable templates.

Getting Started

The first thing you need to do is install the latest version of Plaster from the PowerShell Gallery.

```
Install-Module Plaster
```

Plaster is still in development and is an open source project. If you encounter issues or wish to learn more, head to the project's Github repository at https://github.com/PowerShellOrg/Plaster[83]. The current version of the module only has a handful of commands.

```
PS C:\> get-command -module plaster

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Function        Get-PlasterTemplate                                1.1.3      plaster
Function        Invoke-Plaster                                     1.1.3      plaster
Function        New-PlasterManifest                                1.1.3      plaster
Function        Test-PlasterManifest                               1.1.3      plaster
```

We'll take a look at these commands throughout the chapter.

Plaster was created by Microsoft and maintained by them for a number of years. In June of 2020 project ownership was transferred to PowerShell.org.

[83] https://github.com/PowerShellOrg/Plaster

Plaster Fundamentals

Plaster works by parsing an XML manifest (think of it as a template) that you create which in turn generates a file and folder structure for your commands and module. Creating the template is the most time-consuming task, but once completed it makes spinning up new projects incredibly easy to do in a consistent manner.

Plaster's concept is that you set up a template folder structure and an XML manifest file. When you invoke the manifest, Plaster will create a new folder, copying and creating files or folders as needed. The great feature in Plaster is that everything happens dynamically based on information gathered from the template. Let's walk through the process.

Invoking a Plaster Template

Before we create our own it might help to see the Plaster in action. To invoke Plaster we need the path to a Plaster manifest or template file. Running `Get-PlasterTemplate` will show available templates.

```
PS C:\> Get-PlasterTemplate


Title        : AddPSScriptAnalyzerSettings
Author       : Plaster project
Version      : 1.0.0
Description  : Add a PowerShell Script Analyzer settings file to the root of your wo\
rkspace.
Tags         : {PSScriptAnalyzer, settings}
TemplatePath : C:\Program Files\WindowsPowerShell\Modules\Plaster\1.1.3\Templates\Ad\
dPSScriptAnalyzerSettings

Title        : New PowerShell Manifest Module
Author       : Plaster
Version      : 1.1.0
Description  : Creates files for a simple, non-shared PowerShell script module.
Tags         : {Module, ScriptModule, ModuleManifest}
TemplatePath : C:\Program Files\WindowsPowerShell\Modules\Plaster\1.1.3\Templates\Ne\
wPowerShellScriptModule
```

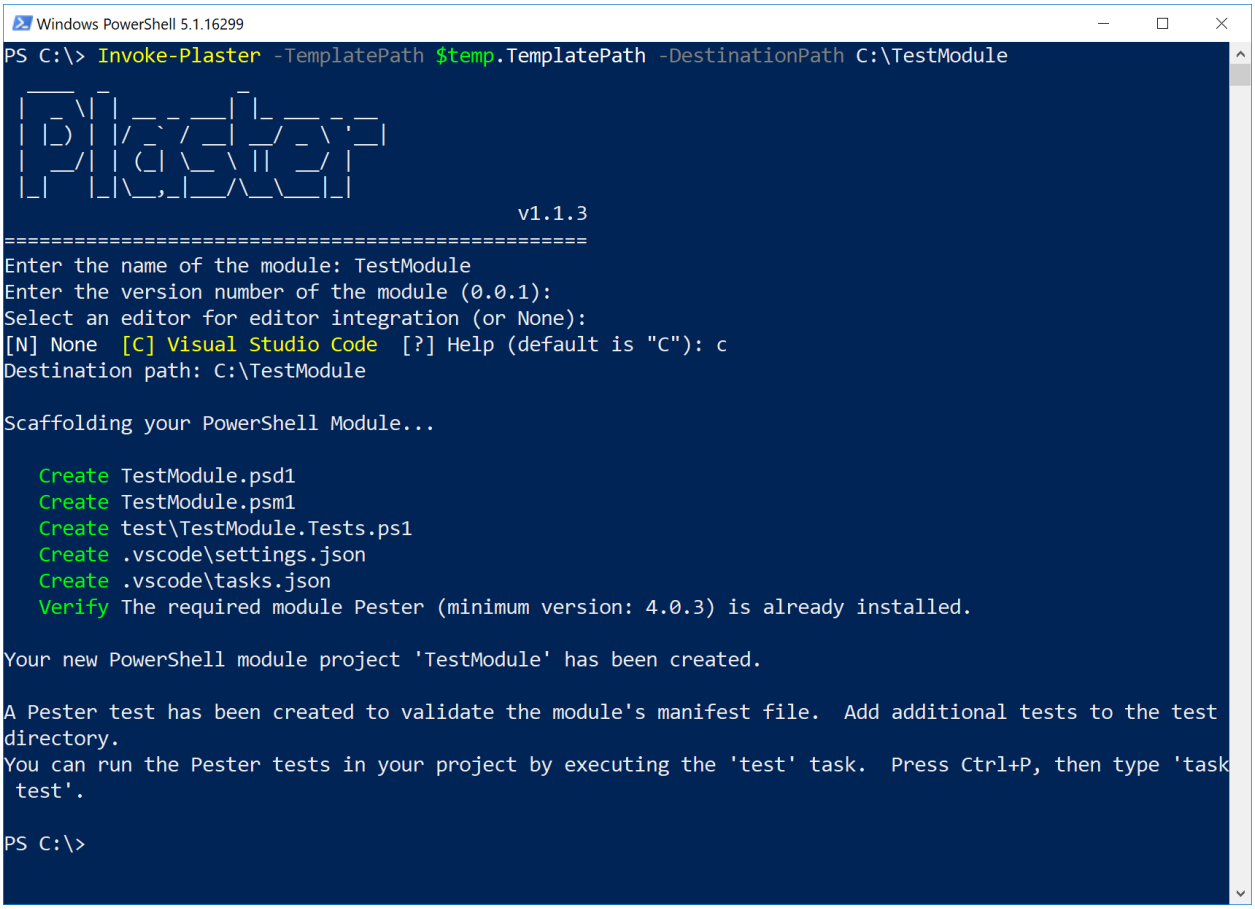
These are the templates included with the Plaster module. The important piece of information you need is the TemplatePath property. Let's use the module template.

```
PS C:\> $temp = Get-PlasterTemplate | select-object -last 1
PS C:\> dir $temp.TemplatePath


    Directory: C:\Program
    Files\WindowsPowerShell\Modules\Plaster\1.1.3\Templates\NewPowerShellScriptModule


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        1/30/2018  12:25 PM                editor
d-----        1/30/2018  12:25 PM                test
-a----       10/27/2017   6:10 AM            323 Module.psm1
-a----       10/27/2017   6:10 AM           3129 plasterManifest.xml
```

To make it easy we'll save it to a variable. Now we can invoke it using `Invoke-Plaster`. We'll need to supply the path to the template file and a destination for our new module.

```
PS C:\> Invoke-Plaster -TemplatePath $temp.TemplatePath -DestinationPath C:\TestModu\
le
  ____  _           _
 |  _ \| | __ _ ___| |_ ___ _ __
 | |_) | |/ _` / __| __/ _ \ '__|
 |  __/| | (_| \__ \ ||  __/ |
 |_|   |_|\__,_|___/\__\___|_|
                                                 v1.1.3
================================================
Enter the name of the module:
```

The first thing we get is a prompt for the name of our module. We'll call it TestModule to match the destination folder. Plaster will then prompt us for additional information.



Invoking a Plaster Manifest

Within a matter of seconds, a new module directory has been created, complete with the beginnings of a Pester test.

```
PS C:\> dir C:\TestModule\


    Directory: C:\TestModule


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         2/8/2018   5:42 PM                .vscode
d-----         2/8/2018   5:42 PM                test
-a----         2/8/2018   5:42 PM           3867 TestModule.psd1
-a----       10/27/2017   6:10 AM            323 TestModule.psm1
```

How did all of this work and how can you make it work for you? This is where the real fun begins.

Creating a Plaster Module Template

The first step is to create a simple manifest using the `New-PlasterManifest` cmdlet. You will need to provide a name for the template and ideally a description. You also should specify the path. This is where the manifest XML file will be created. If you don't specify a path everything goes into the current directory. You should create the directory before creating the manifest

```
PS C:\> New-PlasterManifest -TemplateName MySample -TemplateType Project -Author "Ar\
t Deco" -Description "my sample template" -Path C:\mySample\plastermanifest.xml
```

The Plaster manifest is always called `plastermanifest.xml`. All we did was create it in our new directory with the file.

```
PS C:\> dir .\mySample\


    Directory: C:\mySample


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         2/8/2018   5:29 PM            511 plastermanifest.xml
```

The only thing in this file is the Plaster *metadata*.

```
<?xml version="1.0" encoding="utf-8"?>
<plasterManifest
  schemaVersion="1.1"
  templateType="Project" xmlns="http://www.microsoft.com/schemas/PowerShell/Plaster/\
v1">
  <metadata>
    <name>MySample</name>
    <id>c6cc56bb-e3cc-4af5-a38b-d97b9649ecef</id>
    <version>1.0.0</version>
    <title>MySample</title>
    <description>my sample template</description>
    <author>Art Deco</author>
    <tags></tags>
  </metadata>
  <parameters></parameters>
  <content></content>
</plasterManifest>
```

The only items you might want to change going forward are the version or description. As it stands now this Plaster manifest doesn't do anything. It needs some content. While you can specify content with the `New-PlasterManifest` command, we think you'll find it easier to open the file in your scripting editor. Remember, that this is an xml file so watch the case in your tags.

Adding Prompts

As you saw when we ran the sample template, Plaster can prompt you for key pieces of information. We can do the same thing by defining entries in the `<parameters></parameters>` section. First, we'll prompt for the name of the module.

```
 <parameter name='ModuleName' type='text' prompt='Enter the name of the module'/>
```

It might also be helpful to include a version.

```
<parameter name='Version' type='text' prompt='Enter the initial module version' defa\
ult = '0.1.0'/>
```

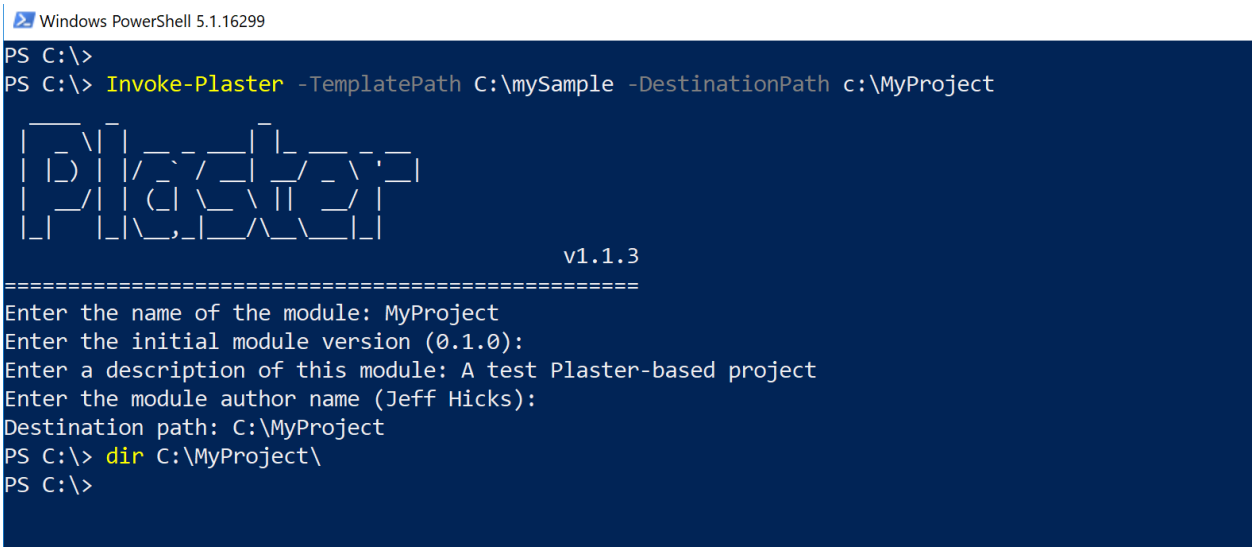
Notice that with this parameter we are also including a default value. We'll also prompt the an author name and description.

```
    <parameter name='Description' type='text' prompt='Enter a description of this mo\
dule'/>
    <parameter name="ModuleAuthor" type='user-fullname' prompt='Enter the module aut\
hor name'/>
```

The 'user-fullname' type will use the name associated with your git configuration. Here's the current state of the Plaster manifest.

The Plaster Manifest with Parameters

If we run Plaster we can see the prompts in action.



Testing Plaster Parameters

As you can tell from the PowerShell session, the manifest didn't do anything because we haven't defined any actual content. That is, we haven't told it what files or folders to create or copy.

Adding Files and Folders

Since we are creating a new module, we most likely need a module manifest. Plaster can create that file for us with an XML declaration like this:

```
<newModuleManifest
 destination='${PLASTER_PARAM_ModuleName}.psd1'
 moduleVersion = '$PLASTER_PARAM_Version'
 rootModule = '${PLASTER_PARAM_ModuleName}.psm1'
 encoding = 'UTF8-NoBOM'
 author = '$PLASTER_PARAM_ModuleAuthor'
 description = '$PLASTER_PARAM_Description'
 openInEditor = "true"
 />
```

The newModuleManifest tag is essentially a proxy for the `New-ModuleManifest` cmdlet. You'll also notice that Plaster has a way for us to pass the parameter values. The format is $PLASTER_PARAM_-[your parameter name].

Next, we might want a consistent folder structure. We can instruct Plaster to create new folders for us. Because we intend to create help documentation with Platyps we'll create the necessary folders.

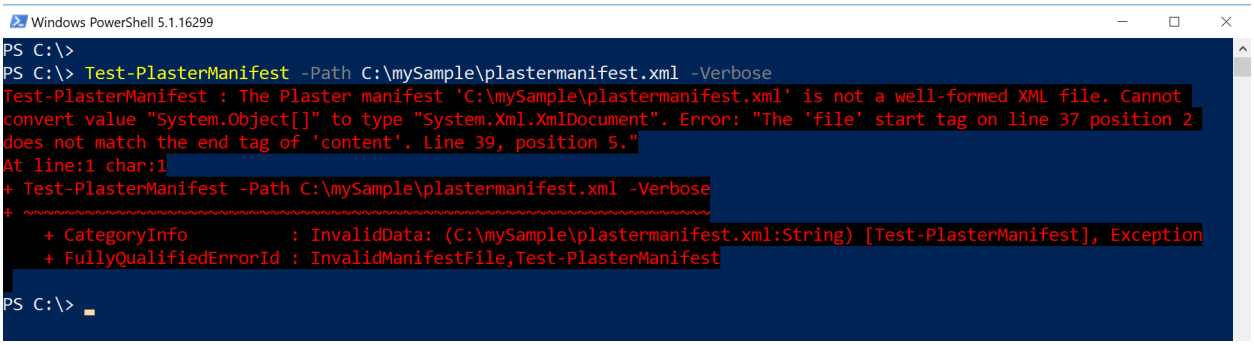
```
<file destination='docs' source=''/>
<file destination='en-us' source=''/>
```

However the Plaster philosophy is to have a model folder that you can build from. Any files and folders you want to reference are relative to the plastermanifest.xml file. We'll add a few files to the directory and update the Plaster manifest.

```
<file source='Module.psm1' destination='${PLASTER_PARAM_ModuleName}.psm1'/>
<file source='changelog.txt' destination='changelog.txt'/>
<file source='README.md' destination='README.md'/>
<file source='license.txt' destination='license.txt'>
```

Notice how we are using Plaster parameters to change the name of of module.psm1 file.

Before we see how this works so far, and because we're working with XML which can be picky, let's test the manifest.



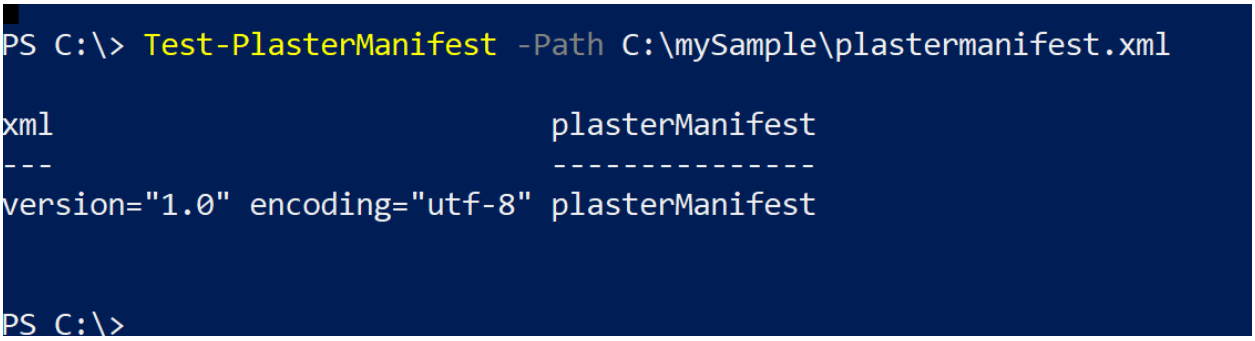
Testing the Plaster Manifest

Sure enough we goofed and it is a common mistake. Looking at the file in VS Code we see that we forgot to close a tag.

```
<file source='Module.psm1' destination='${PLASTER_PARAM_ModuleName}.psm1'/>
<file source='changelog.txt' destination='changelog.txt'/>
<file source='README.md' destination='README.md'/>
<file source='license.txt' destination='license.txt'>
```

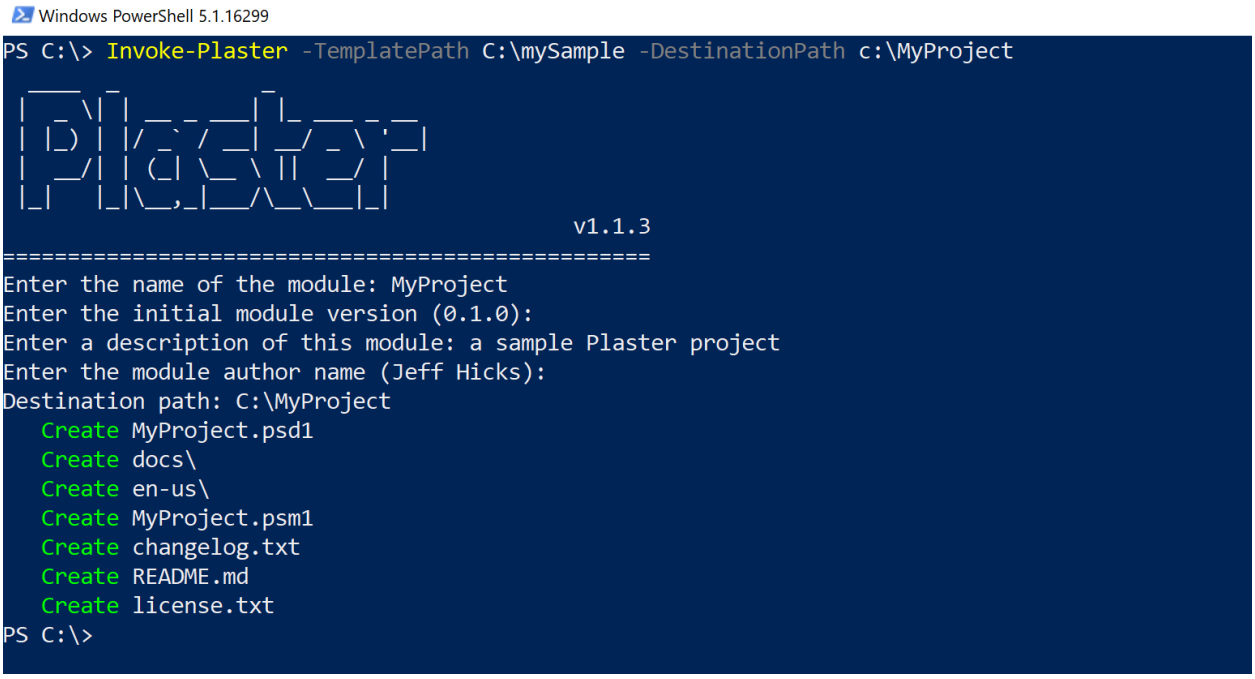
Manifest XML Error

We fix the error, save the file and re-run the test. Now, there are no errors.



Successful Plaster Manifest Test

Let's invoke the manifest and see what happens.



Invoking a Custom Plaster Template

Checking the project folder we specified we see the new files and folders.

```
PS C:\> dir .\MyProject\


    Directory: C:\MyProject


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        2/12/2018   4:30 PM                docs
d-----        2/12/2018   4:30 PM                en-us
-a----        2/12/2018   4:12 PM             14 changelog.txt
-a----        2/12/2018   4:16 PM           2198 license.txt
-a----        2/12/2018   4:30 PM           3906 MyProject.psd1
-a----        2/12/2018   4:16 PM            145 MyProject.psm1
-a----        2/12/2018   4:13 PM             16 README.md


PS C:\>
```

Your source folder can be as complex as you need it to be. You configure the Plaster template to create and copy files as needed.

Using Template Files

If copying PSl files was all Plaster did, that still puts you ahead. But that is just the beginning. Plaster also has the ability to create dynamic content. That is, content based on parameter values or other conditions. If you read through the Plaster documentation on GitHub, you'll learn that you can modify files using regular expressions. However, we think you'll want to use template files.

In a template, you define sections of the file as replaceable parameters wrapped in `<% %>` tags. Plaster will replace the contents with corresponding parameter value. We're going to add a Test folder to be copied with the beginnings of a Pester test. But we want the final file to have the module name. Here's the template file.

```
$ModuleManifestName = '<%=$PLASTER_PARAM_ModuleName%>.psd1'
$ModuleManifestPath = "$PSScriptRoot\..\$ModuleManifestName"

Describe '<%=$PLASTER_PARAM_ModuleName%> Manifest Tests' {
    It 'Passes Test-ModuleManifest' {
        Test-ModuleManifest -Path $ModuleManifestPath | Should Not BeNullOrEmpty
        $? | Should Be $true
    }
}
```

In the manifest, we need to tell Plaster to use this file. Instead of the 'file' directive we use 'templatefile'.

```
<templateFile source='test\Module.T.ps1' destination='test\${PLASTER_PARAM_ModuleNam\
e}.Tests.ps1' />
```

Be very careful here as 'templateFile' is case-sensitive.

We'll save the manifest and re-run the template. The Pester test file now looks like this:

```
$ModuleManifestName = 'myProject.psd1'
$ModuleManifestPath = "$PSScriptRoot\..\$ModuleManifestName"

Describe 'myProject Manifest Tests' {
    It 'Passes Test-ModuleManifest' {
        Test-ModuleManifest -Path $ModuleManifestPath | Should Not BeNullOrEmpty
        $? | Should Be $true
    }
}
```

You can use Plaster parameters that you define as well as a set of hard-coded variables.

- PLASTER_TemplatePath : The absolute path to the template directory.
- PLASTER_DestinationPath : The absolute path to the destination directory.
- PLASTER_DestinationName : The name of the destination directory.
- PLASTER_FileContent : The contents of a file be modified via the `<modify>` directive.
- PLASTER_DirSepChar : The directory separator char for the platform.
- PLASTER_HostName : The PowerShell host name e.g. $Host.Name
- PLASTER_Version : The version of the Plaster module invoking the template.
- PLASTER_Guid1 : Randomly generated GUID value
- PLASTER_Guid2 : Randomly generated GUID value
- PLASTER_Guid3 : Randomly generated GUID value
- PLASTER_Guid4 : Randomly generated GUID value
- PLASTER_Guid5 : Randomly generated GUID value
- PLASTER_Date : Date in short date string format e.g. 10/31/2016
- PLASTER_Time : Time in short time string format e.g. 5:11 PM
- PLASTER_Year : The four digit year

Armed with this information, we can turn the other static files into dynamic templates. For example, the README.md template file now looks like this:

```
# <%=$PLASTER_PARAM_ModuleName%>


*last updated <%=$PLASTER_Date%>*
```

As long as the manifest has the correct settings, Plaster will make the substitutions.

```
<templateFile source='changelog.txt' destination='changelog.txt'/>
<templateFile source='README.md' destination='README.md'/>
<templateFile source='license.txt' destination='license.txt'/>
<templateFile source='test\Module.T.ps1' destination='test\${PLASTER_PARAM_ModuleNam\
e}.Tests.ps1' />
```

We've included a version of the mySample folder with the template file changes in the code download.

Adding Messages

Plaster does a pretty good job at letting you know what is going on. But you can also write additional messages in the `<content/>` block.

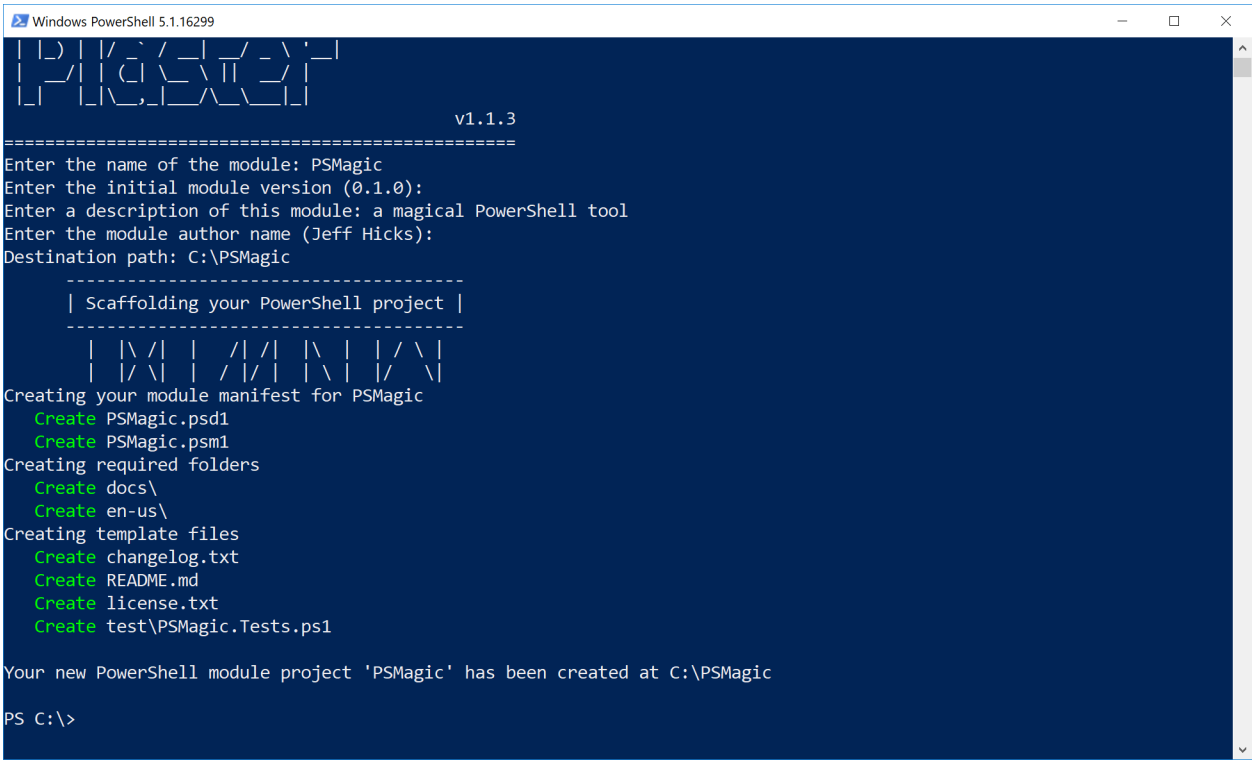
```
<content>
 <message>Scaffolding your PowerShell Project</message>
 <file destination='docs' source=''/>
 <file destination='en-us' source=''/>
 ...
</content
```

Content items are processed sequentially. Here's a manifest excerpt:

```
<content>
      <message>
      --------------------------------------
      | Scaffolding your PowerShell project |
      --------------------------------------
        |  |\ /|  |   /| /|  |\  |  | / \ |
        |  |/ \|  |  / |/ |  | \ |  |/   \|
      </message>
      <message>Creating your module manifest for ${PLASTER_PARAM_ModuleName}</messag\
e>
    <newModuleManifest
      destination='${PLASTER_PARAM_ModuleName}.psd1'
      moduleVersion = '$PLASTER_PARAM_Version'
      rootModule = '${PLASTER_PARAM_ModuleName}.psm1'
      encoding = 'UTF8-NoBOM'
      author = '$PLASTER_PARAM_ModuleAuthor'
      description = '$PLASTER_PARAM_Description'
      openInEditor = "true"
      />
      <file source='Module.psm1' destination='${PLASTER_PARAM_ModuleName}.psm1'/>
    <message>Creating required folders</message>
    <file destination='docs' source=''/>
    <file destination='en-us' source=''/>
    <message>Creating template files</message>
    <templateFile source='changelog.txt' destination='changelog.txt'/>
    <templateFile source='README.md' destination='README.md'/>
    <templateFile source='license.txt' destination='license.txt'/>
    <templateFile source='test\Module.T.ps1' destination='test\${PLASTER_PARAM_Modul\
eName}.Tests.ps1' />
     <message>

Your new PowerShell module project '$PLASTER_PARAM_ModuleName' has been created at $\
PLASTER_DestinationPath
```

```
    </message>
  </content>
```

As you can see, you can include Plaster variables and parameters in your message text. This is the output from invoking the template manifest:



Plaster template messages

Creating a Plaster Function Template

Creating a folder structure for a new PowerShell project is very helpful. But you can leverage the template file feature to create function code. This needs a different type of Plaster manifest - an *Item* manifest. The concepts are still the same. We'll end up with a plastermanifest.xml file in the root of a folder with supporting files.

```
PS C:\myFunction> New-PlasterManifest -TemplateName myFunction -TemplateType Item -D\
escription "Function scaffolding" -author "Art Deco"
```

This gives us the beginning of a manifest.

```
<?xml version="1.0" encoding="utf-8"?>
<plasterManifest
  schemaVersion="1.1"
  templateType="Item" xmlns="http://www.microsoft.com/schemas/PowerShell/Plaster/v1">
  <metadata>
    <name>myFunction</name>
    <id>4a3def64-dd0a-4b46-b959-1fb56a525c19</id>
    <version>1.0.0</version>
    <title>myFunction</title>
    <description>Function scaffolding</description>
    <author>Art Deco</author>
    <tags></tags>
  </metadata>
  <parameters></parameters>
  <content></content>
</plasterManifest>
```

We're going to need some information which we can get through a set of parameters.

```
<parameter name='Name' type='text' prompt='Enter the name of your function.'/>
<parameter name='Version' type='text' prompt='What is the function version?' default\
='0.1.0'/>
<parameter name='OutputType' type='text' prompt='What type of output is expected' de\
fault="[PSCustomObject]"/>
```

Even though we didn't mention it with the module-based manifests, you can create parameters that offer a choice of possible values. This will come in handy when scaffolding a function. For example, we might want to ask if the function needs to include code for SupportsShouldProcess.

```
<parameter name="ShouldProcess" type="choice" prompt="Do you need to support -WhatIf\
" default='1'>
  <choice label="&amp;Yes" help="Adds SupportsShouldProcess" value="Yes" />
  <choice label="&amp;No" help="Does not add SupportsShouldProcess" value="No" />
</parameter>
```

Within the parameter tag you'll define a set of `<choice>` tags. The `&` helps identify an accelerator character and goes in front of the desired character. The end result will be `_Yes` and `_No` where the user only needs to type `Y` or `N`. The Default value is based on a `0`-based array. In our example, the default is No.

Here is the finished manifest which you'll also find in the code downloads.

```
<?xml version="1.0" encoding="utf-8"?>
<plasterManifest
  schemaVersion="1.1"
  templateType="Item" xmlns="http://www.microsoft.com/schemas/PowerShell/Plaster/v1">
  <metadata>
    <name>myFunction</name>
    <id>4a3def64-dd0a-4b46-b959-1fb56a525c19</id>
    <version>1.0.0</version>
    <title>myFunction</title>
    <description>Function scaffolding</description>
    <author>Art Deco</author>
    <tags></tags>
  </metadata>
  <parameters>
        <parameter name='Name' type='text' prompt='Enter the name of your function.'\
/>
        <parameter name='Version' type='text' prompt='What is the function version?'\
 default='0.1.0'/>
        <parameter name='OutputType' type='text' prompt='What type of output is expe\
cted' default="[PSCustomObject]"/>
        <parameter name="ShouldProcess" type="choice" prompt="Do you need to support\
 -WhatIf ?" default='1'>
        <choice label="&amp;Yes" help="Adds SupportsShouldProcess" value="Yes" />
        <choice label="&amp;No" help="Does not add SupportsShouldProcess" value="No"\
 />
      </parameter>
      <parameter name="Help" type="choice" prompt="Do you need comment based help?" \
default='1'>
        <choice label="&amp;Yes" help="Add comment based help outline" value="Yes" />
        <choice label="&amp;No" help="Does not add comment based help" value="No" />
        </parameter>
      <parameter name="ComputerName" type="choice" prompt="Add a parameter for Compu\
tername?" default='0'>
        <choice label="&amp;Yes" help="Adds a default parameter for computername" va\
lue="Yes" />
        <choice label="&amp;No" help="Does not include computername parameter" value\
="No" />
        </parameter>
  </parameters>
  <content>
  <message>'/|= Scaffolding your PowerShell function $PLASTER_PARAM_Name =|\'</messa\
ge>
      <templateFile source='function-template.ps1' destination='${PLASTER_PARAM_Name\
```

```
}.ps1'/>
      <message>Your function, '$PLASTER_PARAM_Name', has been saved to '$PLASTER_Des\
tinationPath\$PLASTER_PARAM_Name.ps1'</message>
  </content>
</plasterManifest>
```

Before we can run this, we need to create the function template. We'll use the same concept that we used with files in the module manifest where Plaster will replace `<% plaster-variables %>` with text. For example, we might want to include a comment header in the function with the version information and creation date. Our function template file would include code like this:

```
<%
@"
# version: $PLASTER_PARAM_version
# created: $PLASTER_Date
"@
%>
```

You need to explicitly tell Plaster you are replacing everything inside the `<% %>` with text. In this case, a here string. Of course, we can do the same thing with the name of the new function.

```
<%
"Function $PLASTER_PARAM_Name {"
%>
```

But now it gets interesting. We can use some simple logic to dynamically add content. In our manifest we're prompting if the user wants to include comment based help. Their response is saved to a Plaster parameter value. We can test that parameter in the function template file and insert a string of text if the answer is Yes.

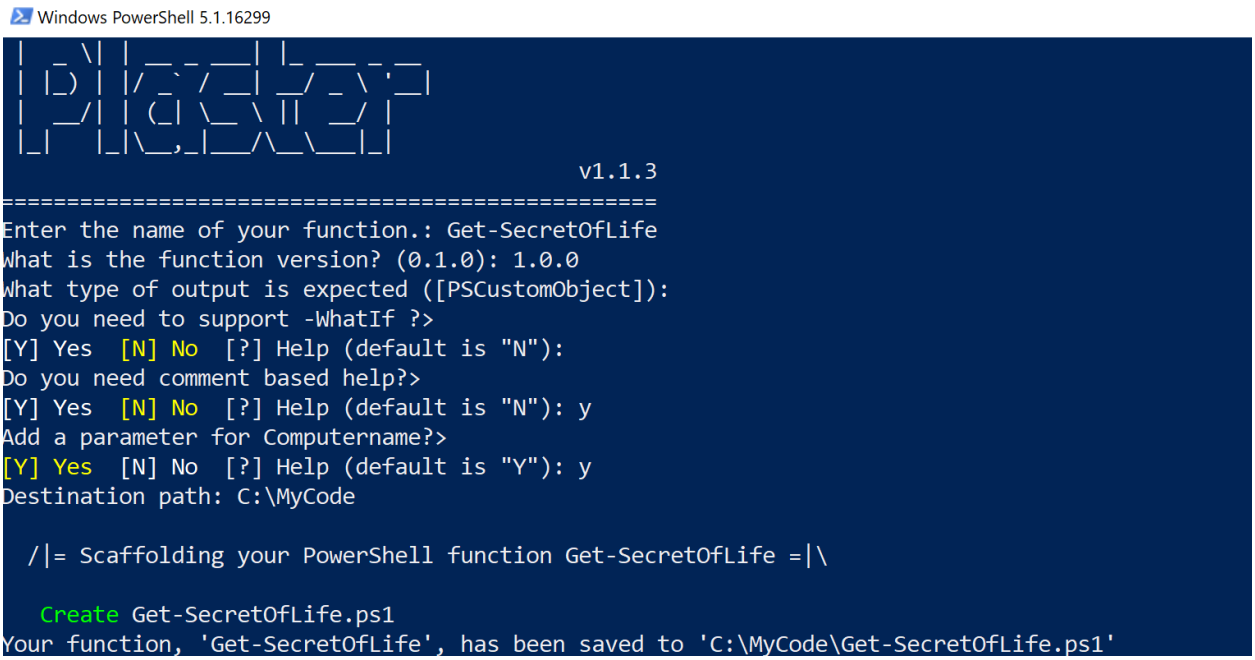
```
<%
    If ($PLASTER_PARAM_Help -eq 'Yes')
    {
        @"
  <#
    .SYNOPSIS
      Short description
    .DESCRIPTION
      Long description
    .PARAMETER XXX
      Describe the parameter
    .EXAMPLE
```

```
      Example of how to use this cmdlet
    .NOTES
      insert any notes
    .LINK
      insert links
  #>
"@
    }
%>
```

You have to be careful. The curly braces are part of the `If` statement and not part of the text you are inserting into the file. The inserted text is the here string with the comment-based help. We can repeat this process for other parts of the function, based on the user's answers to the parameter prompts.

```
<%
    if ($PLASTER_PARAM_ShouldProcess -eq 'Yes') {
        "[cmdletbinding(SupportsShouldProcess)]"
    }
    else {
        "[cmdletbinding()]"
    }
%>
<%
"[OutputType($PLASTER_PARAM_OutputType)]"
%>
<%
    if ($PLASTER_PARAM_computername -eq 'Yes') {
    @'
    Param(
        [Parameter(Position=0,ValueFromPipeline,ValueFromPipelineByPropertyName)]
        [ValidateNotNullorEmpty()]
        [string[]]$ComputerName = $env:COMPUTERNAME
    )
'@
    }
    else {
    @'
    Param()
'@
    }
%>
```

You can find the completed template in the code downloads. Let's invoke the template.



Plaster Function Template

Within seconds we have the beginnings of an advanced PowerShell function.

```
#requires -version 5.0

# version: 1.0.0
# created: 2/19/2020

Function Get-SecretOfLife {
  <#
    .SYNOPSIS
      Short description
    .DESCRIPTION
      Long description
    .PARAMETER XXX
      Describe the parameter
    .EXAMPLE
      Example of how to use this cmdlet
    .NOTES
      insert any notes
    .LINK
      insert links
  #>
```

```
[cmdletbinding()]
[OutputType([PSCustomObject])]

    Param(
        [Parameter(Position=0,ValueFromPipeline,ValueFromPipelineByPropertyName)]
        [ValidateNotNullorEmpty()]
        [string[]]$ComputerName = $env:COMPUTERNAME
    )

    Begin {
        Write-Verbose "[$((Get-Date).TimeofDay) BEGIN  ] Starting $($myinvocation.my\
command)"

    } #begin

    Process {
            Foreach ($computer in $Computername) {
            Write-Verbose "[$((Get-Date).TimeofDay) PROCESS] Processing $($computer.\
toUpper())"
              #<insert code here>
            }
    } #process

    End {
        Write-Verbose "[$((Get-Date).TimeofDay) END    ] Ending $($myinvocation.myco\
mmand)"
    } #end

} #close Get-SecretOfLife
```

Integrating Plaster into your PowerShell Experience

We've spent a great deal of time in this chapter on Plaster mechanics. Before we go though, we want to show you why this is worth the effort, especially if you are locked into VS Code as your primary editor. If you recall at the beginning of the chapter we showed how to use `Get-PlasterTemplate` to identify available templates. But this command has an additional parameter, `-IncludeInstalledModules`. When you include this parameter, Plaster will check installed module manifests for particular bit of code to indicate that the module contains templates.

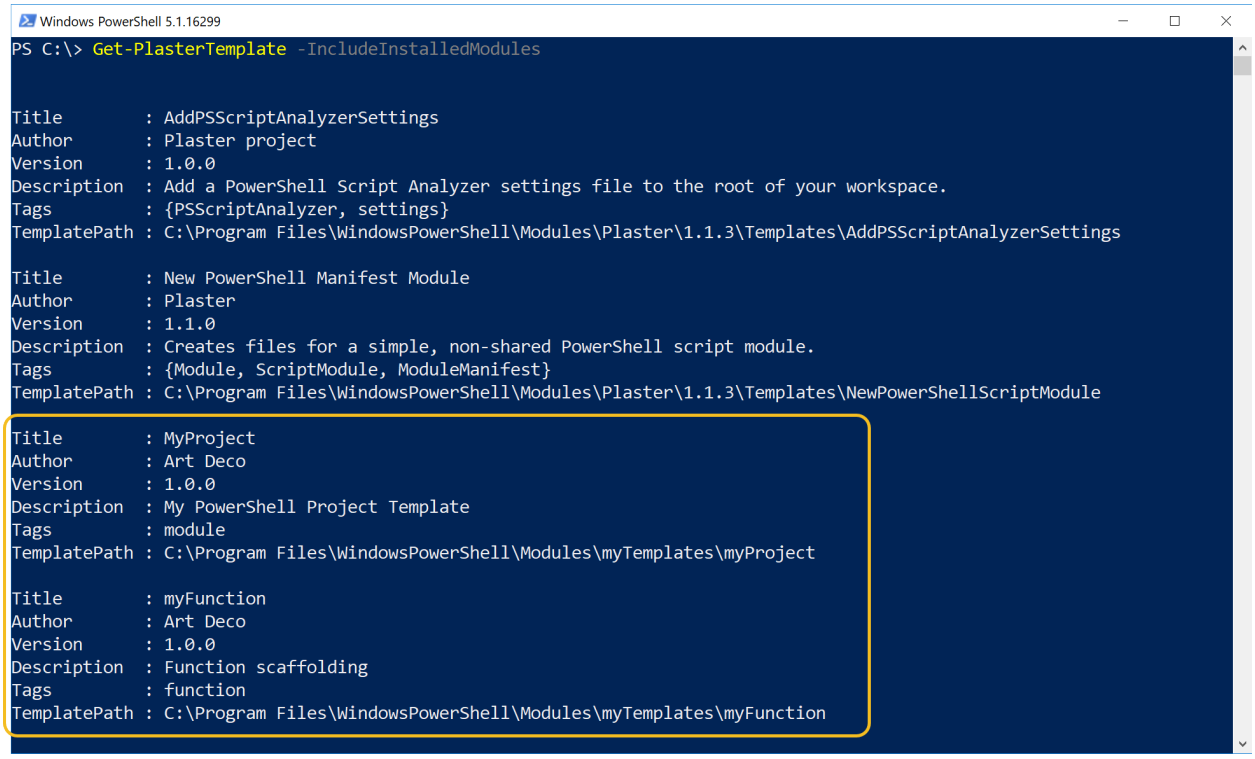
Under the PSData section of the module manifest, we're going to add a setting for Extensions and specify an array of Plaster template names.

```
    Extensions = @(
        @{
            Module = "Plaster"
            Details = @{
                TemplatePaths = @("myProject","myFunction")
            }
        }
    )
```

The paths are relative to the module manifest. We took our function and project template folders and copied them to a new module called MyTemplates.

```
PS C:\> dir 'C:\Program Files\WindowsPowerShell\Modules\myTemplates\'


    Directory: C:\Program Files\WindowsPowerShell\Modules\myTemplates


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        2/19/2018  11:03 AM                myFunction
d-----        2/19/2018  11:06 AM                myProject
-a----        2/19/2018  11:25 AM           8186 myTemplates.psd1
-a----        2/19/2018  11:23 AM             95 myTemplates.psm1
```

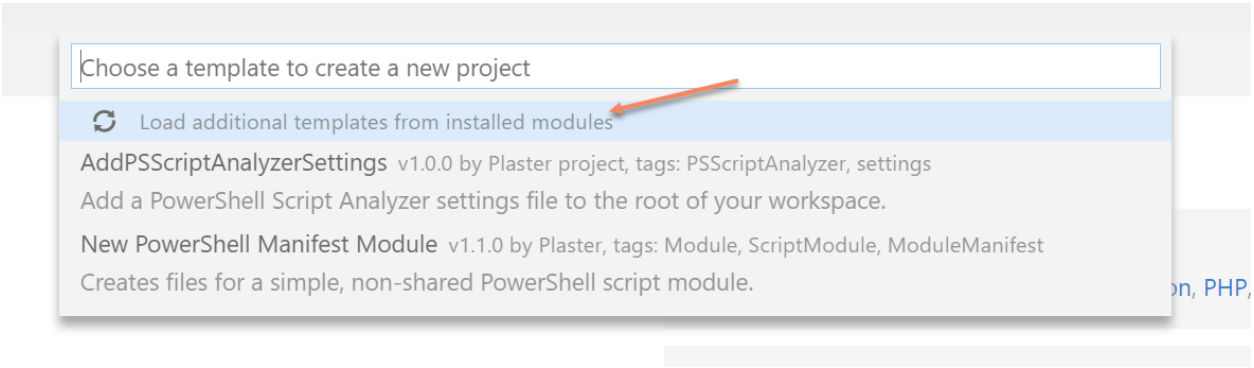
The psm1 file is empty except for a comment indicating why it is empty. The manifest file includes the Extensions setting and also specifies Plaster as a required module. Now we get these templates.



Get custom templates

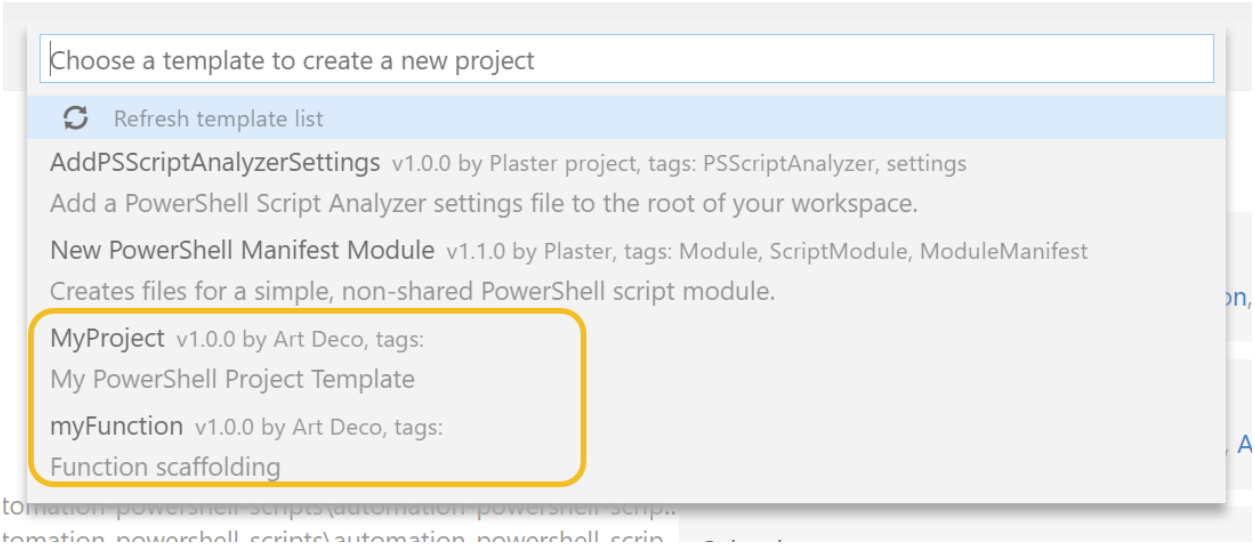
We could use these templates like we did at the beginning of this chapter. But because we're using VSCode we can also invoke them via the command palette.

In VSCode press `Ctrl+Shift+P` to bring up the command palette. Then start typing 'PowerShell: Create New Project from Plaster Template'. After a moment or two, click on the option to load additional templates.



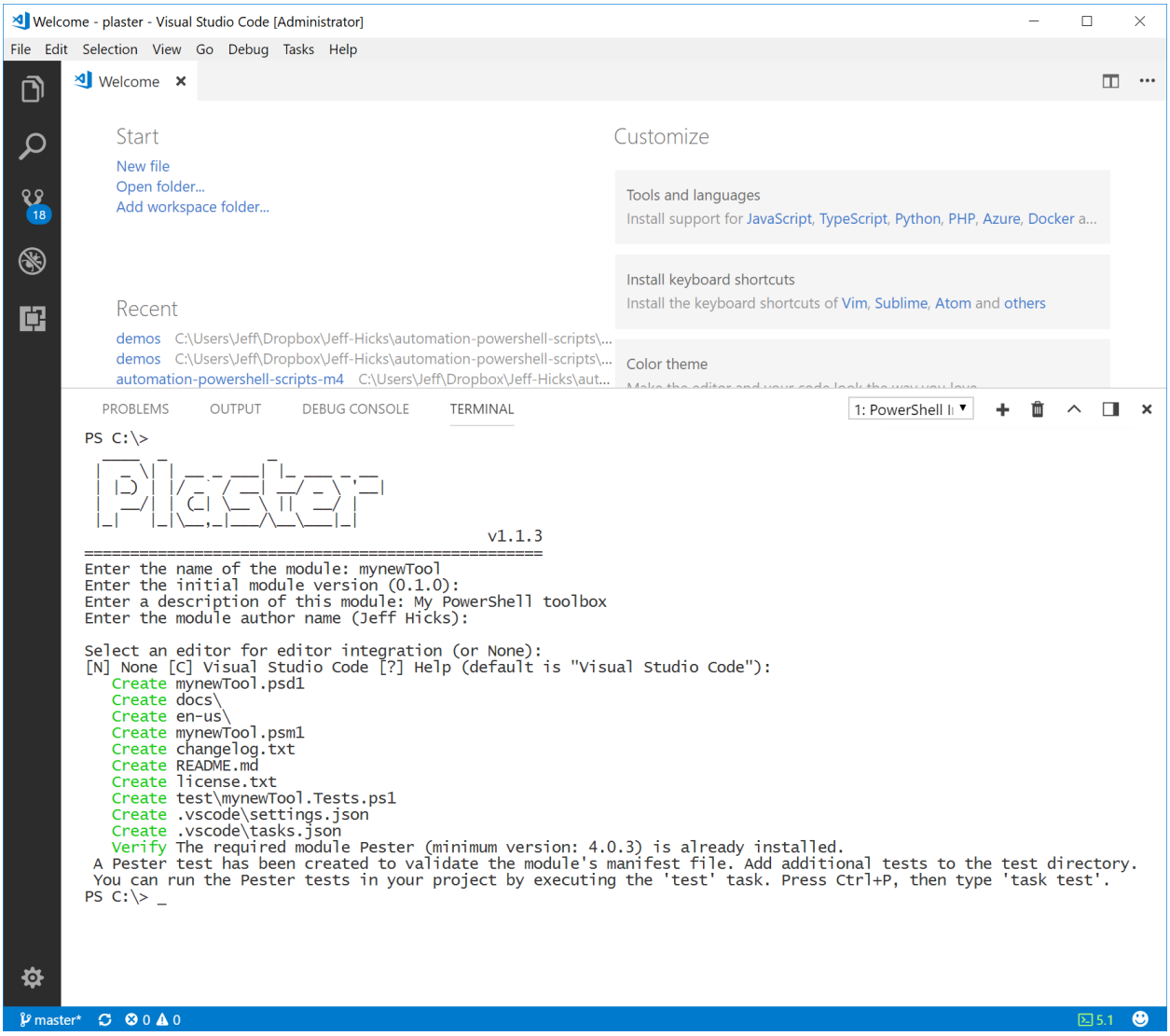
Load additional Plaster templates

Within seconds you should get the list of your custom templates.



Available custom templates

When you select one, VSCode will prompt you for parameter values.



Invoking a Plaster template in VSCode

We created a new module project which is then immediately opened up in VSCode. We can now invoke the MyFunction manifest and begin adding code to the module. Within minutes we can create the outlines of a complete PowerShell module.

Tip

To skip the extra step of forcing VS Code to always search installed modules for Plaster templates, you can take advantage of $PSDefaultParameterValues. In either the PowerShell profile script for all hosts, or the VS Code specific profile profile add this command:

```
$PSDefaultParameterValues."Get-PlasterTemplate:IncludeInstalledModules"=$True
```

Now when you invoke the VS Code command to create a new project from Plaster it will automatically display all templates from modules.

Creating Plaster Tooling

Finally, we take advantage of one other Plaster feature and create everything from a PowerShell prompt. Even though our Plaster manifests prompt for values, Plaster will dynamically generate parameters for `Invoke-Plaster`. This means we can use something like this to rapidly generate outlines for numerous files.

```
"Get-MyThing","Set-MyThing","Remove-MyThing","Invoke-Something" |
foreach-object -begin {

$splat = @{
TemplatePath = 'C:\Program Files\WindowsPowerShell\Modules\myTemplates\myFunction\'
DestinationPath = "c:\MyNewTool"
version = "0.1.0"
Outputtype = "[PSCustomObject]"
shouldprocess = "Yes"
help = "No"
computername = "yes"
NoLogo = $True
}

} -process {
 #add the name
 $splat.name = $_
 if ($_ -match 'Get') {
    $splat.ShouldProcess = "No"
 }
 Invoke-Plaster @splat
}
```

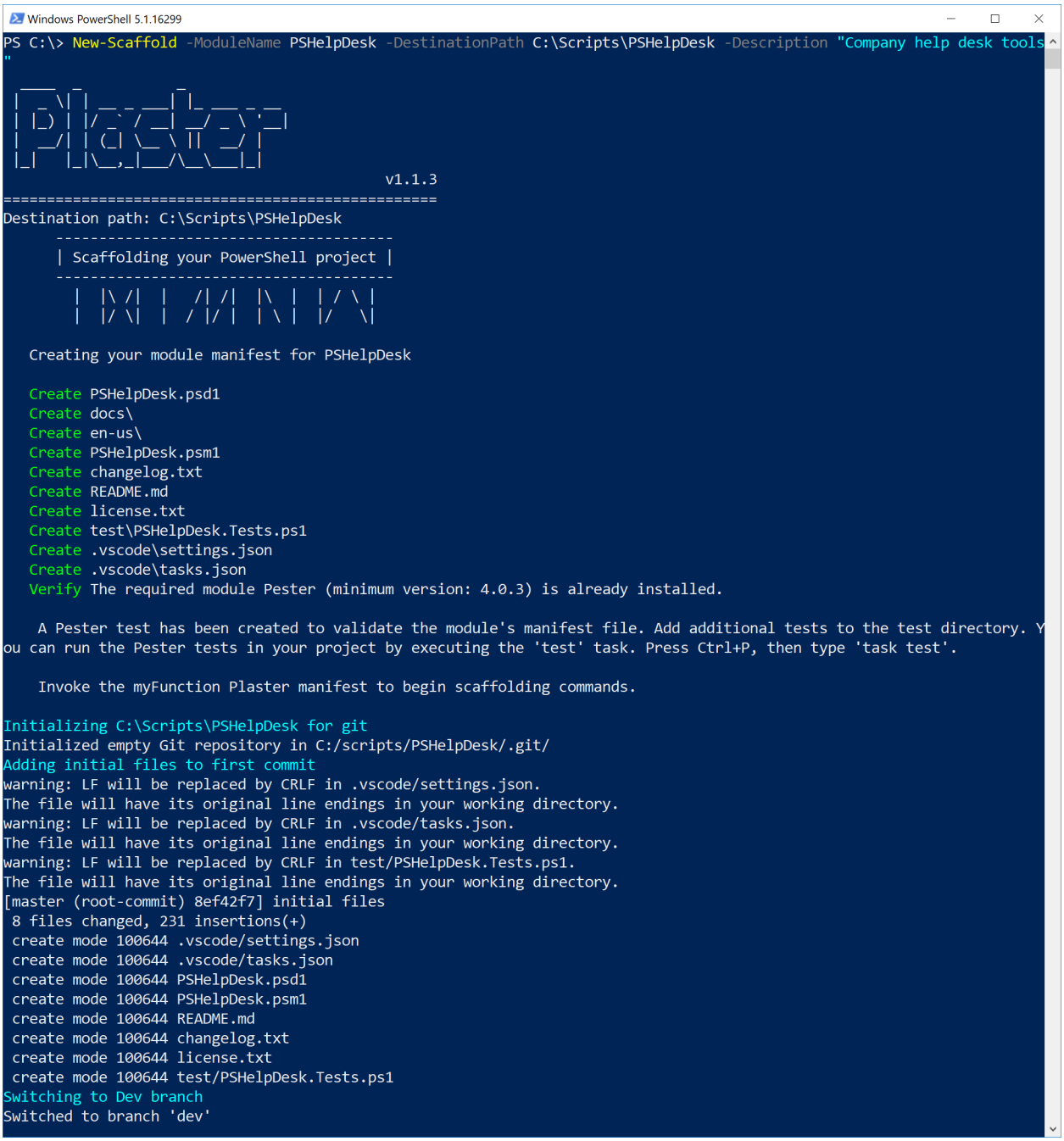
This means that you can create your own tooling around your Plaster templates that don't rely on VS Code.

New-Scaffold.ps1

```
#requires -version 5.0
#requires -module Plaster

#this function assumes you have git installed and configured

Function New-Scaffold {
    [cmdletbinding(SupportsShouldProcess)]
    Param(
        [Parameter(
            Mandatory,
            HelpMessage = "Enter the name of your new module."
            )]
        [ValidateNotNullorEmpty()]
        [string]$ModuleName,
        [Parameter(
            Mandatory,
            HelpMessage = "The folder name for your new module. The top level name s\
hould match the module name"
            )]
        [ValidateNotNullorEmpty()]
        [string]$DestinationPath,
        [Parameter(
            Mandatory,
            HelpMessage = "Enter a brief description about your project"
         )]
        [ValidateNotNullorEmpty()]
        [string]$Description,
        [Parameter(HelpMessage ="The module version")]
        [string]$Version = "0.1.0",
        [Parameter(HelpMessage ="The module author which should be your git user nam\
e")]
        [string]$ModuleAuthor = $(git config --get user.name),
        [ValidateSet("none", "VSCode")]
        [Parameter(HelpMessage = "Do you want to include VSCode settings?")]
        [string]$Editor = "VSCode",
        [Parameter(HelpMessage = "The minimum required version of PowerShell for you\
r module")]
        [string]$PSVersion = "5.0",
        [Parameter(HelpMessage = "The path to the Plaster template")]
        [ValidateNotNullorEmpty()]
        [ValidateScript({ Test-Path $_ })]
        [string]$TemplatePath = "C:\Program Files\WindowsPowerShell\Modules\myTempla\
```

```
tes\myProject\"
    )

    if (-Not (Test-PlasterManifest -Path $TemplatePath\plastermanifest.xml)) {
        write-Warning "Failed to find a valid plastermanifest.xml file in $TemplateP\
ath"
        #bail out
        return
    }
    if (-Not $PSBoundParameters.ContainsKey("templatePath")) {
        $PSBoundParameters["TemplatePath"] = $TemplatePath
    }
    if (-not $PSBoundParameters.ContainsKey("version")) {
        $PSBoundParameters["version"] = $version
    }
    if (-not $PSBoundParameters.ContainsKey("ModuleAuthor")) {
        $PSBoundParameters["ModuleAuthor"] = $ModuleAuthor
    }
    if (-not $PSBoundParameters.ContainsKey("Editor")) {
        $PSBoundParameters["editor"] = $editor
    }
    if (-not $PSBoundParameters.ContainsKey("PSVersion")) {
        $PSBoundParameters["PSVersion"] = $PSVersion
    }

    $PSBoundParameters | Out-String | Write-Verbose
    Invoke-Plaster @PSBoundParameters

    if ($PSCmdlet.ShouldProcess($DestinationPath)) {
        Write-Host "Initializing $DestinationPath for git" -ForegroundColor cyan
        Set-Location $DestinationPath
        git init
        Write-Host "Adding initial files to first commit" -ForegroundColor cyan
        git add .
        git commit -m "initial files"
        Write-Host "Switching to Dev branch" -ForegroundColor cyan
        git branch dev
        git checkout dev
    }
    Write-Host "Scaffolding complete" -ForegroundColor green
}
```

This script assumes you have `git` installed. After the Plaster creates the module scaffolding, the function then uses git to initialize the folder as a git repository, make an initial commit of files and then checkout a dev branch. Now, one command sets up everything! The file in the code download won't have the wonky line wrapping you might see here.



Scaffolding with a Plaster-based function

There's no doubt that there is a learning curve for Plaster. But once you take the time to put together

a template, you'll find yourself using it all the time.

Adding Auto Completion

PowerShell has always been designed and intended to be easy to use. Nobody wants to spend any more typing than they actually have to. The PowerShell team thought about this and you probably don't even think about it. You can type a command like `Get-Service`, press the space bar and then start pressing <kbd>Tab</kbd>. PowerShell will cycle through all the possible values for the positional `Name` parameter. Once you get used to that kind of auto completion, you start looking for it everywhere. And for a professional toolmaker, this might be something you want to add to your own work.

ValidateSet

Technically, this is a little bit different than what we have in mind for this chapter, but it can have the same net result. Look at this simple demo function.

Get-Foo

```
Function Get-Foo {
    [cmdletbinding()]
    Param(
        [Parameter(Position = 0)]
        [ValidateSet("This","That","Other","Squirrel")]
        [string]$Item = "This"
    )

   Write-Host "Working with $item" -ForegroundColor green
}
```

We've added parameter validation specifying that the value for the `-Item` parameter must be one of the values. When someone runs the function they press <kbd>Tab</kbd> to cycle through the options. But don't forget that the user can't specify any other value.

Argument Completer Attribute

The other option you have is to add an `[ArgumentCompleter()]` attribute to a parameter in your function. Here's an example.

Get-ProcessDetail

```
Function Get-ProcessDetail {
    [cmdletbinding()]
    Param(
        [Parameter(Position = 0, Mandatory)]
        [ArgumentCompleter({(Get-Process).name})]
        [string]$Name
    )

    Get-Process -Name $Name |
    Select-Object -property ID, Name, StartTime, WorkingSet,
    @{Name = "Path" ; Expression = {$_.MainModule.FileName}},
    @{Name = "RunTime"; Expression = {(Get-Date) - $_.starttime}}
}
```

The attribute needs a scriptblock that will generate the argument completer results. In our sample it is running `Get-Process` and returning the name property. You can have as much code as you need in the script block but we suggest it be something that can execute quickly. You don't want to be waiting around the argument completion to finish.

The user isn't forced to use the completer or its results. They can type whatever value they want.

Advanced Argument Completers

For all your other auto completion needs, you can create a separate argument completer. This thing exists outside of your command. You'll want to spend a few minutes looking at the help and examples for `Register-ArgumentCompleter`. In fact, pay close attention to the examples because you're going to use them as boilerplate templates.

The ultimate goal is to come up with a scriptblock that will generate the values you need. Again, this shouldn't take very long to run. What's nice with this approach is that you can add wildcard support so that the user can start typing part of the value and your argument completer can fill in the rest. Let's build one for an existing cmdlet that doesn't have auto complete but we wish it did.

When you use `Get-Eventlog`, PowerShell will cycle through and auto complete all of the possible log names. However, `Get-WinEvent` does not. You can list them, but the `-LogName` parameter doesn't auto complete. We can fix that.

First, we need the scriptblock.

Get-WinEvent LogName Scriptblock

```
$sb = {
 param(
  $commandName,
  $parameterName,
  $wordToComplete,
  $commandAst,
  $fakeBoundParameter
  )

 (Get-WinEvent -listlog "$wordtoComplete*").logname |
  ForEach-Object {
  [System.Management.Automation.CompletionResult]::new($_,$_,'ParameterValue',$_)
 }
}
```

We're using the same parameter names from the help example. What you need to come up with is the code that will generate the list of values. In our example that's what `(Get-Winevent -listlog "$wordtoComplete*").logname` is doing. But here's the fun part. You can include the `$wordtoComplete` variable as placeholder for what the user is typing. You can include wildcards. We're adding one at the end. If the user starts typing "Micro" and presses <kbd>Tab</kbd the auto completer will fill in the return the rest. You can decide how much wildcard support you need.

The values from the `Get-WinEvent` expression are used to create a CompletionResult object. The `new()` method takes needs 4 parameters.

```
new(completion text,listitem text,result type,Tooltip)
```

- *Completion Text* This is the value that will be used for the parameter.
- *ListItem Text* This will be used in lists that you can see when using the PowerShell ISE or the PSReadline module.
- *Result Type* Will be `ParameterValue`.
- *ToolTip* Gives you an opportunity to display additional information about the value. You'll need something like the PowerShell ISE to really see this.

In our example, we are using log name for all values.

The last step is to register the completer.

```
$params = @{
 CommandName = "Get-WinEvent"
 ParameterName = "Logname"
 ScriptBlock   = $sb
}
Register-ArgumentCompleter @params
```

With this in place, we now get tab completion for the LogName parameter. It may take a moment to populate the results. This change only lasts for as long as your PowerShell session is running. If this is something you want all the time, you would add this code to your PowerShell profile.

Sadly, there isn't a `Get-ArgumentCompleter` command. But you can see it with a bit of .NET code.

```
PS C:\> [System.Management.Automation.CompletionCompleters]::CompleteCommand("Get-Wi\
nEvent")

CompletionText ListItemText ResultType ToolTip
-------------- ------------ ---------- -------
Get-WinEvent   Get-WinEvent    Command â€¦
```

If you need to make a change, simply re-register a new completer.

And you don't have to use the `$WordtoComplete` variable. Here's a variation on our function to get a folder size.

Measure Folder

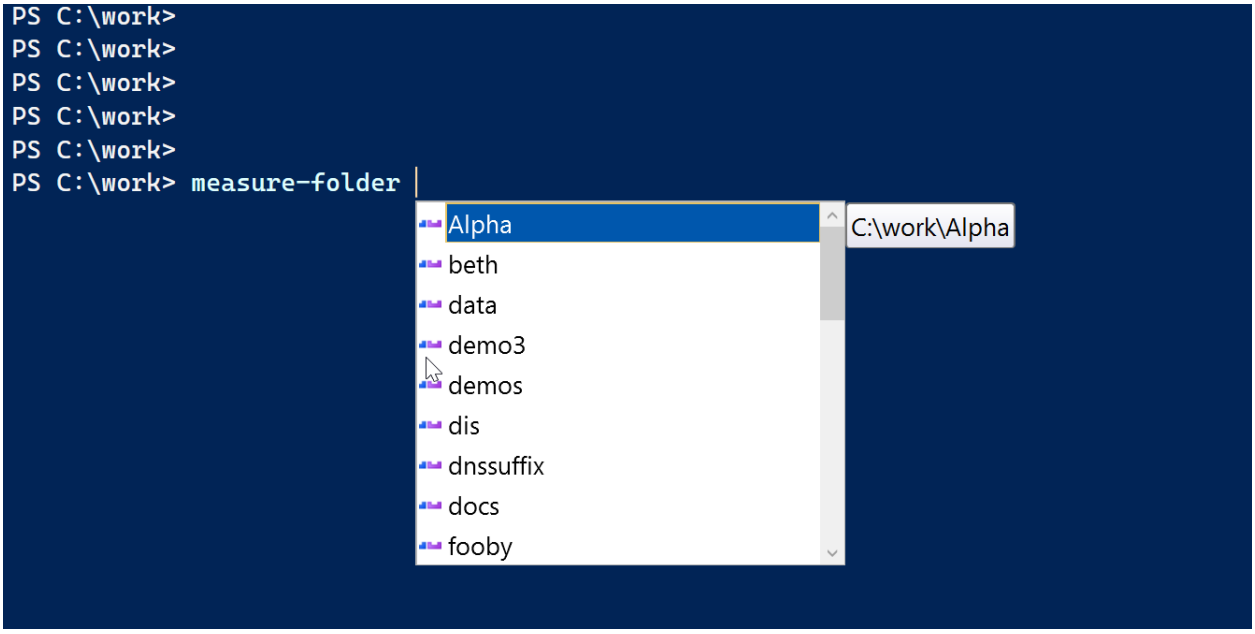
```
Function Measure-Folder {
[cmdletbinding()]
Param(
[Parameter(Position = 0,HelpMessage = "Specify the folder path to measure")]
[string]$Name = "."
)

Get-ChildItem -path $Name -Recurse -File |
Measure-Object -Property length -sum -Average |
Select-Object @{Name="Path";Expression={Convert-Path $Name}},
Count,Sum,Average

}
```

And an argument completer for the `-Name` parameter.

Measure Folder Auto Completer

```
$sb = {
 param(
  $commandName,
  $parameterName,
  $wordToComplete,
  $commandAst,
  $fakeBoundParameter
)
 Get-Childitem -path . -Directory |
 ForEach-Object {
  [System.Management.Automation.CompletionResult]::new($_.fullname,$_.name,
  'ParameterValue',$_.fullname)
 }
 }

$params = @{
 CommandName = "Measure-Folder"
 ParameterName = "Nanme"
 ScriptBlock   = $sb
}
Register-ArgumentCompleter @params
```

The scriptblock is defaulting to getting the directories in the current location. Here's how it all looks in the PowerShell ISE.



AutoCompleter

This makes it clear that the Name property is the list item and the full name is the tool tip.

If you want to include argument completers, the easiest thing is to put the code in the module's .psm1 file. The completers will be registered automatically when the module is imported.

Your Turn

Let's see what you can do with what we've shown you in this chapter. If you have a function of your own that you think could use auto completion, go ahead and add it. Otherwise, we'll provide an exercise.

Start Here

The `Get-Command` command has a `-Verb` parameter. But you have to type the name of a standard verb. Create an auto completion code snippet so that this parameter can auto-complete with a standard verb.

Our Take

Here's what we came up with.

Verb Auto Completer

```
$sb = {
    param (
      $commandName,
      $parameterName,
      $wordToComplete,
      $commandAst,
      $fakeBoundParameter
      )

    Get-Verb -Verb "$wordToComplete*" |
    ForEach-Object {
        [System.Management.Automation.CompletionResult]::new($_.Verb,$_.Verb,
        'ParameterValue',("Group: $($_.Group)"))
    }
}

$params = @{
 CommandName = "Get-Command"
 ParameterName = "Verb"
 ScriptBlock   = $sb
}

Register-ArgumentCompleter @params
```

We're populating the auto completion list with results from `Get-Verb`. We even added a tool tip that you can see when using the PSReadline module.

```
>> }
PS C:\>
PS C:\> Register-ArgumentCompleter @params
PS C:\> get-command -verb Add
Add         Lock        Resize      Watch       Export      Sync        Invoke      Wait        Receive
Clear       Move        Search      Backup      Group       Unpublish   Register    Debug       Send
Close       New         Select      Checkpoint  Import      Update      Request     Measure     Write
Copy        Open        Set         Compare     Initialize  Approve     Restart     Ping        Block
Enter       Optimize    Show        Compress    Limit       Assert      Resume      Repair      Grant
Exit        Pop         Skip        Convert     Merge       Complete    Start       Resolve     Protect
Find        Push        Split       ConvertFrom Mount       Confirm     Stop        Test        Revoke
Format      Redo        Step        ConvertTo   Out         Deny        Submit      Trace       Unblock
Get         Remove      Switch      Dismount    Publish     Disable     Suspend     Connect     Unprotect
Hide        Rename      Undo        Edit        Restore     Enable      Uninstall   Disconnect  Use
Join        Reset       Unlock      Expand      Save        Install     Unregister  Read

Group: Common
```

Verb Auto completion

Let's Review

1. What parameter attribute would you add to your function to get auto complete functionality?
2. What command would you use to add auto complete functionality to any command?
3. What is the benefit of using an argument completion attribute in your function instead of `[ValidateSet()]`?
4. If adding auto complete functionality to commands in your module, where is the best place for the code?

Review Answers

1. `[ArgumentCompleter()]`
2. Register-ArgumentCompleter
3. When using `[ValidateSet()]` the user can only use a value in the defined set. With an argument completer they can still enter whatever value they want. Although you may still want to validate the value.
4. In the module's .psm1 file.

Adding Custom Formatting

At this point in the book you should recognize the value of "objects in the pipeline". For the most part, we're kinda letting PowerShell do its own thing. We give it a bunch of objects and it decides how to display them. But maybe you want a little more control or want to have a nicer user experience for your tool. You might need to tell PowerShell how to format your objects.

Here's an example function.

New-SysInfo

```
Function New-SysInfo {
    [cmdletbinding()]
    Param([string[]]$Computername = $env:Computername)

    $cim = @{
        Classname    = ""
        Computername = ""
        ErrorAction  = "Stop"
    }
    foreach ($computer in $computername) {
        $cim.Computername = $Computer
        Try {
            $cim.classname = "Win32_OperatingSystem"
            $os = Get-CimInstance @Cim
            $cim.classname = "Win32_Process"
            $ps = Get-CimInstance @cim

            [PSCustomobject]@{
                Computername = $os.CSName
                Processes    = $PS.Count
                OS           = $os.Caption
                Build        = $os.BuildNumber
                BootTime     = $os.LastBootUpTime
            }
        }
        Catch {
            Write-Warning "Failed to get information from $($Computer.ToUpper())"
            Write-Warning $($_.exception.message)
        }
```

```
    } #foreach
} #New-SysInfo
```

When you run the command, you'll most likely get output as a list.

```
PS C:\> new-sysinfo


Computername : BOVINE320
Processes    : 316
OS           : Microsoft Windows 10 Pro
Build        : 19041
BootTime     : 6/11/2020 10:11:20 PM
```

But really, this would look fine as a table.

```
PS C:\> new-sysinfo | format-table

Computername Processes OS                       Build BootTime
------------ --------- --                       ----- --------
BOVINE320          316 Microsoft Windows 10 Pro 19041 6/11/2020 10:11:20 PM
```

But you don't want to force the user to remember to do this. You want the default display to always be a table.

Or perhaps your command is writing a very rich object to the pipeline and by default you want to see a specific set of properties displayed in a certain way. This too will require some custom formatting.

Format.ps1xml

PowerShell maintains its formatting rules in a set of XML files. These files will often follow a naming convention of `<typename>.format.ps1xml`. In Windows PowerShell you can find these files in the $PSHome folder. Here's a snippet that shows the formatting for service objects.

```
<?xml version="1.0" encoding="utf-8" ?>
<Configuration>
    <ViewDefinitions>
        <View>
            <Name>service</Name>
            <ViewSelectedBy>
                <TypeName>System.ServiceProcess.ServiceController</TypeName>
            </ViewSelectedBy>
             <TableControl>
                <TableHeaders>
                    <TableColumnHeader>
                        <Width>8</Width>
                    </TableColumnHeader>
                    <TableColumnHeader>
                        <Width>18</Width>
                    </TableColumnHeader>
                    <TableColumnHeader>
                        <Width>38</Width>
                    </TableColumnHeader>
                </TableHeaders>
                <TableRowEntries>
                    <TableRowEntry>
                        <TableColumnItems>
                            <TableColumnItem>
                                <PropertyName>Status</PropertyName>
                            </TableColumnItem>
                            <TableColumnItem>
                                <PropertyName>Name</PropertyName>
                            </TableColumnItem>
                            <TableColumnItem>
                                <PropertyName>DisplayName</PropertyName>
                            </TableColumnItem>
                        </TableColumnItems>
                    </TableRowEntry>
                </TableRowEntries>
            </TableControl>
        </View>
    </ViewDefinitions>
</Configuration>
```

This is why when you run `Get-Service` you see what you see. To create a custom format for your object, you need to create your own format xml file.

In PowerShell 7, these format xml files have been moved to inside the code for performance reasons. You can still create and use custom xml files but you'll need to use the ones in Windows PowerShell as examples.

The file contains a collection of View definitions. Each view has a name and is associated with a given typename.

```
<View>
   <Name>service</Name>
       <ViewSelectedBy>
           <TypeName>System.ServiceProcess.ServiceController</TypeName>
       </ViewSelectedBy>
```

Each view can then dictates what type of display to use such as table, list or wide. Table and List are most widely used.

Define a TypeName

Before you can use a format ps1xml file, you need to make sure your object has a unique typename. We've shown you a few ways in the book. If we want to add a default table view, we need to modify our function to include a custom typename. You can't define custom settings for a generic PSObject.

```
[PSCustomobject]@{
    PSTypename   = 'SysInfo'
    Computername = $os.CSName
    Processes    = $PS.Count
    OS           = $os.Caption
    Build        = $os.BuildNumber
    BootTime     = $os.LastBootUpTime
}
```

We can verify this with `Get-Member`.

```
PS C:\> new-sysinfo thinkp1 | get-member


   TypeName: SysInfo

Name         MemberType   Definition
----         ----------   ----------
Equals       Method       bool Equals(System.Object obj)
GetHashCode  Method       int GetHashCode()
GetType      Method       type GetType()
ToString     Method       string ToString()
BootTime     NoteProperty datetime BootTime=6/23/2020 8:46:39 AM
Build        NoteProperty string Build=19041
Computername NoteProperty string Computername=THINKP1
OS           NoteProperty string OS=Microsoft Windows 10 Pro
Processes    NoteProperty int Processes=240
```

Defining a View Definition

The table format for services is pretty close to what we want. We can copy and paste it into our editor, along with the opening and closing tags. We can then add column headers and items to the view, adjusting the column width as necessary. We also need to change the typename.

```
<?xml version="1.0" encoding="utf-8" ?>
<Configuration>
    <ViewDefinitions>
        <View>
            <Name>default</Name>
            <ViewSelectedBy>
                <TypeName>SysInfo</TypeName>
            </ViewSelectedBy>
             <TableControl>
                <TableHeaders>
                    <TableColumnHeader>
                        <Width>15</Width>
                    </TableColumnHeader>
                    <TableColumnHeader>
                        <Width>10</Width>
                        <Alignment>Right</Alignment>
                    </TableColumnHeader>
                    <TableColumnHeader>
```

```
                    <Width>25</Width>
                </TableColumnHeader>
                <TableColumnHeader>
                    <Width>6</Width>
                </TableColumnHeader>
                <TableColumnHeader>
                    <Label>LastBoot</Label>
                    <Width>22</Width>
                </TableColumnHeader>
            </TableHeaders>
            <TableRowEntries>
                <TableRowEntry>
                    <TableColumnItems>
                        <TableColumnItem>
                            <PropertyName>Computername</PropertyName>
                        </TableColumnItem>
                        <TableColumnItem>
                            <PropertyName>Processes</PropertyName>
                        </TableColumnItem>
                        <TableColumnItem>
                            <PropertyName>OS</PropertyName>
                        </TableColumnItem>
                         <TableColumnItem>
                            <PropertyName>Build</PropertyName>
                        </TableColumnItem>
                         <TableColumnItem>
                            <PropertyName>BootTime</PropertyName>
                        </TableColumnItem>
                    </TableColumnItems>
                </TableRowEntry>
            </TableRowEntries>
        </TableControl>
    </View>
  </ViewDefinitions>
</Configuration>
```

One change we made was to add an `<Alignment/>` tag to the seconds column which is the number of processes. We want it the values to be right-justified. We also added a `<Label/>` for the last entry. Instead of using the property name, we want to use a custom entry. This looks more complicated than it really is and in a little bit we'll show you and even easier way to create this XML.

Update-FormatData

Once the file is created, we need to load it into PowerShell. This is done with `Update-FormatData` and the path to your xml file.

```
Update-FormatData .\sysinfo.format.ps1xml
```

When you read the cmdlet help you'll see there are options for prepending or appending. This means do you want your definition to take precedence over anything Microsoft might have defined or come after. In this case it doesn't matter since we're using a unique and custom typename.

You only have to run this command once to load the format definition into your session. Typically, for a standalone function you can add the `Update-FormatData` command at the end of your script file. But now when we run the command we get a formatted table by default.

```
PS C:\> new-sysinfo thinkp1,bovine320,srv1

Computername     Processes OS                         Build  LastBoot
------------     --------- --                         -----  --------
THINKP1                240 Microsoft Windows 10 Pro  19041  6/23/2020 8:46:39 AM
BOVINE320              306 Microsoft Windows 10 Pro  19041  6/11/2020 10:11:20 PM
SRV1                    32 Microsoft Windows Serv... 14393  6/22/2020 2:54:08 PM
```

New-PSFormatXML

It can be tedious copying and pasting XML so Jeff wrote a function called `New-PSFormatXML` which is part of the PSScriptTools module. The process is to pipe a single instance of the object to the command, specify what type of view you want and the properties. The command creates the XML which you can fine tune if you want.

Since we already have a file, we're going to add a new view called `computer` that will be a custom table.

```
$new = @{
    Path       = '.\sysinfo.format.ps1xml'
    Properties = 'Processes', 'OS', 'BootTime'
    GroupBy    = 'Computername'
    ViewName   = 'computer'
    FormatType = 'table'
    Append     = $True
}
New-SysInfo | New-PSFormatXML @new
```

The command makes a best guess as to column widths if you want to use them but defaults to an AutoSize setting.

A nice feature of formatting files is that you can customize the heck out them. In our file, we added a new column called `Uptime` that gets its value from a `<ScriptBlock/>` entry. We also changed a few other property names to display differently than the property. The underlying object doesn't change. Only the way it is displayed.

Computer View

```
    <View>
      <!--Created 06/23/2020 09:16:53 by BOVINE320\Jeff-->
      <Name>computer</Name>
      <ViewSelectedBy>
        <TypeName>SysInfo</TypeName>
      </ViewSelectedBy>
      <GroupBy>
        <!--
            You can also use a scriptblock to define a custom property name.
            You must have a Label tag.
            <ScriptBlock>$_.machinename.toUpper()</ScriptBlock>
            <Label>Computername</Label>

            Use <Label> to set the displayed value.
          -->
        <PropertyName>Computername</PropertyName>
        <Label>Computername</Label>
      </GroupBy>
      <TableControl>
        <!--Delete the AutoSize node if you want to use the defined widths.-->
        <AutoSize />
        <TableHeaders>
          <TableColumnHeader>
            <Label>Processes</Label>
```

```
            <Width>12</Width>
            <Alignment>left</Alignment>
          </TableColumnHeader>
          <TableColumnHeader>
            <Label>OperatingSystem</Label>
            <Width>27</Width>
            <Alignment>left</Alignment>
          </TableColumnHeader>
          <TableColumnHeader>
            <Label>LastBootUpTime</Label>
            <Width>24</Width>
            <Alignment>right</Alignment>
          </TableColumnHeader>
          <TableColumnHeader>
            <Label>Uptime</Label>
            <Width>24</Width>
            <Alignment>right</Alignment>
          </TableColumnHeader>
        </TableHeaders>
        <TableRowEntries>
          <TableRowEntry>
            <TableColumnItems>
              <!--
            By default the entries use property names, but you can replace them with\
scriptblocks.
            <ScriptBlock>$_.foo /1mb -as [int]</ScriptBlock>
                -->
              <TableColumnItem>
                <PropertyName>Processes</PropertyName>
              </TableColumnItem>
              <TableColumnItem>
                <PropertyName>OS</PropertyName>
              </TableColumnItem>
              <TableColumnItem>
                <PropertyName>BootTime</PropertyName>
              </TableColumnItem>
               <TableColumnItem>
                <ScriptBlock>
                $ts = (Get-Date) - $_.BootTime
                $ts.ToString("dd\.hh\:mm\:ss")
                </ScriptBlock>
              </TableColumnItem>
            </TableColumnItems>
```

```
        </TableRowEntry>
      </TableRowEntries>
    </TableControl>
  </View>
```

As you are developing and testing your format file, don't forget to re-run `Update-FormatData`.

With the updated view loaded, we can now get formatted results that are more useful to us.

```
PS C:\> New-SysInfo -Computername Thinkp1,Bovine320,Srv1 | Format-Table -View computer


   Computername: THINKP1

Processes OperatingSystem               LastBootUpTime      Uptime
--------- ---------------               --------------      ------
230       Microsoft Windows 10 Pro 6/23/2020 8:46:39 AM 00.01:10:42


   Computername: BOVINE320

Processes OperatingSystem                LastBootUpTime      Uptime
--------- ---------------                --------------      ------
306       Microsoft Windows 10 Pro 6/11/2020 10:11:20 PM 11.11:46:02


   Computername: SRV1

Processes OperatingSystem                                LastBootUpTime      Uptime
--------- ---------------                                --------------      ------
32        Microsoft Windows Server 2016 Standard  6/22/2020 2:54:08 PM 00.19:03:17
```

New-SysInfo

This view is using autosizing which we could go in and remove. Then the view would use the defined column widths.

Adding to a Module

You can add as many format .psxml files to your module as you'd like. Instead of invoking `Update-FormatData` you can add the files to the module manifest. Normally this section is commented out.

```
FormatsToProcess = @('mything.format.ps1xml')
```

You can put the format file in the module root. Or some people like to keep them organized.

```
FormatsToProcess = @('formats\this.format.ps1xml','formats\that.format.ps1xml')
```

When the module is imported, the formatting files are automatically.

Your Turn

This is such a useful scripting tool that we want to make sure you try it out. It may feel awkward at first but after you do it a few times, and especially if you use `New-PSFormatXML`, you'll find yourself using it all the time.

Start Here

In the first part of this book you worked on a module to get machine information. The default output contained a lot of information.

```
PS C:\> get-machineinfo

ComputerName    : BOVINE320
OSVersion       : 10.0.19041
OSBuild         : 19041
Manufacturer    : LENOVO
Model           : 30C2CTO1WW
Processors      : 1
Cores           : 8
RAM             : 31.8933715820313
SystemFreeSpace : 92282933248
Architecture    : 64
```

Your boss doesn't want to see all of this by default. In the downloads for this chapter you'll find the TMMachineInfo module. Use that as your starting point. The module we're using has a class-definition for the custom object so the class name is the type name.

Your Task

At a minimum, create a formatting file that has a default table view that displays these properties:

- The computername
- The OS version
- The manufacturer
- The RAM formatted as an integer and displayed as `MemGB`

If want an extra challenge create a second table view called `hardware` that meets these specifications:

- The computername
- The manufacturer
- The model
- The number of processors labeled as `CPUs` and right justified
- The number of cores right justified
- The RAM formatted as an integer and displayed as `MemGB`and right justified
- The SystemFreeSpace formatted as an integer with a lable of `SysFreeGB` and right justified

Our Take

Using Jeff's function we can create a format file and add it to the module for the default.

```
$new = @{
    Path       = 'TMMachineInfo\tmmachineinfo.format.ps1xml'
    Properties = 'Computername', 'OSVersion', 'Manufacturer','RAM'
    ViewName   = 'default'
    FormatType = 'table'
}
Get-Machineinfo | New-PSFormatXML @new
```

We edited the xml file to make some minor adjustments such as this snippet:

```
<TableColumnHeader>
    <Label>MemGB</Label>
    <Width>5</Width>
    <Alignment>right</Alignment>
</TableColumnHeader>
...
<TableColumnItem>
    <ScriptBlock>$_.RAM -as [int]</ScriptBlock>
</TableColumnItem>
```

We tested by manually running `Update-FormatData`.

```
PS C:\> get-machineinfo -Computername thinkp1

ComputerName    OSVersion     Manufacturer    MemGB
------------    ---------     ------------    -----
THINKP1         10.0.19041    LENOVO             32
```

The default output is now our custom table. The object is still there.

```
PS C:\> get-machineinfo -Computername thinkp1 | Select-object *


ComputerName    : THINKP1
OSVersion       : 10.0.19041
OSBuild         : 19041
Manufacturer    : LENOVO
Model           : 20MD0029US
Processors      : 1
Cores           : 12
RAM             : 31.5475845336914
SystemFreeSpace : 128774402048
Architecture    : 64
```

We also created a second table view called ‘hardware’.

```
$new = @{
    Path       = 'TMMachineInfo\tmmachineinfo.format.ps1xml'
    Properties = 'Computername','Manufacturer','Model','Processors','Cores',
    'RAM','SystemFreeSpace'
    ViewName   = 'hardware'
    FormatType = 'table'
    Append     = $True
}
Get-Machineinfo | New-PSFormatXML @new
```

Here’s a peek at some of the file.

```
...
<Label>MemGB</Label>
<Width>5</Width>
<Alignment>right</Alignment>
</TableColumnHeader>
<TableColumnHeader>
<Label>SysFreeGB</Label>
<Width>10</Width>
<Alignment>right</Alignment>
</TableColumnHeader>
</TableHeaders>
<TableRowEntries>
<TableRowEntry>
<TableColumnItems>
    <TableColumnItem>
    <PropertyName>ComputerName</PropertyName>
    </TableColumnItem>
    <TableColumnItem>
    <PropertyName>Manufacturer</PropertyName>
    </TableColumnItem>
    <TableColumnItem>
    <PropertyName>Model</PropertyName>
    </TableColumnItem>
    <TableColumnItem>
    <PropertyName>Processors</PropertyName>
    </TableColumnItem>
    <TableColumnItem>
    <PropertyName>Cores</PropertyName>
    </TableColumnItem>
    <TableColumnItem>
    <ScriptBlock>$_.RAM -as [int]</ScriptBlock>
    </TableColumnItem>
    <TableColumnItem>
    <ScriptBlock>[math]::Round($_.SystemFreeSpace/1GB,4)</ScriptBlock>
    </TableColumnItem>
...
```

We modified the manifest:

```
FormatsToProcess = @('tmmachineinfo.format.ps1xml')
```

With the custom views we have a more flexible and easy to use tool.

```
PS C:\> get-machineinfo thinkp1, bovine320,srv1 | format-table -view hardware

ComputerName    Manufacturer         Model                CPUs  Cores MemGB  SysFreeGB
------------    ------------         -----                ----  ----- -----  ---------
THINKP1         LENOVO               20MD0029US              1     12    32   119.9286
BOVINE320       LENOVO               30C2CTO1WW              1      8    32    85.9495
SRV1            Microsoft Corpora ... Virtual Machine        1      1     1   115.5887
```

Hardware View

The solution version of the module is in the download material for this chapter if you want to see all the changes we made.

Let's Review

1. Why would you want to create a custom formatting file?
2. What file extension do you use with a custom formatting file?
3. What command can you manually run to load your custom formatting?
4. What module manifest setting do you need to modify to load your custom formatting?

Review Answers

1. You have a rich object but only really need to see a subset of properties by default.
2. The extensions is `.ps1xml` although the naming convention is `<typename>.format.ps1xml` to make it super clear.
3. `Update-FormatData`
4. `FormatsToProcess`

Adding Logging

Even though we've touched on this topic elsewhere in the book, it comes up so frequently that we felt we had to devote a chapter. We can't tell you the number of times people ask about how to add logging to their function or they want to have a paper trail or their boss is being unreasonable. Whatever the case, hopefully we can shed some light on this subject and at least get you headed in the right direction.

We'll tell you right now that there is no magic or hidden log command. You'll have to build it. Which shouldn't be that difficult once you finish this book. But we need to start with a question many IT Pros seem to over look.

Why Are You Logging

Let's say you have written a killer function that has gotten you rave reviews from your boss and a big fat raise. But now the boss says, "We need to add logging to the command." Without sounding flippant, your response should be "Why?". The process for adding "logging" to your command is no different than the command itself. You need to gather some business requirements.

- Who is going to look at the logging results?
- Where will logging results be stored?
- How will the logging results be accessed or utilized?
- How long will logging results need to be maintained or kept?
- Will logging be optional or automatic?
- What mechanism or process will be put in place to manage the logging results?
- How sensitive are the logging results?
- What data needs to be logged?
- What are the implications if logging fails?

Your choice of scripting technique and commands will vary depending on these answers. Building a long term logging result for independent auditor will likely entail something different than a need to log offline or unreachable computers.

Logging or Transcript

During the course of requirements gathering you should at some point need to determine if you need a *log* or *transcript*. People will say they need logging when what they really want is a transcript of

who ran the command and what happened. To our way of thinking, a *log* is some artifact that records specific pieces of information. A *transcript* is a written log of an interactive session or command.

PowerShell already has a built-in transcript feature and it supports nested transcripts. Here's a taste of how you might take advantage of this feature.

Get-Bits

```
Function Get-Bits {
  [cmdletbinding()]
  Param([string[]]$Computername = $env:computername)

  Begin {
    $file = "{0}_{1}.txt" -f (Get-Date -f "yyyy_MMddhhmm"),
    $($myinvocation.MyCommand)
    $tfile = Join-Path -Path $env:temp -ChildPath $file
    [void](Start-Transcript -Path $tfile)
    Write-Verbose "Starting $($myinvocation.MyCommand)"
    $PSBoundParameters | Out-String | Write-Verbose
  }
  Process {
    foreach ($computer in $Computername) {
      Write-Host "Getting BITS from $computer" -ForegroundColor green
      Get-Service -Name Bits -ComputerName $computer
    }
  }
  End {
    Write-Verbose "Ending $($myinvocation.MyCommand)"
    [void](Stop-Transcript)
  }

} #end function
```

When someone runs the command, it will create a transcript file in the %TEMP% folder using a naming convention of a time stamp and the command name. All output from the command will be captured in the transcript file. It is up to you to figure out how to manage or use the transcript. One bonus about using a transcript is that it will include metadata about the user and their PowerShell session. You don't have to code anything extra.

PowerShell also has other native logging features that may be what you really need. In PowerShell 7 you can read the help topic about_logging_windows[84] or follow the online link. As with transcripts, you'll have to figure out how to manage the logged results.

[84] https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_logging_windows?view=powershell-7

Structured vs Unstructured

So let's say you've decided you need some type of logging, which doesn't mean you can't also add a transcript feature. And you've thought about what pieces of information you need to record and in what situations. Next you should decide what format to use. Do you need plain, unstructured results like a text file of events? Or do you need some sort of structured data like an XML or JSON object? Structured data tends to be something you can search or more easily manipulate than plain text. Or maybe you need to write log stuff to a SQL database.

This is where the answers to the questions we posed earlier come into play. If you need secure, long-term storage with reporting capabilities, writing to a SQL database may be the smart move. Or maybe you have your command emit a CSV file and with a separate process, you suck up all the CSV logs into a central database. Bigger companies can probably afford bigger tools so you'll have to figure out what works for you.

Write-Information

We thought it might help to revisit the `Write-Information` command. This is a command that probably doesn't get the love it deserves because most people don't know about it or how to use it. Here's a revision to the sample `Get-Bits` function that uses `Write-Information`.

Get-Bits with Write-Information

```
Function Get-Bits {
  [cmdletbinding()]
  Param(
    [Parameter(Position = 0, ValueFromPipeline)]
    [string[]]$Computername = $env:computername
  )

  Begin {
    $start = Get-Date
    $msg = "[$start] Starting $($myinvocation.MyCommand)"
    Write-Verbose $msg
    Write-Information $msg -Tags meta,begin

    $count = 0
    $errorcount = 0
    $PSBoundParameters | Out-String | Write-Verbose

    $cim = @{
      ClassName    = 'Win32_Service'
      Filter       = "name='bits'"
```

```
      ErrorAction  = "Stop"
      Computername = ""
    }
  }
  Process {
    foreach ($computer in $Computername) {
      $count++
      $cim.Computername = $Computer
      Write-Information "Query $computer" -Tags process
      Try {
        Write-Host "Getting BITS from $computer" -ForegroundColor green
        Get-CimInstance @cim |
        Select-Object @{Name = "Computername";Expression = {$_.SystemName}},
        Name, State, StartMode
      }
      Catch {
        $errorcount++
        $msg = "Failed to query $computer. $($_.exception.message)"
        Write-Warning $msg
        Write-Information $msg -Tags process, error
      }
    }
  }
  End {
    $end = Get-Date
    $timespan = New-Timespan -start $start -end $end
    $sum = "Processed $count computer(s) with $errorcount error(s) in $timespan"
    Write-Information $sum -Tags meta,end
    $msg = "[$end] Ending $($myinvocation.MyCommand)"
    Write-Information $msg -Tags meta,end

    Write-Verbose $msg
  }

} #end function
```

The tags parameter lets you add whatever tagging taxonomy you'd like. If you are writing PowerShell in a team environment, we encourage you to define a standard set and usage guidelines. The information commands are ignored unless the user running your command knows enough to use the common information parameters.

```
$r = "thinkp1","srv1","srv3","bovine320" |
Get-Bits -InformationVariable iv -Verbose
```

All of the information records generated by `Write-Information` are stored in $iv. These are objects with their own set of properties.

```
PS C\> $iv | Select-Object -property TimeGenerated,MessageData,tags

TimeGenerated        MessageData
-------------        -----------
6/23/2020 1:15:49 PM [06/23/2020 13:15:49] Starting Get-Bits
6/23/2020 1:15:50 PM Query thinkp1
6/23/2020 1:15:50 PM Getting BITS from thinkp1
6/23/2020 1:15:50 PM Query srv1
6/23/2020 1:15:50 PM Getting BITS from srv1
6/23/2020 1:15:52 PM Query srv3
6/23/2020 1:15:52 PM Getting BITS from srv3
6/23/2020 1:15:54 PM Failed to query srv3. The WinRM client cannot process ...
6/23/2020 1:15:54 PM Query bovine320
6/23/2020 1:15:54 PM Getting BITS from bovine320
6/23/2020 1:15:55 PM Processed 4 computer(s) with 1 error(s) in 00:00:05.1028159
6/23/2020 1:15:55 PM [06/23/2020 13:15:55] Ending Get-Bits
```

Or you can filter on tags.

```
PS C:\> $iv | Where-Object tags -contains meta |
Select-Object -property TimeGenerated,User,MessageData |
Format-Table -GroupBy User -Property TimeGenerated,MessageData

   User: BOVINE320\Jeff

TimeGenerated        MessageData
-------------        -----------
6/23/2020 1:15:49 PM [06/23/2020 13:15:49] Starting Get-Bits
6/23/2020 1:15:55 PM Processed 4 computer(s) with 1 error(s) in 00:00:05.1028159
6/23/2020 1:15:55 PM [06/23/2020 13:15:55] Ending Get-Bits
```

Due to some quirk with this object type, we had to pipe to `Select-Object` before we could use `Format-Table`.

The downside is that you are relying on the user. But here's a trick you might try to take advantage of `Write-Information`, which is structured data if you haven't figured that out, and persistent storage. In the `End` block of your function, insert code like this:

```
$xml = "$($env:computername)-{0}_{1}.xml" -f (Get-Date -f "yyyy_MMddhhmm"),
$($myinvocation.MyCommand)
$export = Join-Path -path C:\work -ChildPath $xml
$PSCmdlet.GetVariableValue($PSBoundParameters["InformationVariable"]) |
Export-Clixml $export
```

We're defining a location for a file that we're going to create by sending the information variable to `Export-CliXml`. Naturally, you need to ensure the location exists.

The second step is to provide a default value for the `InformationVariable`. In the module file or the .ps1 file that will be dot sourced, add an entry to PSDefaultParameterValues.

```
$PSDefaultParameterValues["Get-Bits:InformationVariable"]="myIV"
```

If the user runs `Get-Bits` without specifying the parameter, the default will be used, information records will be generated and exported at the end. If they specify the parameter, nothing changes. You still get your export, which the user won't know about, and they get their variable to work with.

As we've hopefully made clear, "adding logging" is more than adding a few lines of `Out-File`. Or at least it should be. We're giving you a lot of techniques and concepts in this book that you can use to build your own logging mechanism that meets your business requirements.

Toolmaking Tips and Tricks

We've been scripting and toolmaking since the earliest days of PowerShell. Throw in our experiences with VBScript and batch files and we've been automating since the days of dirt. We've shared as much of our experiences throughout the book, but there's always something else – some little tidbit that might make your work easier or enjoyable.

So without further fuss, and in no particular order of importance, here are some things to keep in mind during your PowerShell toolmaking adventures.

- We can't stress enough the importance of white space and formatting your code. Nobody wants to troubleshoot a 1000 lines of left-justified single-space code. It doesn't matter to PowerShell but it will matter to the next person who has to read or maintain your command. If you are using VS Code, take advantage of it's automatic formatting feature. Right-click on the open file and select "Format Document" from the context menu. Or use the `Alt+Shift+F` shortcut.
- When writing an expression with operators include spacing around the operator. This `$d=Get-Date` will work but `$d = Get-Date` is easier to read and is more likely to be parsed better, especially in the PowerShell ISE. In earlier versions of PowerShell the parser worked better with spaces but regardless, you should always keep readability in mind and a little extra white space never hurts.
- Do NOT use archaic prefixes for variable names like `$strComputername`. That is so 20th century and VBScript-ish. For that matter, we can't see any reason to use anything but alphanumeric characters in variable names. `$Computername` is definitely better than `$_Computer` or `$Computer-Name`.
- If you are using VS Code, you can open an entire directory right from your PowerShell session.

```
code c:\scripts\mycooltool
```

This will launch VS Code and load the specified folder. You can then select files to edit from the file tree.

- Another VS Code related tip is to configure the editor to treat any new file as a PowerShell file. If you create a new file (`Ctrl+N`), look in the lower right corner of the status bar to see what language VS Code is using. If it does not say PowerShell, open up your preferences (`Ctrl+,`). In the search box look for `files: default language`. It might be hard to detect depending on your theme, but just under `The default language mode that is assigned to new files` there is a text box. Type in `powershell`.

The change is immediate. Now, every time you create a new file VS Code will treat it as a PowerShell file which means you'll get all the PowerShell-related functionality like snippets and command completion.

- You might develop your commands in the PowerShell ISE or Visual Studio Code, perhaps even running them in those tools. But you should test your commands from the PowerShell console, especially if that is where you expect them to be run.
- If you will be developing PowerShell tools in a team environment, agree on scripting conventions such as whether braces { }, go on the same line

```
Function Get-Awesomeness {
<code> }
```

or after:

```
Function Get-Awesomeness
{
<code>
}
```

PowerShell doesn't care where your braces are, but some people do.

- When writing a construct that uses () or {}, especially when you expect multiple lines of code to be between them, type the opening and closing piece then go back and fill in the code between. VS Code will do this for you automatically which is another reason you might consider adopting it. Too often beginners will forget to put in the closing parentheses or brace and then get errors when running the command.
- Use `Write-Verbose` not only as a way to provide detailed feedback but also as internal documentation. We covered this in the chapter on adding verbose output.
- Build up the muscle memory to use tab-completion. Any PowerShell editor worth your time will offer some sort of command-completion feature. Use it.
- Do we really have to remind you to use full cmdlet and parameter names? You only have to write your command once and if you take advantage of tab completion it isn't even that much of a burden. Sure, some PowerShell cmdlets have unwieldy names, but that doesn't mean you have to manually type every character in the name. This is super-critical if you are developing, or plan to develop, tools that will work cross platform in PowerShell Core on non-Windows systems.
- Get in the habit of reading full help and examples (don't forget the About topics), even for things you *think* you know. Content changes, bugs are fixed and sometimes you may gloss over something only later to go back and discover it when you really need it.

- Leverage snippets. Learn how to take advantage of the snippet or clip feature of your preferred scripting editor. The PowerShell ISE ships with a number of snippets which you can insert with `Ctrl+J` and you can add your own. Snippets keep your code consistent and make you more efficient.
- We prefer reading scripts vertically. By that we mean, try to avoid writing long expressions that force you to scroll horizontally. Splatting is a big help.
- Don't feel compelled to write long, complex pipelined expressions. Yes, we stress the importance of using the pipeline but sometimes your code will be easier to read (or debug) if you break a long command expression into several steps. Depending on what you are doing, it might even perform better. You might have a long pipelined command in your script like this:

```
Get-ChildItem ~\Documents -Directory | foreach-object {
$stats = Get-ChildItem $_.fullname -Recurse -File |
Measure-Object length -sum
$_ | Select-Object Fullname,@{Name="Size";Expression={$stats.sum}},
@{Name="Files";Expression={$stats.count}}
} | Sort Size
```

That's a pretty unwieldy chunk of code. Something like this might make more sense in a script:

```
$folders = Get-ChildItem -path ~\Documents -Directory
Write-Verbose "Found $($folders.count) top level folders"
#process each folder and save all results to a variable
$data = $folders | foreach-object {
    Write-Verbose "Processing $($_.fullname)"

    #measure the total size of all files
    $stats = Get-ChildItem -Path $_.fullname -Recurse -File |
    Measure-Object length -sum

    #write the custom object to the pipeline
    $_ | Select-Object Fullname,
    @{Name = "Size"; Expression = {$stats.sum}},
    @{Name = "Files"; Expression = {$stats.count}}
} #end foreach folder

#write sorted results to the pipeline
$data | Sort-Object -property Size
```

You can also see how much easier it is to insert comments and `Write-Verbose` commands.

- Be open to thinking outside the box or exploring alternative approaches. Use `Measure-Command` to test and compare code. Although don't assume faster code is always better code in your script, unless we're talking orders of magnitude. For example, if we measure how long it takes to run the code from the previous tip on Jeff's desktop it took 3.9 seconds. Then we tried code like this:

```
$folders = Get-ChildItem -path ~\Documents -Directory
Write-Verbose "Found $($folders.count) top level folders"

#process each folder and save all results to a variable
$data = foreach ($folder in $folders) {
    $stats = (Get-ChildItem -path $folder.fullname -file -Recurse |
    Measure-Object -Property length -sum)
    #create a custom object for each top-level folder
    [pscustomobject]@{
        Path  = $folder.FullName
        Files = $stats.count
        Size  = $stats.sum
    }
} #foreach folder
$data | Sort-Object -property Size
```

This code uses the `ForEach` enumerator in place of `ForEach-Object` and creates a custom object instead of relying on `Select-Object`. The end result is the same but this code took 3.6 seconds to complete. Is this approach better for saving 300 milliseconds? Is it better for you? That's for you to figure out. You might test with different folders sizes. You might decide based on how easy it is to read the code. You might decide based on what else you are considering adding to the code. The point is, be open to testing alternative solutions.

Format Code

One of the best reasons for using VS code as your editor of choice is for its formatting features. We constantly talk about the importance of writing code that is easy to read. Using indentation and whitespace is a part of this. But we're all a little lazy. Fortunately, VS Code can save us from ourselves. Suppose you have a chunk of code that is all left justified and otherwise not as well-formatted as you would like. Open the file in VS Code, right click anywhere in the document and select `Format Document` from the context menu. Or you can use the keyboard shortcut 'Shift+Alt+F'. VS Code should then make your code "prettier" and more professional.

Note that VS Code needs to be able to identify your file as a PowerShell file. It also needs to be able to parse it without errors. If you have syntax errors or other problems that would prevent your code from running, formatting probably won't work until you fix the errors.

PowerShell scripting and toolmaking is a much an art as anything. We can't teach you to be "artistic" in your PowerShell scripting, but good mechanics and discipline will go a long way in making it an enjoyable and productive experience.

Part 6: Pester

We provided an introduction to Pester earlier in this book, but now we'd like to really dig deep. Pester is a pretty important part of the PowerShell universe these days, and if you're going to be a professional-grade PowerShell toolmaker, you should make Pester a big part of your world.

Why Pester Matters

In the world of DevOps and automation, it's crucial that your code - you know, the thing that *enables* your automation - be reliable. In the past, you'd accomplish reliability, or attempt to, by manually testing your code. The problems with manual testing are legion:

- You're likely to be inconsistent. That is, you might forget to test some things some times, which opens the door to devastating bugs.
- You're going to spend a lot of time, if you're doing it right (and the time commitment is what makes most people not "do it right" in the first place).
- You end up wasting time setting up "test harnesses" to safely test your code, amongst other "supporting" tasks.

This is where Pester comes in. Simply put, it's a testing automation tool for PowerShell code, as we explained earlier in this book.

- Pester is consistent. It tests the same things, every time, so you never "miss" anything. And, if you discover a new bug that you weren't testing for, you can add to your automated tests to make sure that bug never "sneaks by" again.
- Pester can be automated, so it takes none of your time to perform tests.
- Pester integrates well with continual integration tools, like Visual Studio Team Services (VSTS), Jenkins, Team City, and so on, so that spinning up test environments and running tests can also be completely automated.

The vision goes something like this:

1. You check in your latest PowerShell code to a code repository, like Git or VSTS. That code includes Pester tests.
2. A miracle occurs.
3. Your tested code is either rejected due to failed tests (and you're notified), or your code appears in a production repository, such as a NuGet repository where it can be deployed via PowerShellGet.

The "miracle" here is some kind of automated workflow. VSTS, for example, might spin up a test environment, load your code into it, and run your Pester tests against your code. We're not going to cover how to make the miracle work, as it's not really a PowerShell thing per se, and because there are so many combinations of options you could choose. We *are* going to focus on how to write those Pester tests, though.

The big thing here is that *you need to be writing testable code*, a concept we'll devote a specific chapter to. But if you're looking for the short answer on, "what is testable code?" It's basically "follow the advice we've been giving you in this book." Write concise, self-contained, single-task functions to do *everything*.

The other thing you'll want to quickly embrace is *to write your Pester tests immediately*, if not actually *in advance* of your code (something we'll discuss more in the chapter on test-driven development). This is going to require an act of will for most PowerShell folks, because we tend to want to just dive in and start experimenting, rather than worrying about writing tests. But the difference between the adults and the babies, here, is that the adults do the right thing because they know it's the right thing to do. Having tests available *from the outset of your project* is how you reap the advantages of Pester, and indeed of PowerShell more generally.

So that's why Pester is important. We don't think anyone should really write *any* code unless they're also going to write automated tests for it.

It's also important to understand what Pester really *does*, and this gets a bit squishy. First, it's worth considering the different kinds of testing you might want to perform in your life. Here are few, but by no means all:

- *Unit testing* is really just making sure your code *runs*. You want to make sure it behaves properly when passed various combinations of parameters, for example, and that its internal logic behaves as expected. You usually try to test the code in isolation, meaning you prevent it from making any permanent changes to systems, databases, and so on. You're also *just testing your code*, not anybody else's. If you code internally runs `Get-CimInstance`, then you *actually prevent it from doing so*, since `Get-CimInstance` isn't *your* code. Unit testing is what Pester is all about, and it contains functionality to help you achieve all of the above. The idea is to isolate your code as much as possible to make testing more practical, and to make debugging easier.
- *Integration testing* is a bit more far-reaching. It's designed to test your code *running in conjunction* with whatever other code is involved. This is where you'd go ahead and let internal `Get-CimInstance` calls run correctly, so make sure your code operates will *when integrated* with other code. Integration testing is more "for real" than unit testing, and typically runs in something close to a production environment, rather than in isolation.
- *Infrastructure validation* can be thought of as an extension to integration testing. Because so much of our PowerShell code is about modifying computer systems, such as building VMs or deploying software, infrastructure validation runs our code, and then *reaches out to check the results*. Pester can also be used for this kind of testing, and we'll get into it more later in this book.

All of this is important to understand, because it helps you better understand what a Pester test looks like. If you've written a function that's little more than a wrapper around `ConvertTo-HTML`, for example, then your Pester tests aren't going to be very complex, because you probably didn't write much code. You're not trying to make sure `ConvertTo-HTML` itself works, because that's not

your code, so it's not your problem in a *unit test*. Because *so much* of our PowerShell code is really leveraging other people's code, our own Pester tests are often simpler and easier to grasp.

Core Pester Concepts

Although we touched on Pester briefly earlier in the book, we want to take a step back and really dig into some of its core concepts. A lot of people, we find, get a bit intimidated by Pester, and it's mainly because they're overthinking what it does. We don't want that to happen to you, so please â€" start here.

Installing Pester

Pester actually shipped *with Microsoft Windows* for the first time in Windows 10 and Windows Server 2016. The problem is that the version shipping with the OS (3.4.0) is grossly outdated, and you can't update it. Instead, you have to install a new version. This chapter is based on Pester 4.2.0, which was released right about the time we wrote this section of the book.

Most of our content should still be relevant and even correct. You'll probably even find lots of Pester test examples in the wild. But be careful. In June 2020, Pester v5 was released with some major changes. The concepts which we'll cover haven't changed but some of the implementation details might. If you upgrade to v5 be sure to look at the release documentation[85]

Installing a new version isn't hard, as Pester is available in PowerShell Gallery, but because you're installing a *new version of a module that's already installed in Windows*, you have to take a couple of extra precautions. First, you must be running PowerShell **as Administrator** for this to work, meaning your console window must have "as Administrator" in the window title bar. Then, run this:

```
Install-Module -Name Pester -Force -SkipPublisherCheck
```

This will pull the latest Pester down from PowerShell Gallery and install it. From now on, you can update this newly installed version from the Gallery as new versions become available:

```
Update-Module -Name Pester
```

This will update the Gallery-installed version, not the "came with Windows" version. There's no need to delete the "came with Windows" version, and indeed the OS will usually try and put it back if you do delete it.

[85] https://github.com/pester/Pester

For versions of Windows that *don't* come with Pester, you can just run the `Install-Module` command above. For PowerShell versions earlier than 5, you may need to first install PowerShellGet from PowerShellGallery.com; PowerShellGet is where `Install-Module` comes from.

If you need it, https://pester.dev/docs/introduction/installation[86]] contains more information about installing Pester in other situations.

What is Pester

The previous chapter dug into what Pester is at a high level, but let's carefully present a technical definition:

Pester is a Behavior-Driven Development (BDD) Unit Test execution framework. Pester exposes a Domain Specific Language (DSL) for defining Unit tests, and a file naming convention that makes it easier to run tests in an automated fashion. Pester contains a set of Mocking functions, allowing it to mimic the functionality of any PowerShell command inside a test, thereby "faking" a command for the purposes of a test.

In short, Pester is a special set of PowerShell commands, written in PowerShell itself, which are used to define and run unit tests against your PowerShell code.

Pester's Weak Point

Pester's main weak point is that it's designed to run and mock *PowerShell commands*. We get a *lot* of people who incorrectly conflate "doing stuff in PowerShell" with "PowerShell commands." For example, the following is without a doubt a PowerShell script:

```
$Computer = 'COMPUTER1'
Try
{
  $filter = "(&(objectCategory=computer)(objectClass=computer)(cn=$Computer))"
  $ComputerObject = ([adsisearcher]$filter).FindOne()
  $CertStore = New-Object System.Security.Cryptography.X509Certificates.X509Store "\\
\$Computer\My", "LocalMachine" -ErrorAction Stop
  $CertStore.Open([System.Security.Cryptography.X509Certificates.OpenFlags]::ReadOnl\
y)
  If ($CertStore.Certificates) {
    Foreach ($Cert in $CertStore.Certificates)  {
        ### PERFORM ACTION WITH EACH CERT... ###
        }
    }
```

[86] https://pester.dev/docs/introduction/installation

```
}
Catch{
    ### CATCH ERRORS ###
}
```

But you'll notice that there are basically no *actual PowerShell commands* being run, there. It's all .NET Framework classes. Yes, this is *in* PowerShell, but if we may be deeply philosophical for a moment, this is not *of* PowerShell. It's more like a C# program translated into PowerShell script. Pester will not be great at unit-testing this.

Instead, we'd suggest *wrapping all of the above into PowerShell functions*. That's essentially been the theme for this entire book, right? *Write PowerShell functions*. Anything not already a PowerShell command (that is, a cmdlet or a function, for our purposes) should be made into a PowerShell command. No "raw" .NET Framework unless it's wrapped in a PowerShell command. PowerShell commands maintain the consistency of PowerShell's naming conventions and behaviors, and as a bonus are exactly what Pester can help you unit test.

So calling this "Pester's weak point" is actually unfair of us. If you're doing the right thing in PowerShell, then this isn't a "weak point" in Pester; if you're *not* doing the right thing in PowerShell, then this is *your* "weak point," not Pester's.

Understand Unit Testing

Unit testing, which is Pester's first and main use case, is designed to do specific things. While we're also going to show you (a bit later) some other kinds of tests Pester can do, it's important to understand what unit testing is, what it is meant to do, and what it *isn't* meant to do. Without really buying into the scope of what unit testing is for, it's easy to go down a rabbit hole with Pester and spend all your time trying to do stuff that you're really not meant to.

A unit test is very explicitly not meant to modify anything in the environment. It's not supposed to talk to the network, make changes to a database, or anything else. Yes, those activities may introduce different kinds of errors, and you do need to test for those, but that's an *integration test*, not a unit test. Unit tests are meant to be as self-contained as possible, and that's actually where *mocks* come into play. If you're writing a command which uses `Get-DataFromDatabase` (a second command you or someone else wrote), then you would *mock* `Get-DataFromDatabase` in your unit test. The mock would return some static "dummy" data, making it seem as if `Get-DataFromDatabase` was running, but in fact not actually running it. This way, your unit test is *only testing* ***your*** *code*, not the code from `Get-DataFromDatabase` as well. `Get-DataFromDatabase` would, presumably, have its own units tests to test its code.

There's a great comparison chart between unit testing and integration testing at https://www.guru99.com/unit-test-vs-integration-test.html[87], and it's worth a few minutes to read it. Unit testing isn't *meant* to ensure the Total Correct Functionality of code you write; it's meant to catch specific kinds of

[87] https://www.guru99.com/unit-test-vs-integration-test.html

problems. Integration tests may be necessary, but they're a distinct thing, harder to write, and harder to execute, and harder to maintain. So we start with unit tests.

Scope

Compared to stricter, more full-fledged programming languages, PowerShell is pretty lightweight when it comes to *scope*. Basically, the shell itself is a global scope, any scripts you run get their own script scope, and the inside of any functions is an independent scope. But that's it: inside a `ForEach` loop, for example, isn't a distinct scope. So PowerShell coders aren't usually accustomed to thinking a whole lot about scope.

Pester provides a rich scoping mechanism. It includes three basic structures, which we'll discuss in the coming chapters: `Describe`, `Context`, and `It`. Each of these represents their own scope, meaning certain things done within those structures "vanish" when the structure is finished executing. It's important to pay attention to that scoping as you go, because you can create some pretty unexpected results if you don't. We'll dig into the specifics as we hit each structure, but we wanted to call out the importance of paying attention to scope right up front.

Here's a simple way to think about scope: how would you manually test a function that you've written? You might start by running it with a particular set of parameters, providing sample input to each of them, and then examining several pieces of output to make sure they were as you expected. You might then run it a second time with similar parameters but slightly different input, and again check the output. Then you might run it in a completely different way, with totally different parameters and input, and check the output again. Pester actually provides *structures* where you'd define those three tests, and they might very well end up being *different scopes*, depending on how you needed to manage the input data between those three test runs.

Sample Code

For the remainder of our Pester chapters, we're going to use the following as the function we're writing unit tests for. You'll find this in the downloadable sample code, if you want to pop it open in VS Code and follow along with us.

Get-ServiceRemote

```
function Get-ServiceRemote {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory,
                   ValueFromPipeline,
                   ValueFromPipelineByPropertyName
                   )]
        [string[]]$ComputerName,

        [Parameter(ValueFromPipelineByPropertyName)]
        [Alias('Name')]
        [string[]]$ServiceName
    )

    if ($PSBoundParameters.ContainsKey('ServiceName')) {
        Invoke-Command -ComputerName $ComputerName -ScriptBlock {
            Get-Service -Name $using:ServiceName
        }
    } else {
        Invoke-Command -ComputerName $ComputerName -ScriptBlock {
            Get-Service
        }
    }
}
```

This isn't meant to be a fancy script. It's just designed for a PowerShell 7 world, where `Get-Service` no longer has a `-ComputerName` parameter. Instead, it's making a convenient replacement for `Get-Service` that uses PowerShell Remoting under the hood. It will default to retrieving all running services from each specified computer, and it can optionally retrieve only a specified list of services.

Let's go ahead and get a basic Pester test file set up for this script, and add this script to a file.

New-Fixture

The Pester module includes a command that makes it brain-dead easy to scaffold up TDD environment. The Pester paradigm is that your tests are written **before** your code so the `New-Fixture` command will create the outline of a Pester test and a file for your command. Because these files are considered a discrete unit, they will be created in a separate directory. You need to provide the path to that directory and the name of the command you are creating. Pester will create the folder if it doesn't exist.

If you end up using the Plaster module template we provide this will add the testing framework for you.

```
New-Fixture -path c:\tools -name Get-ServiceRemote
```

This will create two files in C:\tools: Get-ServiceRemote.ps1 which will be the script file and Get-ServiceRemote.Tests.ps1 which in the unit test file. The script file is nothing more than an empty Function declaration:

```
Function Get-ServiceRemote {

}
```

You can copy the sample code from the downloads into this file. The Tests file handles loading the command (which you supposedly haven't written yet) and defines a simple `Describe` block with a sample test.

```
$here = Split-Path -Parent $MyInvocation.MyCommand.Path
$sut = (Split-Path -Leaf $MyInvocation.MyCommand.Path) -replace '\.Tests\.', '.'
. "$here\$sut"

Describe "Get-ServiceRemote" {
    It "does something useful" {
        $true | Should -Be $false
    }
}
```

Pester assumes that the name of your test file begins with the name of the command you intend to test. The code before the `Describe` block is kinda optional and can be modified as needed. In our case since we are testing a standalone function it needs to be dot sourced. Later, when we get to testing a module, we might modify this to import the module being tested. Or add whatever other code you might need to setup your unit tests. But for now this should suffice.

Writing Testable Code

Before we go any further, we need to stress that Pester isn't designed to work with just any old script you might bang out. Like, you could *probably* get it to effectively test pretty much anything, but you'd have to put a lot of unnecessary effort into it.

Pester is designed to test *tools*. Modularized, self-contained functions that receive their inputs via parameters, and produce output to the PowerShell pipeline. That's Pester's bread and butter, and not incidentally it's the pattern *this entire book* has been pushing you to use (as does "Learn PowerShell Scripting in a Month of Lunches," this book's "prequel"). Pester will work great if you're following PowerShell's native patterns and practices which, again, is what this book has focused on. If you're just cranking out ad-hoc scripts with no structure, no parameters, and not really using the pipeline, then Pester isn't your guy. You're not *toolmaking*.

So before you even dip your toe into Pester-infested waters, stop and look at your code. Here are some warning signs that you've done things wrong:

Your code isn't a function. Pester is designed to test *tools*, which in PowerShell's world are *commands*, which in our specific case means *advanced functions*.

Your input doesn't come from parameters. 100% of a function's input should come from parameters. Not out-of-scope variables (although a module-level variable could, for example, be used as the default value of a parameter). Not from files (although part of your code's purpose might be to read files). Only parameters. Of course, some of those parameters might accept pipeline input, which is fantastic. But the point is that parameters represent the only way for data to enter your function and control its behavior.

Your output doesn't go wholly to the pipeline. This doesn't mean you can't produce other kinds of *messages*, by using commands like `Write-Verbose` or `Write-Error`; those aren't "output," though. You shouldn't be using `Write-Host`, unless the *only purpose of your command* is to display information on-screen. Now, sometimes, your functions might not produce any output. That's fine. Maybe your function accepts input and formats it into a file on disk (which, for our definition, is a *work product* and not *output* per se), or puts something into a database, but doesn't write anything to the pipeline. Fine. More than likely you have internal code that is using PowerShell commands to achieve those ends. *That* is something you can test with Pester.

Your function does more than one thing. The classic example of this is a function with a `-ComputerName` parameter to accept computer names, but also a `-FilePath` parameter, giving it the path of a text file which it will open and read computer names from. This is two different things, and the `-FilePath` thing should go away. You don't want the same kind of potential input (computer names, in this case) coming from multiple possible sources *within the function itself*. This isn't so much a "will break Pester" thing as it is a bad design pattern that will make testing (and debugging, and maintenance, and usage) harder than it should be.

If you can safely say that your code doesn't violate any of these patterns, then you're good to keep reading.

What to Test

We've touched on this a bit, but we wanted to pull out some more specific discussions are the idea of "what exactly do I test?" since it's an area that often gets Pester newcomers pretty wound up. We've some core guidelines that'll get you started.

Don't test other people's commands. Look, if `Get-ADUser` isn't functioning, there's not much you can do about it. You don't necessarily need to write a test to confirm that someone else's commands are working; *they* should have done that. So you'll usually *mock* other people's commands, so that they don't represent a "moving part" in your code that you have to worry about. In some cases, you may find that once you mock everyone else's commands, there's not much left to *your* command except parameter definitions. That's fine!

Don't just test the positive. You don't want to *only* write tests that makes sure things run the way you expect. You also need to write tests to verify "negative" conditions. That means feeding bad parameters to your commands, to make sure everything behaves as it should. It means feeding incorrect data or object types to your parameters. It means forcing your command to run in a situation that'll generate an error. You want to test all the "use cases."

Test parameter conditions. A big part of Pester testing is making sure your own parameters behave the way you want. Make sure they're accepting the right object types, make sure parameter sets behave the way you want, etc. If you've declared a parameter as mandatory, test to make sure that PowerShell is treating it that way. We don't do this because we mistrust PowerShell, we do it because if someone else gets into our code and removes the Mandatory attribute on a parameter, our test will catch that as a fail condition.

Describe Blocks

When you had Pester set up its basic template for tests, it created a single `Describe` block for you. This is where you'll start writing your tests.

We've discussed the importance of scope in Pester, and `Describe` is the "outermost" scope Pester offers. When you define *mocks*, or use Pester's TESTDRIVE: testing drive location, those things persist throughout the `Describe` block and then cease to exist after the `Describe` block finishes. In many cases, you can probably get away with a single `Describe` block for all of your tests. However, in many other cases you'll end up breaking your tests into different `Describe` blocks, within the same script file, so that you can separate the mocks and TESTDRIVE: uses of different test scenarios. Other times, with especially large test sets, you may use multiple `Describe` blocks simply to help keep your various test scenarios organized.

A simple `Describe` block comes with just a name, that... well, *describes* the scenario you're testing.

```
Describe "Get-ServiceRemote" {
}

Describe "Do-Something with database input" {
}

Describe "Do-Something with file input" {
}
```

You can also *tag* these blocks. Doing so allows you to run only those block having a certain tag. This is *really* useful when, for example, you're developing a whole bunch of tests and you only need to be able to run certain sets of them at a time. Group those sets into `Describe` blocks that have tags:

```
Describe -Tag "FileTests" "Get-ServiceRemote" {
}

Describe -Tag "FileTests","Remoting" -Name "Get-ServiceRemote2" {
}
```

Pester doesn't care if you define tags before or after the description. But you should try to be consistent.

In the second example, there, we explicitly used the `-Name` parameter, which we didn't do previously.

And that's kind of all there is to `Describe`. It's not meant to be complex or complicated; it's just the outermost holder that Pester works with. A `Describe` is the smallest unit of work Pester can execute; when you run tests, you'll run at least one `Describe`, and everything it will execute.

In Pester terms, a `Describe` block can contain `Context` blocks and `It` blocks, both of which we'll discuss in the upcoming chapters. But more broadly, a `Describe` block can contain pretty much any other PowerShell code you might want, and it will execute those top-to-bottom, just like any PowerShell script. For example, prior to executing any tests, you might create a sample CSV file in the Pester TESTDRIVE:, and then use that CSV file in the tests that follow.

Context Blocks

The `Context` block is one of two Pester structures, in addition to `It`, that can live in a `Describe` block.

You can kind of think of `Context` as a mini-`Describe` that lives inside a `Describe`. That is, like a `Describe` block, a `Context` block acts as a scope for your tests. Mocks and the contents of TESTDRIVE: *which are created within the `Context` block* will be cleared after the `Context` block reaches its end. It's worth noting that the TESTDRIVE: business is a little tricky. Files *added* to TESTDRIVE: inside a `Context` will be removed; files *removed* inside a `Context` will not miraculously come back to life, and changes to files already in TESTDRIVE: will persist.

A `Context` block, in Pester terms, can contain `It` blocks, and the purpose of `Context` is to logically group some number of `It` blocks that require a shared scope. A `Context` block, like a `Describe` block, can also contain other PowerShell commands if needed.

It's entirely possible to *not* use `Context` at all. If your tests are small and only require a single scope, then you might not find any reason to sub-group them in `Context` blocks. If you only have one `Context` block, then you didn't really need it; the containing `Describe` block would have been sufficient.

BeforeEach and AfterEach

This is a good time to discuss the `BeforeEach` and `AfterEach` structures, although they're valid in a `Describe` block as well as within a `Context` block. These structures are scoped; if you define them in a `Context`, then they apply only to that `Context`, which is why we *usually* see them in a `Context` and not in a `Describe`.

These are designed to let you define set-up and tear-down code, which will run *before each* `It` block and *after each* `It` block. Say, for example, that each little test you run requires a database connection. You want to make a fresh connection for each test, for some reason - perhaps to ensure each one is completely isolated from the others. So in a `BeforeEach`, you'd set up the connection, in the `AfterEach`, you'd close the connection, and use the connection in each `It` block. That'd help modularize the connect/disconnect code into a central place. Any variables defined in a `BeforeEach` or `AfterEach` are valid within your `It` blocks.

`BeforeEach` and `AfterEach` can be defined in both `Describe` and `Context`, as we've already mentioned. We traditionally put them at the top of whatever block they're within, because that keeps them visible - otherwise, we worry about forgetting they exist and screwing something up. That's important to remember: even if an `If` block precedes your `AfterEach` and `BeforeEach`, they will still apply to that `It` block. No matter where `BeforeEach` and `AfterEach` are defined, they apply

to the entire `Describe` or `Context` they live in. That's why we like to keep them at the top, as a reminder that they're there.

In the event that you define these in *both* a `Context` and its containing `Describe`, the order of execution goes like this:

1. The `Describe`-level `BeforeEach`
2. The `Context`-level `BeforeEach`
3. The `Context`-level `AfterEach`
4. The `Describe`-level `AfterEach`

These are simple to define:

```
Describe "Get-ServiceRemote" {

  BeforeEach {
  }

  AfterEach {
  }

  Context {

    BeforeEach {
    }

    AfterEach {
    }

   }

}
```

It Blocks

The `It` block is the heart of Pester. This is an actual *test*. The `Describe` and `Context` blocks we've discussed up to this point are all about containing, organizing, and scoping tests; the `It` block is the test itself (and must live inside either a `Describe` or a `Context` block).

In testing lingo, the `It` block is where you define *assertions*, and each `It` block should normally contain one *assertion*. Think of an assertion as an English statement, such as, "this command will return two objects if I run it and provide two computer names." It is your expectation. The `It` block is where you actually run it with those two computer names, and then see if it indeed behaves as you have asserted. `It` blocks get a name, which should pretty clearly describe what's been asserted:

```
It "returns two objects when run with two computer names" {
}
```

The `It` block is not the assertion itself; it's *stating what the assertion will be*, and providing a small container for the test code to live in. `It` blocks have to end in one of 5 states:

- **Passed**. The code executed, and whatever was asserted was indeed true.
- **Failed**. The code executed, but its behavior was other than what was asserted.
- **Skipped**. The test was skipped because you told it to. More on that in a bit.
- **Pending**. The test was empty, or you explicitly marked it as `-Pending`. This is a neat trick when you're developing a bunch of tests - you can kind of sketch them all out as `It` blocks, mark them as `-Pending`, and then fill them in as you go.
- **Inconclusive**. The test was set to this status by using the `Set-TestInconclusive` command within the `It` block itself.

All `It` blocks have a name, and you can explicitly use the `-Name` parameter if you like:

```
It -Name "returns two objects when run with two computer names" {
}
```

But a lot of people don't use the parameter name, simply because not doing so lets the whole thing read as an English sentence:

```
It "rubs lotion on itself or it gets the hose" {
}
```

There are some other parameters you can play with:

The **`-Test`** parameter is what holds the actual code. As with the `-Name` parameter, it's rare to see this explicitly used, but it would look like this:

```
It -Name "returns two objects when run with two computer names" -Test {
}
```

Omitting both the `-Name` and `-Test` parameter names makes the `It` block read more like a PowerShell construct, like an `If` construct, which is what `It` is pretending to be.

The **`-Skip`** parameter is just a switch, and it tells Pester to skip the `It` block. You'd normally do this instead of just "commenting out" an unused test, because it keeps the test explicitly listed, as "Skipped," in the test output. Similarly, **`-Pending`** will also skip the test and output it as "Pending."

The **`-TestCases`** parameter is the most complex, and requires that you be familiar with the PowerShell concept of *splatting*. As a reminder, splatting is a way of bundling up a command's parameters into a dictionary or hash table. So, instead of running this:

```
Get-CimInstance -Class Win32_Service `
 -ComputerName SERVER1 `
 -Filter "Name LIKE '%svchost%'"
```

You could instead do this:

```
$params = @{
    Class = "Win32_Service"
    ComputerName = "SERVER1"
    Filter = "Name LIKE '%svchost%'"
    }
Get-CimInstance @params
```

The `-TestCases` parameter takes an array of those dictionary objects. So, you'd define several variations of our $params variable, feed them to `-TestCases`, and the `It` block would automatically repeat one time for each dictionary object you fed in. You include placeholders in the `It` block's name to "pull in" those values, so that your test output will clearly show what's happening. Each time the `It` block runs, a new dictionary will be splatted to the test block, allowing you to use those values. You'll need to construct a `Param()` block, inside the test code and with matching parameter names, for this all to work. Here's an example (pretend that this is inside a `Describe` block):

```
$params[0] = @{CN='SERVER1';Class='Win32_Service'}
$params[1] = @{CN='SERVER2';Class='Win32_Service'}
$params[2] = @{CN='SERVER2';Class='Win32_Process'}
It "Gets <Class> for <CN>" -TestCases $params {
    Param($CN,$Class)
    Get-CimInstance -Computer $CN -Class $Class
}
```

This example would run three times. The hashtable values will be splatted as parameters to the test code. As an added bonus Pester will also pass the values to the `It` statement.

```
Describing Foo
 [+] Gets Win32_Service for SERVER1 66ms
 [+] Gets Win32_Service for SERVER2 14ms
 [+] Gets Win32_Process for SERVER2 20ms
```

Now, it's **super important** to realize that this is a *partial example*. We've not actually made any assertions; we've just set up the structure in which we could do so. We'll play with this a bit more in the next chapter, though, which is where those assertions actually get made.

Should and Assertions

When you run code inside an `It` block, you normally examine the output of your code to see if it's what you expected. This examination is called the *assertion* of the test. It's where you lay out what you *think* should have happened. Pester's convention is to *throw a terminating error*, using the PowerShell `throw` command, if the *assertion was not met*. It's kind of a, "no news is good news" attitude; if the assertion failed, you throw an error. If the assertion succeeded, you don't do anything.

It is **hugely important** to realize that you can **run any code you want** in the `It` block to examine the output of whatever it is you're testing. All you need to do is throw an exception if things aren't up to snuff. However, for convenience and better readability, Pester provides the `Should` command, along with a variety of comparison operators. The `Should` command accepts your assertion, and you're meant to pipe output to it. If that output and the assertion match, `Should` doesn't do anything. If they don't match, `Should` throws a terminating exception. Let's take this really simple example:

```
It "returns 4" {
 $var = 2 + 2
 If ($var -ne 4) {
   Throw "was not 4"
 }
}
```

This is completely legal, but it's a little harder to read than this:

```
It "returns 4" {
 $var = 2 + 2
 $var | Should -Be 4
}
```

Internally, `Should` is basically doing the same `If` logic with a `Throw`, but it reads more easily, making the test itself easier to understand, follow, and maintain. `Should` supports a universal `-Not` switch, which looks like this:

```
It "doesn't return 4" {
 $var = 2 + 3
 $var | Should -Not -Be 4
}
```

Let's return to the example from last chapter, and add some assertions:

```
$params[0] = @{CN='SERVER1';Class='Win32_Service';Count=100}
$params[1] = @{CN='SERVER2';Class='Win32_Service';Count=100}
$params[2] = @{CN='SERVER2';Class='Win32_Process';Count=100}
It "Gets <Class> for <CN>" -TestCases $params {
    Param($CN,$Class,$Count)
    (Get-CimInstance -Computer $CN -Class $Class).Count |
    Should -Be $Count
}
```

Here, we've added an addition parameter to each hash table, indicating how many objects we expect each command to return. We've used that in our `Should` assertion. So this will run three tests. Note how we contained our main command in parentheses, so that we could access the `Count` property of the array that the command should return.

Should Operators

The power of `Should` comes in its various operators. This is not an exhaustive list.

- **-Be**. Tests for equality - not case-sensitive for strings.
- **-BeExactly**. Same as -Be, but case-sensitive for strings.
- **-BeGreaterThan**. Greater than.
- **-BeLessThan**. Less than.
- **-BeIn**. Tests to see that the piped-in value is contained in an array you pass, such as `'Don' | Should -BeIn @('Jeff','Don')`.
- **-BeLike**. Supports wildcard matches, just like PowerShell's `-like` operator. Not case-sensitive.
- **-BeExactlyLike**. Same as `-BeLike`, but case-sensitive.
- **-Exist**. Expects you to pipe in a path. This needn't be a file path, but can instead be any path available in any PSDrive, such as a registry key. Checks to see if the path exists.
- **-FileContentMatch** checks the filename that you pipe in, to see if it contains the content you specify. `'c:\test.txt' | Should -FileContentMatch "this text"`. This comparison is not case-sensitive, and it uses standard .NET regular expression syntax. If you're piping in a string, as we did here, it *must* be quoted or you'll get an error.
- **-FileContentMatchExactly**. Same as the above, but case-sensitive.
- **-Match**. A regular expression match. Not case-sensitive.
- **-MatchExactly**. Same as the above, but case-sensitive.
- **-Throw**. This is a fun one! You pipe a script block to it. It will run the block, and if the block throws an exception, then the assertion is passed. This is great for those, "I need to make sure this situation causes an error" scenarios. For example, `{NotA-Command} | Should -Throw` will pass, because `NotA-Command` isn't a command, and PowerShell will throw an exception when it tries to run it and can't.
- **-BeNullOrEmpty**. Checks that whatever you piped in is $null, or an empty array. Good for those situations where, "I need to make sure this command doesn't return anything."

Just bear in mind that *you don't have to use* `Should`; if you have some situation that one of these operators doesn't cover, you can code up whatever logic you need, and just `Throw` an exception to indicate a failed assertion.

See https://pester.dev/docs/usage/assertions[88] for more details.

[88] https://pester.dev/docs/usage/assertions

Mocks

Mocks are, for us, the heart and soul of what makes any testing framework so useful. To understand the purpose of mocks, you first have to embrace something that a lot of people don't always take to easily:

The purpose of unit testing is to test **your code**, not someone else's.

Consider our sample function:

Get-ServiceRemote

```
function Get-ServiceRemote {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory,
                   ValueFromPipeline,
                   ValueFromPipelineByPropertyName
                   )]
        [string[]]$ComputerName,

        [Parameter(ValueFromPipelineByPropertyName)]
        [Alias('Name')]
        [string[]]$ServiceName
    )

    if ($PSBoundParameters.ContainsKey('ServiceName')) {
        Invoke-Command -ComputerName $ComputerName -ScriptBlock {
            Get-Service -Name $using:ServiceName
        }
    } else {
        Invoke-Command -ComputerName $ComputerName -ScriptBlock {
            Get-Service
        }
    }
}
```

`Get-Service` isn't our code. We didn't write that, Microsoft did. So we're not going to try and test it. If it's broken, we can't fix it. Because it's not ours, we need to somehow "remove it" from our test â€" and that's what a mock lets us do. We can *mock*, or *fake out*, that command for our testing purposes.

Rather than running the real `Get-Service`, we'll run a fake version that returns a predetermined output.

You're going to want to create a mock for any commands in your script that aren't yours, and that aren't the specific subject of a test.

Where to Mock

The next question is, "where do I put mocks?" You've got three basic choices:

- In a `Describe` (high level)
- In a `Context` (mid level)
- In an `It` (low level)

Generally speaking, you want to put your mocks in the smallest scope that the mock will apply to. Keep in mind that you'll often code a mock to produce some predetermined output that the rest of your code will work with; that output might need to be different for different tests. So you might end up mocking a given command multiple times, with slightly different fake output each time.

A mock that will apply globally to all of your tests might best live in the top-level `Describe` block. If you've got a couple of `Context` blocks, and each one might need a different mock, then the `Context` would be the place for those mocks. If you've got a mock that will apply only to a specific test, then it might live inside the `It` for that test.

Defining a mock at a low level will override any mocks for the same command defined at a higher level. So, an `It` block mocking `Get-Service` would override any mocks for `Get-Service` appearing in the containing `Context` or `Describe` blocks.

If you are using Pester v5.x there might be some slight variations from what we are going to show you. You'll need to check the latest documentation for proper implementation.

How to Mock

The most basic mock requires the name of the command you are mocking and a scriptblock of code. The code in the scriptblock returns a hashtable with only the keys (fake properties) that you need. Here's how you do it:

```
Mock Get-Service {
  return @{'Name'='Svchost'}
}
```

Pretty easy! This will return a single object having a Name property, which will contain "Svchost."

Verifiable Mocks

You can also mark a mock as *verifiable*. It looks like this:

```
Mock Get-Service {
   return @{'Name'='Svchost'}
} -Verifiable
```

By itself, this does nothing. However, somewhere in one of your `It` blocks you can run `Assert-VerifiableMocks`. This command will then scan for any mocks that you've defined as `-Verifiable`, and make sure that each of those mocks has been run at least once. If it finds one that *hasn't* been run, it throws an exception. Inside of an `It` block, that exception causes the test to fail. This is kind of an easy way of making sure that the code you're testing ran through all of the code paths you wanted it to. It's a way of saying, "I want to check and make sure my code actually tried to run `Get-Service`, and if it didn't, I want to fail that test."

There's a similar command called `Assert-MockCalled` that's more granular and specific. It doesn't care about `-Verifiable` at all. Instead, you give it the name of the specific mock you're interested in, and the minimum number of times you wanted that mock to be called:

```
Assert-MockCalled Get-Service -Times 3
```

If the `Get-Service` mock was called fewer than three times, an exception is thrown. You can also make it check for an exact number of calls, meaning it can't be less *or more than* the times you specify:

```
Assert-MockCalled Get-Service -Times 3 -Exactly
```

Parameter Filters

This is a super-fancy addition to a mock. It's a bit like parameter sets in PowerShell, enabling you to define a different mock for the same command, based on the inputs passed to the command using the `-ParameterFilter` scriptblock. In the scriptblock you define a comparison using the mocked parameter name as a variable.

Here's a quick example:

```
Mock Get-Process { Return @{'Id'=1234;'Name'='svchost'} ) `
     -ParameterFilter { $Name -eq 'svchost' }
```

This mock will only run if `Get-Process -Name svchost` is run. If your code tries to just run `Get-Process` by itself, with no `-Name svchost`, then the mock wouldn't run in response to that. If you needed a mock that would apply to command without any parameters, you could define another mock:

```
Mock Get-Process { Return @{'Id'=789;'Name'='notepad'} )
```

With this mock, any `Get-Process` command in your code will get an object with a predefined ID and Name.

Mocking the Unmockable

A trick with mocks is that they can only "fake out" *PowerShell commands*. So if you've got code that runs this:

```
[System.Math]::Abs($x)
```

You can't mock that, because it's a .NET Framework static method, not a PowerShell cmdlet or function. That's why we hold firm to our opinion that *all* .NET Framework calls should be "wrapped" in a function:

```
Function Get-AbsoluteValue {
  Param(
    [float]$inputObject
  )
  [System.Math]::Abs($inputObject)
}
```

Now, we can mock the `Get-AbsoluteValue` command if we need to. By the way, this also applies to any command line utilities you might need to run. You cannot mock an expression like `whomai /user /fo csv` but you can if you wrap it in a PowerShell function.

Read the help topic `about_mocking` for more examples.

Pester's TESTDRIVE

So many operations require some sort of disk access that Pester provides a specific disk drive, the TESTDRIVE:, for that purpose. There are two main advantages to using TESTDRIVE: and, as we'll discuss, *only* use TESTDRIVE:, for file access during your tests.

1. Pester cleans up TESTDRIVE: automatically, so you're not leaving artifacts behind after your test, and each test starts with a "clean slate."
2. TESTDRIVE: exists wherever Pester runs, so even if you write tests on one machine and run them elsewhere (like in a continuous integration pipeline), you know TESTDRIVE: will be there for you.

If you're curious, Pester dynamically creates the TESTDRIVE: under your %TEMP% folder. In most situations this won't matter to you. But depending on your test, you may need to use a cmdlet like `Convert-Path` to resolve TESTDRIVE: to a "real" file path.

Clean Slate and Auto-Cleanup

TESTDRIVE: is well-scoped. What that means is, it "starts" existing when a `Describe` block runs, and it ceases to exist after that `Describe` block finishes. Further, once you enter a `Context` block, if you use those, Pester "tracks" what's done inside that block. Once the `Context` block ends, TESTDRIVE: reverts back to whatever it looked like when the same `Context` block started.

That reversion is a little less magical than you might think, though. It only applies to file *creation*. So, any file *created* inside a `Context` block will be deleted once that block finishes. Any files that exist *prior* to the `Context` block *that get changed inside the `Context` block* will remain changed after the block completes. Similarly, if you delete a file within a `Context` block, it stays deleted.

Working with Sample Data

One thing we deal with a lot when writing unit tests is the creation of test data. For example, suppose you have a command that's intended to take pipeline input from a CSV file. How should you do that? After all, the test data won't exist on TESTDRIVE:, and ideally, you shouldn't read test data from any other location because that would involve creating permanent artifacts on the testing system. So what do you do?

One option is to simply mock a command like `Import-Csv` so that, instead of reading an actual file, it just spews out test data that you hard-code into the mock itself.

Another option is to, at the start of a `Describe` or `Context` block, write out hard-coded test data to a file on TESTDRIVE:. You can then read or modify that data throughout your test as needed, knowing that it'll vanish once the block exits or the test is complete.

It may seem like "cheating" to hard-code test data into your tests, but we don't see it that way. We see it as making the tests more self-contained. It also helps the test preserve knowledge of past bugs. For example, one famous kind of bug involves people's last names being inserted into databases, using less-than-ideally-designed queries. A name like "O'Shea," with that single quote in the middle, can break those queries. Once you realize that, you can make sure that kind of name is included in your test data, preventing that kind of bug from ever happening again. That's really the ultimate goal of a unit test: to make sure bugs you've solved in the past never crop up unnoticed again.

Using TESTDRIVE

Use TESTDRIVE: just as you would any other drive. Instead of starting paths with `C:`, just start them with `TESTDRIVE:`. Here's a sample function that creates an HTML report.

New-DiskReport

```
Function New-DiskReport {
  [cmdletbinding()]
  Param(
      [Parameter(Mandatory)]
      [string]$Computername,

      [Parameter(Mandatory)]
      [ValidatePattern("\.htm(l)?$")]
      [string]$Path
  )

  $cimparams = @{
      ClassName    = 'Win32_logicaldisk'
      filter       = "drivetype=3"
      ComputerName = $Computername
  }

  $data = Get-CimInstance @cimparams | Select-Object -property DeviceID,
  VolumeName,
  @{Name="SizeGB";Expression={$_.size / 1gb -as [int32]}},
  @{Name="FreeGB";Expression={[math]::Round($_.freespace/1gb,4)}},
  @{Name="PctFree";Expression={[math]::Round(($_.freespace /$_.size) * 100,2)}}

  $htmlParams = @{
```

```
   Title = "$($Computername.toUpper()) Disk Report"
   PreContent = "<H1>$($Computername.toUpper())</H1>"
  }

  $html = $data | ConvertTo-Html @htmlParams

  Set-Content -Value $html -Path $Path
} #end function
```

We might build a Pester test to verify a file gets created.

Describe New-DiskReport

```
Describe New-DiskReport {

    Mock Get-CimInstance {
        return @{
            DeviceID   = "C:"
            Size       = 200GB
            Free       = 100GB
            VolumeName = "System"
        }

    } -ParameterFilter {$classname -eq 'win32_logicaldisk' -AND `
    $filter -eq "drivetype=3" -AND $computername -eq 'FOO'} -Verifiable


    New-DiskReport -Computername FOO -Path TESTDRIVE:\foo.html
    It "Should call Get-CimInstance" {
      Assert-VerifiableMock
    }

    It "Should create a file" {
      Test-Path -Path TESTDRIVE:\foo.html | Should be $True
    }

    It "Should throw an error with an invalid file extension" {
      {New-Diskreport -computername FOO -Path TESTDRIVE:\foo.ht} | Should Throw
    }
}
```

The html file is actually created in TESTDRIVE:. We aren't mocking the `Set-Content` cmdlet. As long as the test is running we could do whatever we wanted with the file such as testing to ensure

it is greater than 0 bytes or looking at the content. When the test finishes the drive is removed including our test file.

Pester for Infrastructure Validation

So far, we've discussed Pester's use as a unit testing framework. As we outlined in the beginning of this Part, unit testing tries really hard *to never make actual changes to the system*. That is, you try to exercise your code up to the point where something actually happens, and at that point, you try to use mocks so that you're not actually changing anything. The idea here is to isolate your code as much as possible from the external world, so that you're testing *just* your code.

But plenty of administrators need to go a bit further. *In addition* to their unit tests, they want to step up to letting their code *actually make changes*, and then testing to see if those changes were made as desired. That's what the community often refers to as *validation testing*, or specifically in the case of server and network infrastructure, *infrastructure validation*.

This might be creating a bunch of Active Directory users, and then verifying that they were in fact created correctly. Or it might involve configuring a remote server in a certain way, and then testing to see that the configuration "took" as expected. But you need to be a bit careful in how you scope these validation tests.

For one, think about *how* you're going to validate. For example, if you're writing code that uses `New-ADUser` to create a new user account, and then plan to use `Get-ADUser` to see if the accounts were really created… well what, exactly, are you testing? "I just want to make sure `New-ADUser` worked" is a poor answer, because *that's not your code*. If it's "I want to make sure that I fed the right data to the `New-ADUser` parameters," then that's a better answer, although you could potentially verify that by cleverly mocking `New-ADUser`. Anyway, what you don't want to do is put yourself in a position where you distrust All The Code Ever Written By Anyone, because your test workload will quickly balloon out of control. Think about *why* you're testing, and if the answer is, "I don't trust someone else's code," make sure you've a *reason* for that lack of trust beyond mere paranoia.

Spinning Up the Validation Environment

Of course, since you're going to be making changes to an actual environment, you'll need a test environment. This is part of what continuous integration frameworks like Team City and its ilk are for: they can help coordinate the spin-up and provisioning of virtual machines, which you can run your tests against and then de-provision. **In no circumstances should you run validation tests against your production infrastructure**. If spinning up a validation environment is going to be a bunch of manual tasks for you, then you're not ready for validation testing; this is only a good idea if you can automate the entire process from start to finish, and if you have tools that will let you do so.

Taking Actual Action

It's likely that you'll be mocking fewer commands in a validation test, since much the point of it is to let stuff actually happen. But that doesn't mean you won't need to set up certain pre-conditions. That might include test data files, or specify certain environmental configurations. You might "inject" those into the environment at the start of your `Describe` block, or you might have them "baked into" the environment in the form of virtual machine images or something similar. Whatever the case, the key thing here is to understand that there's a bit more "setup" involved, because you're no longer simply focused on your code and only your code.

Timing your tests can be tricky, too, which again is where orchestration tools can come into play. For example, if you're authoring Desired State Configuration (DSC) resources, you may need to spin up a test virtual machine, inject a DSC configuration that uses your resource, *let DSC stew on all that for half an hour or whatever*, and *then* run your Pester tests. Those are all tasks you'll have to plan out, and the highlight how much more complex validation testing can be.

Testing the Outcomes of Your Actions

Your `It` blocks and `Should` commands remain the foundation of your tests. For example, suppose you need to ensure that a given test virtual machine has build 2004. You might:

```
It "Has Windows build 2004" {
  $p = @{
    ComputerName='TESTMACHINE'
    Class='Win32_OperatingSystem'
  }
  Get-CimInstance @p |
  Select-Object -Expandproperty BuildNumber |
  Should Be 2004
}
```

This would throw an exception, failing the test, if the correct result didn't come back. And as always, you don't *need* to use `Should`; you can use any kind of code you want, and simply `throw` an exception if your criteria aren't met.

This is a very different approach to testing, but it's one Pester is well-suited for. Obviously, there's a lot more "lifting" on you, in terms of setting up test environments, deciding what to test, and coding up the tests themselves, but if this is what you need to do, then you now have an idea on how to go about it.

Measuring Code Coverage

Code coverage is the idea of making sure that your unit tests are "exercising" all of your code. For example, consider this snippet:

```
If ($condition) {
    Get-CimInstance -Class Win32_Service
} else {
    Get-WmiObject -Class Win32_Service
}
```

You'd want to make sure that a unit test of this code ran both possible conditions. You'd likely mock both `Get-CimInstance` and `Get-WmiObject`, since they're not your code, but you'd want to ensure that both "code paths" executed under the correct conditions.

Pester can help you measure your tests' code coverage, so that you can better estimate if every "code path" has run. However, Pester's code coverage tools, like similar tools in any unit testing framework, can only do so much. Specifically, they can simply look at the number of lines of code you've written, and tell you how many of those lines have actually executed. What they can't do is make sure you're testing all the different conditions you should be. In the above snippet, for example, you might think to write one test where `$condition` is `$True`, and another where it's `$False`. If you ran both of those tests, Pester would indicate that you'd hit 100% code coverage for that snippet. But Pester couldn't remind you to test your code where `$condition` was equal to `"Purple"` or some other unexpected value. In other words, Pester can't tell you if you've *run a complete test of all logical possibilities;* merely if every line of code has executed. So code coverage is a *tool*, but it's not the only tool you should use. The best tool is your own brain, and your understanding of *your* code.

Displaying Code Coverage Metrics

Unlike Pester itself, which will run on PowerShell v2 and later, code coverage metrics require PowerShell v3 or later. We're trusting this is no longer an issue for you bue we don't want to make any assumptions.

To generate code coverage statistics, simply add the `-CodeCoverage` parameter when you run `Invoke-Pester` to execute your tests. The parameter accepts strings, which should be the file paths (and can include wildcards) of the scripts you want to generate coverage for. You can also get more granular by passing a hash table to the parameter, like this:

```
@{ Path = 'c:\path\to\script*'
   Function = 'Get-*'
   StartLine = 120
   EndLine = 150 }
```

Only `Path` is required (and you can use `p` instead). If you specify `Function` (or `f`), you can provide the name of a function (or wildcards) that you want to generate coverage metrics for. Alternately, you can provide `StartLine` and `EndLine` (or `s` and `e`); these will be ignored if you used `Function` or `f`, but otherwise indicate the lines of code you want coverage metrics for. If you include `StartLine` and omit `EndLine`, it'll just run through the end of the specified file(s).

Even though we're trying to provide an introduction to code coverage, this is far from exhaustive coverage. Be sure to read full help for `Invoke-Pester` looking at the code coverage related parameters.

An Example

Let's look at a relatively simple example. The script file and Pester test are included in the chapter downloads. Say you have a function like this that resides in the file FunctionToTest.ps1

Get-MyServer

```
function Get-MyServer {
    [cmdletbinding()]
    Param(
        [Parameter(Mandatory, ValueFromPipeline)]
        [string]$Computername,
        [switch]$ResolveIP,
        [switch]$UseDcom,
        [pscredential]$Credential
    )

    Begin {
        Write-Verbose "Starting $($myinvocation.MyCommand)"
        $params = @{
            SkipTestConnection = $True
        }
    }
    Process {
        if ($UseDcom) {
            Write-Verbose "Connecting with DCOM"
            $opt = New-CimSessionOption -Protocol Dcom
            $params.Add("SessionOption", $opt)
```

```
        }

        if ($Credential) {
            Write-Verbose "Using alternate credential"
            $params.Credential = $Credential
        }

        if ($ResolveIP) {
            Write-Verbose "Resolving IP4 address"
            $resolve = @{
             Name = $Computername
             Type = "A"
             TcpOnly = $True
             ErrorAction = "SilentlyContinue"
            }
            $IP = (Resolve-DnsName @resolve).ip4Address
        }
        else {
            $IP = "0.0.0.0"
        }
        $cs = New-Cimsession @params
        $compsys = $cs | Get-CimInstance -classname win32_computersystem
        $os = $cs | Get-CimInstance -ClassName win32_operatingsystem
        $proc = $cs |
        Get-CimInstance -ClassName win32_processor |
        Select-Object -Property Name -first 1

        [pscustomobject]@{
            Computername = $compsys.Name
            IP           = $IP
            TotalMemGB   = $compsys.TotalPhysicalMemory / 1GB -as [int]
            Model        = $compsys.model
            OS           = $os.Caption
            Build        = $os.BuildNumber
            Processor    = $proc.Name
        }

        Remove-CimSession $cs
    }
    End {
        Write-Verbose "Ending $($myinvocation.MyCommand)"
    }
}
```

In the same directory we started writing a Pester test for the function.

Get-MyServer Pester Test

```
$here = Split-Path -Parent $MyInvocation.MyCommand.Path
$sut = (Split-Path -Leaf $MyInvocation.MyCommand.Path) -replace '\.Tests\.', '.'
. "$here\$sut"

Describe "Get-MyServer" {
    Mock Get-CimInstance {
        New-CimInstance -ClientOnly -ClassName Win32_ComputerSystem -Property @{
            Name                = "SERVER1"
            TotalPhysicalMemory = 32GB
            Model               = "BestServerEver"
        }
    } -ParameterFilter {$classname -eq "win32_computersystem"} -Verifiable

    Mock Get-CimInstance {
      New-CimInstance -ClientOnly -ClassName Win32_OperatingSystem -Property @{
          Caption     = "Windows Server"
          BuildNumber = "1234"
      }
    } -ParameterFilter {$classname -eq "win32_operatingsystem"} -Verifiable

    Mock Get-CimInstance {
      New-CimInstance -ClientOnly -ClassName Win32_Processor -Property @{
          Name = "Flux Capacitor 2K"
      }
    } -ParameterFilter {$classname -eq "win32_processor"} -Verifiable

    Mock Resolve-DNSName {
        @{
            Name       = "SERVER1"
            IP4Address = "10.10.10.10"
            Type       = "A"
        }
    }

    $r = Get-MyServer -Computername SERVER1

    It "should run Get-CimInstance" {
        Assert-VerifiableMock
```

```
    }

    It "should run Get-CimInstance 3 times" {
        Assert-MockCalled Get-Ciminstance -Times 3
    }

    It "The result should have a Computername property of SERVER1" {
        $r.Computername | Should be "SERVER1"
    }

    It "The result should have a Build  property of 1234" {
        $r.build | Should be "1234"
    }
}
```

As written, the function passes all the tests. But is it complete? That's why we check for code coverage.

```
invoke-pester -CodeCoverage @{Path=".\FunctionToTest.ps1";
Function="Get-MyServer"}
```

The tests run but we also get a listing at the end.

Pester code coverage

It is difficult to read some of the output, but the code coverage report says that our tests "Covered 77.14% of 35 analyzed Commands in 1 File". And then you can see the commands that were not tested. Some of the commands, like `Write-Verbose` we probably don't need to test. But some of the others we might. We also want to re-iterate that this is showing up what command execution paths we didn't test for. Code coverage doesn't mean, "What commands am I not testing." Our function uses commands like `Remove-CimSession` which isn't included anywhere in our Pester test. Code coverage didn't detect it because we *are* testing the path where that command gets called.

The idea of Pester code coverage is to just get *one* indicator of whether or not *all of your code ran;* Pester isn't making any commentary on whether all of your code received all of the relevant input variation that might be appropriate.

Test-Driven Development

Test-Driven Development, or TDD, is a *big* philosophy. *Test Driven Development by Example*, by Ken Beck, is one of our favorite texts on the subject, should you want to dive deeper… because this chapter ain't going to dive too deep.

TDD, stripped away of every possible meaningful detail, is simply the practice of writing your unit tests before writing your code. Your unit tests serve, in a way, as a kind of unit-level functional specification for your code. If someone else wrote your code, they'd simply have to make sure all the tests passed, and you'd all agree that the code was good.

This *doesn't* mean sitting down and writing every possible test that you'll ever need to write. The practical reality is that, except for the smallest imaginable chunks of code, TDD is part of an iterative process.

Imagine, for example, that you usually start writing functions by defining your parameters. That's a pretty common approach. In TDD, you'd start by writing tests that verify and validate those parameters, before you write a lick of actual PowerShell code. You'd write tests that ensure pipeline input worked when it was supposed to, parameters accepted the data types they were supposed to, and so on. Once the tests were ready, you'd start writing your code, even though so far as you're "coding" is the `Param` block. You'd keep messing with the `Param` block until you could run the tests and pass every single one.

Then you'd move on to the next bit of your script. Perhaps, for example, you have a switch parameter that tells your function to behave in one way or another. You'd write tests that tested the different behaviors, and *then* write the logic itself, and then run the tests. You'd keep revising the code until the tests passed.

The idea with TDD is to force you to think about what your code should conform to *first*, rather than just diving in and coding by the seat of your pants. Writing tests first implies a certain design stage, where you *think about things* before you start typing. TDD also kind of forces you to get tests in place, which even though you know you *should* do, you won't always *want* to do.

Let's say you finish your function (and its tests!). Later, you discover a bug. Before you fix the bug, you'd write the test(s) necessary to see if the bug exists or not. Initially, that test will fail. But *then* you go and fix your code so that the test passes. With TDD, it's always *write the test first*, and then get the code to comply with the test.

It's a big commitment. And trust us, not every professional software development team even does this. But, those that do have had quite a bit of success with it, and end up writing a fuller and more reliable test suite, as well as having a better shared goal about what the eventual code will need to do.

We're not saying you *have* to use TDD, but we'd recommend considering it on your next project.

Part 7: PowerShell 7 Scripting

PowerShell 7, which is essentially today's PowerShell and runs cross platform, offers a few new features that you can use in your scripting projects. Of course, these will only work when the code is run on a PowerShell 7 platform. The chapters in this section will introduce you to some new operators, variables, and parameters.

PowerShell 7 Scripting Features

The world of PowerShell is slowly shifting. It began a number of years ago when Microsoft make PowerShell an open source project. At the time, this new version of PowerShell would run on Windows, Linux and Mac. These were the days of PowerShell Core or 6.x. Over time, Microsoft slowly started adding back features that people really wanted out of PowerShell. Eventually, we ended up with PowerShell 7.

Microsoft made a major version increment primarily as a way to make a statement. Going forward, PowerShell 7.x *is* PowerShell. Windows PowerShell 5.1 which ships with Windows 10 isn't going anywhere. But it is also finished. Barring critical security bugs, Microsoft is investing in developing PowerShell 7. Notice they even dropped "Windows" from the name. Eventually, you should be able to manage anything from anywhere using whatever client you want. Even today, you can manage Windows servers running Windows PowerShell 5.1 from a Windows 10 desktop running PowerShell 7. However from a scripting perspective PowerShell 7 offers up new features you'll find useful but new challenges as well. For now, let's explore some of the scripting goodies PowerShell 7 brings to the party. Don't forget that if you use any of these features to add `#requires -version 7` at the top of your script file.

If you are interested in what's new in PowerShell 7 and why you might want to make the jump, grab a free (there is an option to make a charitable donation) copy of the #PS7Now[89].

Updating Your Editor

If you are running VS Code and have the PowerShell extension installed, you should be set. It will detect and use PowerShell 7 as part of its integrated terminal. If you need to switch to Windows PowerShell, you can do that as well. In the terminal dropdown, click `Select Default Shell` and pick one.

Even though the recommendation is to use VS Code, there are plenty of legitimate reasons you may have to stick with the PowerShell ISE. This app will never be updated to use PowerShell 7. But there is an easy hack you can implement. Assuming you installed PowerShell 7 and enabled PowerShell remoting, launch the PowerShell ISE *with elevated credentials*. In the console panel, run this code:

```
Enter-PSSession -computername $env:COMPUTERNAME -Configuration PowerShell.7
```

This hack will open a remoting session to yourself using the PowerShell 7 endpoint. Now, the scripting panel will be PowerShell 7 "aware".

[89] https://leanpub.com/ps7now

Ternary Operators

One of the most anticipated PowerShell 7 features is the addition of Ternary Operators. The structure is something like this:

```
<if some condition is true> ? <do this> : <else do this>
```

In Windows PowerShell, you would have written code like this:

```
if ($IsWindows) {
    "ok"
}
else {
    "not ok"
}
```

Sure. You could squeeze it down into a single line.

```
if ($IsWindows) {"ok"} else {"not ok"}
```

But with the ternary operator you can write it like this:

```
$IsWindows ? "ok":"not ok"
```

The first part of the expression is some bit of code that evaluates as True or False. Whatever comes after the `?` runs when true and code after the `:` runs when false. There's no performance benefit as far as we can tell. And frankly, it is a bit cryptic at first glance. Especially when compared to an `If/Else` construct.

As an example, here's a traditional structure.

```
if ($IsWindows) {
    Get-CimInstance -ClassName win32_service -filter "name='bits'"
    Get-CimInstance -ClassName win32_service -filter "name='wsearch'"
}
else {
    Clear-Host
    Get-Date
    Write-Warning "This command requires Windows"
}
```

And here's the ternary equivalent:

```
$IsWindows ? (Get-CimInstance -ClassName win32_service -filter "name='bits'"),
(Get-CimInstance -ClassName win32_service -filter "name='wsearch'") :
(Clear-Host),(Get-Date),(Write-Warning "This command requires Windows")
```

This could be written as a one-line expression. We're accommodating page width restrictions.

As an alternative, in some situations, code like this might make sense:

```
$win = {
 Get-CimInstance -ClassName win32_service -filter "name='bits'"
 Get-CimInstance -ClassName win32_service -filter "name='wsearch'"
}
$nowin = {
 Clear-Host
 Get-Date
 Write-Warning "This command requires Windows"
}

$IsWindows ? (&$win) : (&$nowin)
```

For simple expressions, using the ternary operator might save some typing and create a succinct expression.

```
 $var = (get-date).DayOfWeek -eq "Friday" ? "tgif": "blah"
```

Chain operators

Many shells have conditional operators that allow you to string commands to together. The logic typically goes "run this command and if it is successful then run this next command". In PowerShell, you can now use the `&&` operator.

```
1 && 2
```

If the command on the left side of `&&` is successful, then run the command on the right side. The operator decides based the of the value of the built-in variable `$?`. Here's a more practical example.

```
PS C:\> test-wsman thinkp1 && Get-CimInstance win32_bios -ComputerName Thinkp1

wsmid           : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProtocolVersion : http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd
ProductVendor   : Microsoft Corporation
ProductVersion  : OS: 0.0.0 SP: 0.0 Stack: 3.0


SMBIOSBIOSVersion : N2EET46W (1.28 )
Manufacturer      : LENOVO
Name              : N2EET46W (1.28 )
SerialNumber      : R90SH9GR
Version           : LENOVO - 1280
PSComputerName    : Thinkp1
```

Because the `Test-WSMan` command succeeded, the `Get-CimInstance` command is run. You could have written this with an If construct, and perhaps in a script that would make more sense.

But be careful. Even if the first expression fails, that doesn't mean the second expression won't run.

```
[void](test-wsman foo) && Get-CimInstance win32_bios -ComputerName foo
```

The `Test-WSMan` command will fail, but that doesn't mean it didn't run. The `$?` variable will be True so the second command is run. You may need to force a terminating exception.

```
[void](test-wsman foo -erroraction stop) && Get-CimInstance win32_bios -ComputerName\
 foo
```

You may also want to execute a statement if the first command does not complete. That's where `||` comes into play.

```
1/0 || Write-Warning "What are you trying to do?"
```

Try this. Then try a variation that doesn't fail.

```
1/1 || Write-Warning "What are you trying to do?"
```

That's the easiest way to see this in action.

You can also combine operators into a kind of If/Then/Else equivalent.

```
Get-Service foo && Write-Host "service found" || Write-Host "service failed" -Foregr\
oundColor red
```

But be careful. You might think this would work.

```
$svc = Get-Service bits && Write-Host "service $($svc.name) found" || Write-Host "se\
rvice failed" -ForegroundColor red
```

But you'll get nothing for the service name. The solution is to use the common `OutVariable` parameter.

```
$svc = Get-Service bits -ov s && Write-Host "service $($s.name) found" || Write-Host\
 "service failed" -ForegroundColor red
```

These operators will be cryptic to people just getting started with PowerShell so if you use them in your code, it wouldn't hurt to throw in a comment or two.

Take a few minutes to review the help topic `about_Pipeline_Chain_Operators`.

Null-Coalescing Assignment

Is your head screwed on tight? This next one will have your head spinning, at least at first. PowerShell 7 has a new operator, `??` which is referred to a a *null-coalescing* operator. Huh? This operator goes between 2 statements.

```
<left-side> ?? <right-side>
```

If the left side is **null**, then PowerShell will give you the right-side value. If the left-side is not null, you get the left side.

```
$foo = $null
$foo ?? "bar"
```

If you try this in PowerShell 7, you'll get "bar" because $foo is Null. But be careful. If define `$foo=""` that is NOT null and you'll get an empty line.

```
$foo = $PSVersionTable.PSVersion.ToString()
$foo ?? "bar"
```

Running this will display your PowerShell version because $foo isn't Null.

How might you use this in a script? Let's say a variable called `SomeVariable` is getting processed. If it has a value, it will be for a computer name. But if it is Null, you want to use the local computername. In PowerShell 7 you would write it like this:

```
$c = $SomeVariable
$computername = $c ?? ([system.environment]::MachineName)
```

Yes, you could also simply write an If statement but some people like this succinct approach.

Related to this is the Null-Coalescing assignment operator, `??=`.

```
<left-side> ??= <right-side>
```

In this situation, the value from the right side is applied to left side, if the left side is null. If the left side is not Null the right side is ignored. Using the previous scenario, we could assign a value to `$Computername` if it is Null, using the local computername.

```
$computername ??= ([system.environment]::MachineName)
```

Null Conditional Operators

If your head wasn't spinning before, this'll do it. The Null Conditional operators `?` and `?[]` are used to allow conditional access to an object's member such as a method or property as long as it isn't null. Let's look at what happens with a non-null value.

```
$p = Get-Process -id $pid
${p}?.startTime
```

The result should be the start time of the current PowerShell process. Note that because PowerShell allows the use of `?` in variable names, (don't ask us why you would ever want to do that), you need to reference the variable using that funky ${} syntax. If `$p` was Null, then PowerShell wouldn't give you anything. In fact, it won't even try to resolve the member so you won't get any errors.

You can do something similar with array elements.

```
$n = 2,4,6,8,10
${n}?[2]
```

Because $n is not null, You'll get the array element at index 2 or 6. If $n was Null, PowerShell doesn't even try to get the array element. Again, there's no error message.

The new Null-related operators are documented in about_Operators.

The Null Conditional Operators are a PowerShell 7 experimental feature. If you encounter problems trying to use them, run Get-ExperimentalFeature and see if the PSNullConditionalOperators feature is enabled. If not, you can run Enable-ExperimentalFeature PSNullConditionalOperators and restart your PowerShell 7 session.

ForEach-Object Parallel

One feature that got a lot of people excited for PowerShell 7 was the -Parallel parameter in ForEach-Object. The basic premise is that PowerShell will run your scriptblock in parallel runspaces. The number of parallel operations is controlled by the ThrottleLimit parameter which has a default of 5. The concept looks like this:

```
<input objects> | ForEach-Object -parallel {<do something with $_>}
```

Looks simple enough. But just because you *can* do this doesn't mean you should. Here's a simple demonstration.

```
Measure-Command {1..1000 | ForEach-Object {$_*10}}
```

This takes about 102 milliseconds.

```
Measure-Command {1..1000 | ForEach-Object -parallel {$_*10}}
```

This took 35 seconds! There is overhead in setting up and tearing down the parallel runspaces so you need to be smart about how you use it. And sometimes you simply have to test.

Here's an example where using -Parallel can help. In Windows PowerShell, you would use code like this to process a list of locations.

```
$locations| ForEach-Object {
  $p = $_
 Write-Host "[$(Get-Date -f 'hh:mm:ss.ffff')] Measuring $p" -Fore green
 Get-ChildItem -Path $p -file -Recurse |
 Measure-Object -Property length -sum -Average |
 Select-Object @{Name="Path";Expression = {$p}},Count,
 @{Name="SumKB";Expression={$_.sum/1KB -as [int]}},
 @{Name="AvgKB";Expression={$_.average/1KB -as [int]}}
}
}
```

In our test with 6 locations, including one on a NAS device this took about 2 minutes to complete. Here's the parallel equivalent:

```
$locations | ForEach-Object -parallel {
 $p = $_
 Write-Host "[$(Get-Date -f 'hh:mm:ss.ffff')] Measuring $p" -ForegroundColor green
 Get-ChildItem -Path $p -file -Recurse |
 Measure-Object -Property length -sum -Average |
 Select-Object @{Name="Path";Expression = {$p}},Count,
 @{Name="SumKB";Expression={$_.sum/1KB -as [int]}},
 @{Name="AvgKB";Expression={$_.average/1KB -as [int]}}
}
```

Which took almost 3 minutes. Which obviously isn't much of an improvement. However, there are external factors that can come into play with this particular example such as disk contention and network latency, since we're querying a NAS device. The key takeaway with this feature is this it just depends. You have to take more than your code into consideration to determine if using `-Parallel` is appropriate.

Using ANSI

Even though we probably had support in previous editions, one cool feature in PowerShell 7 is more widespread use of ANSI escape sequences. These are kind of an old-school approach to working with text in a console or terminal. With PowerShell 7, you can do things like color a piece of text without resorting to `Write-Host`.

In short, you wrap your text in an ANSI sequence.

```
"`e[36mHello World`e[0m"
```

This will write "Hello World" to the PowerShell pipeline, but it will be in color!

In PowerShell 7 use `` `e `` for the escape sequence. In Windows PowerShell, you can use `$([char]0x1b)`. You could write the Hello World example as `"$([char]0x1b)[36mHello World$([char]0x1b)[0m"` and it would work in both Windows PowerShell and PowerShell 7. But not in the PowerShell ISE which doesn't know what to do with ANSI characters.

Here's a snippet of a function that takes advantage of this feature. The full function is is in the code downloads.

Get-Status

```
            foreach ($item in $free) {
                $sName = $item.name -replace "Pct", "%"
                if ($IsWindows -AND $IsCoreCLR) {
                    #Colorize values
                    if ([double]$item.value -le 20) {
                        #red
                        $value = "`e[91m$($item.value)`e[0m"
                    }
                    elseif ([double]$item.value -le 50) {
                        #yellow
                        $value = "`e[93m$($item.value)`e[0m"
                    }
                    else {
                        #green
                        $value = "`e[92m$($item.value)`e[0m"
                    }
                }
                else {
                    $value = $item.Value
                }

                $string += " {0}:{1}" -f $sname, $value

            } #foreach item in free

            $string
```

If the user is running PowerShell 7 and uses the functions `-AsString` parameter, they'll get color-coded output.

```
PS C:\> get-status -AsString
BOVINE320 Up:7.20:29:32 %FreeMem:57.96 %FreeC:34.68 %FreeD:42.05
PS C:\>
```

Get-Status -AsString

And yes, the function could be revised to use the `$([char]0x1b)` to work in Windows PowerShell and PowerShell 7.

There is a terrific write-up on this feature in the free ebook #PS7Now[90]. You might also checkout a script[91] Jeff wrote to show ANSI samples in your console. And he has a number of ANSI related features in the PSScriptTools[92] module.

[90] https://leanpub.com/ps7now
[91] https://jdhitsolutions.com/blog/powershell/7502/show-ansi-samples/
[92] https://github.com/jdhitsolutions/PSScriptTools

Cross Platform Scripting

When Microsoft moved PowerShell to the open source world a few years ago, they opened up an entire new world of scripting possibilities. Microsoft wants you to be able to manage anything from anywhere on whatever platform you need. Of course, you'll want to build scripts and tools to use in this new world. And even if you aren't in a situation now thats requires cross-platform management, you never know what's gonna happen tomorrow. And frankly, some of the things you need to keep in mind when it comes to cross-platform scripting can make you a better scripter for Windows PowerShell.

What do we mean when we say "cross-platform"? We mean that you can develop a tool that can be used on any PowerShell-supported platform and ideally, can manage or work with any platform. Instead of writing one tool to run on Linux and another to run on Windows, you write one tool that can run on both. Don't build multiple tools to manage remote servers depending on the target operating system. Write one command that you can use regardless of the OS.

In order to meet these requirements, there are a few things we want you to keep in mind. We are under no illusion that everything you want to do cross-platform will work. It won't. There *will* be situations where a true cross-platform solution simply doesn't work.

Many of the things we're covering in this chapter, technically also apply to PowerShell Core which was the PowerShell 6.x branch.

Know Your OS

When you are building something that can potentially be used cross-platform you need to think about the operating system where your code will be running. As cool as PowerShell 7 is, not every feature or command is supported on every operating system. You don't need to be an expert level Linux engineer, but you do need to understand some basics. For example, Linux doesn't have the same concept of services as we understand them in the Windows world, so there is no `Get-Service` command in PowerShell 7 on non-Windows systems. Likewise, Linux doesn't have WMI or CIM so you won't find `Get-CimInstance`. Nor could you use `Get-CimInstance` to target a remote Linux machine. The OS doesn't support these features so they aren't available.

What this means is that you need to know what OS your code is running on and there are some things you can do to make your life easier.

State Your Requirements

First off, if you are writing a PowerShell function that uses PowerShell 7 features, such as the new ternary operator, you need to make sure the person running your script is using PowerShell 7. This means adding a #requires statement at the top of your file

```
#requires -version 7.0
```

Honestly, this is something you should have been doing all along but you absolutely need it now. If you need specific modules that might be Windows specific, state that requirement as well.

```
#requires -modules CIMCmdlets
```

If someone runs this on a Linux box they'll get an error like:

```
The script 'Get-Miracle.ps1' cannot be run because the following modules that are sp\
ecified by the "#requires" statements of the script are missing: CimCmdlets.
```

If you are building a module, in the manifest you can use this setting to set requirements.

```
# Supported PSEditions
CompatiblePSEditions = @('Desktop','Core')
```

`Desktop` means Windows PowerShell. `Core` means the open-source and cross-platform version of PowerShell, *regardless of operating system*. This can be a little tricky. Because even though `Get-CimInstance` doesn't work on a Mac in PowerShell 7, it will work just fine on a Windows desktop running PowerShell 7. Still, this high level compatibility setting should be set. Delete whatever isn't supported.

Testing Variables

Even though the `Core` setting is potentially problematic, there are a number of new automatic variables you can use in your code, to validate and test.

- `PSEdition` - On PowerShell 7, this will have a value of `Core`. On Windows PowerShell it will return `Desktop`. Look familiar?
- `IsWindows` - A boolean value indicating if you are running Windows. Requires PowerShell 7.
- `IsLinux` - A boolean value indicating if you are running Linux. Requires PowerShell 7.
- `IsMac` - A boolean value indicating if you are running MacOS. Requires PowerShell 7.
- `IsCoreCLR` - A boolean value indicating if you are running .NET Core which most likely means you are running PowerShell 7. This variable isn't defined in Windows PowerShell.

And of course, don't forget `$PSVersionTable`.

```
PS C:\> $PSVersionTable

Name                           Value
----                           -----
PSVersion                      7.0.2
PSEdition                      Core
GitCommitId                    7.0.2
OS                             Microsoft Windows 10.0.19041
Platform                       Win32NT
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0â€¦}
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
WSManStackVersion              3.0
```

Which can vary by platform.

```
PS /home/jeff> $PSVersionTable

Name                           Value
----                           -----
PSVersion                      7.0.2
PSEdition                      Core
GitCommitId                    7.0.2
OS                             Linux 4.19.104-microsoft-standard #1 SMP Wed Feb 19 0\
6:37:35 UTC 2020
Platform                       Unix
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0â€¦}
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
WSManStackVersion              3.0
```

It is also different on Windows PowerShell.

```
PS C:\> $PSVersionTable

Name                           Value
----                           -----
PSVersion                      5.1.19041.1
PSEdition                      Desktop
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0...}
BuildVersion                   10.0.19041.1
CLRVersion                     4.0.30319.42000
```

```
WSManStackVersion               3.0
PSRemotingProtocolVersion       2.3
SerializationVersion            1.1.0.1
```

These are all potentially useful items you can use to build `If` constructs or even dynamic parameters.

Environment Variables

One of the major challenges in cross-platform scripting is breaking out of the old way of doing things. Here's a great example using a common parameter definition you might see in Windows PowerShell.

```
[string[]]$Computername = $env:ComputerName
```

This sets the %COMPUTERNAME% environment variable as the default value for `$Computername`. This will work just fine on Windows PowerShell and even PowerShell 7 running on Windows. But non-Windows platforms have different environment variables and this will fail. Instead, you can use a .NET code snippet to get the same result.

```
[environment]::MachineName
```

This also true with `$env:username` which can be replaced with `[environment]::UserName`. If you are used to referencing environment variables, you'll need to add a step to verify they are defined or begin using .NET alternatives.

Paths

PowerShell has always never cared about the direction of slashes in paths. You can run `Test-Path c:\windows` or `Test-Path c:/windows`. This is even true on non-Windows systems. You can use `test-path /etc/ssh` or `test-path \etc\ssh`. But you really should be careful. Many of you probably have used or seen code like this:

```
$file = "$foo\child\file.dat"
```

There's a good chance it will work cross-platform. But the better approach, which you should be using anyway, is to use the path cmdlets like `Join-Path`. Here's a sample.

Export-Data

```
Function Export-Data {
    [cmdletbinding()]
    Param(
    [ValidateScript({Test-Path $_})]
    [string]$Path = "."
    )

    $time = Get-Date -format FileDate
    $file = "$($time)_export.json"
    $ExportPath = Join-Path -Path (Convert-Path $Path) -ChildPath $file
    Write-Verbose "Exporting data to $exportPath"

    # code ...
}
```

Instead of worrying about which direction to put the slashes, let PowerShell do it for you.

If you need it, you can use `[System.IO.Path]::DirectorySeparatorChar` to get the directory separator character. For the %PATH% variable you can use `[System.IO.Path]::PathSeparator`. Let's say you want to split the %PATH% environment variable into an array of locations. A simple cross-platform approach would be `$env:PATH -split [System.IO.Path]::PathSeparator`. Don't forget that Linux is case-sensitive.

Watch Your Aliases

You've most likely heard us go on and on about the downside of using aliases in your PowerShell scripts. Use them all you want interactively at a prompt but in written code, use full cmdlet names. Now you really need to.

In the old days, we'd happily write a command like this:

```
get-process | sort handles -descending
```

And it would work just fine on Windows, even under PowerShell 7. But not Linux.

```
PS /home/jeff> get-process | sort handles -descending
/usr/bin/sort: invalid option -- 'e'
Try '/usr/bin/sort --help' for more information.
```

What happened? In PowerShell 7 on Linux, any alias that could resolve to a native command has been removed. So there is no `sort` alias for `Sort-Object`. PowerShell thinks you want to run the native `sort` command. If you've been in the habit of using Linux aliases like `ps` or `ls`, you'll need to get over it. You need to start writing expressions like:

```
get-process | sort-object handles -descending
```

Now there's no mistaking what you want to do and it will run everywhere.

If you are using VS Code, and we don't know why you're not, it is very easy to convert aliases. We get it. You have muscle memory and you find it easier to write code using aliases. Fine. When you are finished and before you release you code into the world, open the command palette (<kbd>Ctrl</kbd>+<kbd>Shift</kbd>+<kbd>P</kbd>) and run 'Powershell: Expand alias". Done.

Leverage Remoting

One of the most anticipated features of PowerShell 7 is the use of `ssh` for remoting. Non-Windows systems won't support the WSMan protocol which means no remoting the way you used to do it. But you may still want to use remoting in your toolmaking. In fact, leveraging remoting is a smart idea. Going to PowerShell 7 just means a little more work on your part.

We're not going to dive into the mechanics of getting SSH remoting to work in PowerShell 7. That's all up to you and your organization. We're simply going to assume it already works.

One relatively easy approach you can use is parameter sets. Define one parameter set for a computer name and another for PSSession objects. PowerShell already follows this model. You can too. Here's a proof of concept function.

Get-RemoteData

```
Function Get-RemoteData {
    [cmdletbinding(DefaultParameterSetName = "computer")]
    Param(
        [Parameter(
            Position = 0,
            Mandatory,
            ValueFromPipeline,
            ParameterSetName = "computer"
        )]
        [Alias("cn")]
        [string[]]$Computername,
        [Parameter(ParameterSetName = "computer")]
        [alias("runas")]
        [pscredential]$Credential,
        [Parameter(ValueFromPipeline,ParameterSetName = "session")]
```

```
        [System.Management.Automation.Runspaces.PSSession]$Session
    )
    Begin {
      $sb = {"Getting remote data from $([environment]::MachineName) [$PSEdition]"}
      $PSBoundParameters.Add("Scriptblock",$sb)
    }
    Process {
      Invoke-Command @PSBoundParameters
    }
    End {}
}
```

The person running the function can either pass a computername with an optional credential, or a previously created PSSession object. They may have existing connections to a mix of platforms, some using SSH connections. Now they can run one command that works for all.

```
PS C:\> get-pssession | get-remotedata
Getting remote data from SRV2 [Desktop]
Getting remote data from FRED [Core]
Getting remote data from SRV1 [Desktop]
```

In this example, FRED is a Linux server running Fedora. Start simple like this.

But when you are ready, you can get very creative. Here's a function that defines dynamic parameters if the user is running PowerShell 7.

Stop-RemoteProcess

```
#requires -version 5.1

Function Stop-RemoteProcess {
[cmdletbinding(DefaultParameterSetName = "computer")]
Param(
    [Parameter(
        ParameterSetName = "computer",
        Mandatory,
        Position = 0,
        ValueFromPipeline,
        ValueFromPipelineByPropertyName,
        HelpMessage = "Enter the name of a computer to query."
    )]
    [ValidateNotNullOrEmpty()]
    [Alias("cn")]
    [string[]]$ComputerName,
```

```
    [Parameter(
        ParameterSetName = "computer",
        HelpMessage = "Enter a credential object or username."
    )]
    [Alias("RunAs")]
    [PSCredential]$Credential,
    [Parameter(ParameterSetName = "computer")]
    [switch]$UseSSL,

    [Parameter(
        ParameterSetName = "session",
        ValueFromPipeline
        )]
    [ValidateNotNullOrEmpty()]
    [System.Management.Automation.Runspaces.PSSession[]]$Session,

    [ValidateScript( {$_ -ge 0})]
    [int32]$ThrottleLimit = 32,

    [Parameter(Mandatory,HelpMessage = "Specify the process to stop.")]
    [ValidateNotNullOrEmpty()]
    [string]$ProcessName,

    [Parameter(HelpMessage = "Write the stopped process to the pipeline")]
    [switch]$Passthru,

    [Parameter(HelpMessage = "Run the remote command with -WhatIf")]
    [switch]$WhatIfRemote
)
DynamicParam {
    #Add an SSH dynamic parameter if in PowerShell 7
    if ($isCoreCLR) {

        $paramDictionary = New-Object -Type System.Management.Automation.RuntimeDefi\
nedParameterDictionary

        #a CSV file with dynamic parameters to create
        #this approach doesn't take any type of parameter validation into account
        $data = @"
Name,Type,Mandatory,Default,Help
HostName,string[],1,,"Enter the remote host name."
UserName,string,0,,"Enter the remote user name."
Subsystem,string,0,"powershell","The name of the ssh subsystem. The default is power\
```

```
shell."
Port,int32,0,,"Enter an alternate SSH port"
KeyFilePath,string,0,,"Specify a key file path used by SSH to authenticate the user"
SSHTransport,switch,0,,"Use SSH to connect."
"@

        $data | ConvertFrom-Csv | ForEach-Object -begin { } -process {
            $attributes = New-Object System.Management.Automation.ParameterAttribute
            $attributes.Mandatory = ([int]$_.mandatory) -as [bool]
            $attributes.HelpMessage = $_.Help
            $attributes.ParameterSetName = "SSH"
            $attributeCollection = New-Object -Type System.Collections.ObjectModel.C\
ollection[System.Attribute]
            $attributeCollection.Add($attributes)
            $dynParam = New-Object -Type System.Management.Automation.RuntimeDefined\
Parameter($_.name, $($_.type -as [type]), $attributeCollection)
            $dynParam.Value = $_.Default
            $paramDictionary.Add($_.name, $dynParam)
        } -end {
            return $paramDictionary
        }
    }
} #dynamic param

Begin {
    $start = Get-Date
    #the first verbose message uses a pseudo timespan to reflect the idea we're just\
 starting
    Write-Verbose "[00:00:00.0000000 BEGIN  ] Starting $($myinvocation.mycommand)"

    #a script block to be run remotely
    Write-Verbose "[$(New-TimeSpan -start $start) BEGIN  ] Defining the scriptblock \
to be run remotely"

    $sb = {
    param([string]$ProcessName,[bool]$Passthru,[string]$VerbPref = "SilentlyContinue\
", [bool]$WhatPref)

    $VerbosePreference = $VerbPref
    $WhatIfPreference = $WhatPref

    Try {
        Write-Verbose "[$(New-TimeSpan -start $using:start) REMOTE ] Getting Process\
```

```
 $ProcessName on $([System.Environment]::MachineName)"
        $procs = Get-Process -Name $ProcessName -ErrorAction stop
        Try {
          Write-Verbose "[$(New-TimeSpan -start $using:start) REMOTE ] Stopping $($p\
rocs.count) Processes on $([System.Environment]::MachineName)"
          $procs | Stop-Process -ErrorAction Stop -PassThru:$Passthru
        }
        Catch {
          Write-Warning "[$(New-TimeSpan -start $using:start) REMOTE ] Failed to sto\
p Process $ProcessName on $([System.Environment]::MachineName). $($_.Exception.messa\
ge)."
        }
    }
    Catch {
        Write-Verbose "[$(New-TimeSpan -start $using:start) REMOTE ] Process $Proces\
sName not found on $([System.Environment]::MachineName)"
    }

} #scriptblock

    #parameters to splat to Invoke-Command
    Write-Verbose "[$(New-TimeSpan -start $start) BEGIN  ] Defining parameters for I\
nvoke-Command"

    #remove my parameters from PSBoundparameters because they can't be used with New\
-PSSession
    $myparams = "ProcessName","WhatIfRemote","passthru"
    foreach ($my in $myparams) {
        if ($PSBoundParameters.ContainsKey($my)) {
            [void]($PSBoundParameters.remove($my))
        }
    }

    $icmParams = @{
      Scriptblock      = $sb
      Argumentlist     = @($ProcessName,$Passthru,$VerbosePreference,$WhatIfRemote)
      HideComputerName = $False
      ThrottleLimit    = $ThrottleLimit
      ErrorAction      = "Stop"
      Session          = $null
    }

    #initialize an array to hold session objects
```

```
    [System.Management.Automation.Runspaces.PSSession[]]$All = @()

    If ($Credential.username) {
        Write-Verbose "[$(New-TimeSpan -start $start) BEGIN  ] Using alternate crede\
ntial for $($credential.username)"
    }

} #begin

Process {
    Write-Verbose "[$(New-TimeSpan -start $start) PROCESS] Detected parameter set $(\
$pscmdlet.ParameterSetName)."

    $remotes = @()
    if ($PSCmdlet.ParameterSetName -match "computer|ssh") {
        if ($pscmdlet.ParameterSetName -eq 'ssh') {
            $remotes += $PSBoundParameters.HostName
            $param = "HostName"
        }
        else {
            $remotes += $PSBoundParameters.ComputerName
            $param = "ComputerName"
        }

        foreach ($remote in $remotes) {
            $PSBoundParameters[$param] = $remote
            $PSBoundParameters["ErrorAction"] = "Stop"
            Try {
                #create a session one at a time to better handle errors
                Write-Verbose "[$(New-TimeSpan -start $start) PROCESS] Creating a te\
mporary PSSession to $remote"
                #save each created session to $tmp so it can be removed at the end
                $all += New-PSSession @PSBoundParameters -OutVariable +tmp
            } #Try
            Catch {
                #TODO: Decide what you want to do when the new session fails
                Write-Warning "Failed to create session to $remote. $($_.Exception.M\
essage)."
                #Write-Error $_
            } #catch
        } #foreach remote
    }
    Else {
```

```
        #only add open sessions
        foreach ($sess in $session) {
            if ($sess.state -eq 'opened') {
                Write-Verbose "[$(New-TimeSpan -start $start) PROCESS] Using session\
 for $($sess.ComputerName.toUpper())"
                $all += $sess
            } #if open
        } #foreach session
    } #else sessions
} #process

End {

    $icmParams["session"] = $all

    Try {
        Write-Verbose "[$(New-TimeSpan -start $start) END    ] Querying $($all.count\
) computers"

        Invoke-Command @icmParams | ForEach-Object {
            #TODO: PROCESS RESULTS FROM EACH REMOTE CONNECTION IF NECESSARY
            $_
        } #foreach result
    } #try
    Catch {
        Write-Error $_
    } #catch

    if ($tmp) {
        Write-Verbose "[$(New-TimeSpan -start $start) END    ] Removing $($tmp.count\
) temporary PSSessions"
        $tmp | Remove-PSSession
    }
    Write-Verbose "[$(New-TimeSpan -start $start) END    ] Ending $($myinvocation.my\
command)"
} #end
} #close function
```

We know the code will wrap here. The script file won't have that problem.

This function essentially follows the same model as a previous example. But this one creates several dynamic parameters if PowerShell 7 is detected. Notice we're using one of the new variables. When the user runs help in PowerShell 7, they'll get the new parameters.

```
NAME
    Stop-RemoteProcess

SYNTAX
    Stop-RemoteProcess [-ComputerName] <string[]> -ProcessName <string> [-Credential <pscredential>] [-UseSSL]
    [-ThrottleLimit <int>] [-Passthru] [-WhatIfRemote] [<CommonParameters>]

    Stop-RemoteProcess -ProcessName <string> [-Session <PSSession[]>] [-ThrottleLimit <int>] [-Passthru] [-WhatIfRemote]
    [<CommonParameters>]

    Stop-RemoteProcess -ProcessName <string> -HostName <string[]> [-ThrottleLimit <int>] [-Passthru] [-WhatIfRemote]
    [-UserName <string>] [-Subsystem <string>] [-Port <int>] [-KeyFilePath <string>] [-SSHTransport] [<CommonParameters>]
```

Stop-RemoteProcess Help

The last parameter set is the dynamic one. Instead of having to write several versions of the command, you can write a single function. Check out https://jdhitsolutions.com/blog/powershell/7458/a-powershell-remote-function-framework/[93] for a bit more detail on this function.

Custom Module Manifests

The last cross-platform scripting feature to consider is a custom module manifest. In the code downloads for this chapter you'll find a demo module called CrossDemo. The module has several commands, some of which will only work in PowerShell 7. The goal is to only export the commands (and aliases) that are supported. Here's how.

Normally, a psd1 is static and can't contain code. But there is an exception for a module manifest. You can use a simple If statement to tell PowerShell what functions to export.

```
FunctionsToExport = if ($PSEdition -eq 'desktop') {
    'Export-Data','Get-DiskFree'
    }
    else {
    'Export-Data','Get-DiskFree','Get-Status','Get-RemoteData'
    }
...
AliasesToExport = if ($PSEdition -eq 'desktop') {
    'df'
    }
    else {
    'df','gst'
    }
```

When the module is imported on Windows PowerShell, only 2 functions and 1 alias are exported. PowerShell 7 systems get everything.

Even so, you still may need to fine-tune platform requirements. For example, if you look at our sample code for the `Get-DiskFree` function, you'll see code like this:

[93] https://jdhitsolutions.com/blog/powershell/7458/a-powershell-remote-function-framework/

```
if ($IsWindows -OR $PSEdition -eq 'desktop') {
        $Drive = (Get-Item $Path).Root -replace "\\"
        Write-Verbose "Getting disk information for $drive in $As"
        ...
    } #if Windows
    else {
        Write-Warning 'This command requires a Windows platform.'
    }
```

Because the function uses `Get-CimInstance` it requires a Windows platform. It will work in PowerShell 7 on Windows but not Linux. So even though we're exporting the function for PowerShell 7, it will only run on Windows. You'll need to keep things like this in mind. Will it work in PowerShell 7 and what are the platform dependencies?

Not every sample function in our demo module is using this logic. Feel free to update the code as an exercise.

We'll be honest with you and say that a lot of community accepted best practices for cross-platform scripting are still being developed. But if you use a little common-sense and follow the best practices that *are* accepted, you shouldn't have too much trouble.

Wish List

Some items we're considering:

- A chapter on basic GitHub usage, with a focus on forking existing projects and submitting PRs.
- Something on how to use generic lists and why you would want to.
- Scripting with regular expressions
- If there's a topic you think we're missing - send a message to @JeffHicks

Release Notes

2020-June-30

- Update Lab setup
- Updated Introduction
- Updated A Note on Code Listings
- New teaching examples and code in Part 1
- Added a chapter on working with CSV files in Part 4
- Added a chapter on Auto Completers to Part 5
- Added a chapter on custom formatting to Part 5
- Added a chapter on logging to Part 5
- Added Part 7 to cover PowerShell 7 scripting concepts and techniques
- Links updated and revised throughout the entire book
- Spelling, grammar and code clean up throughout the entire book
- Sample code files encoding set to UTF8

2019-Jul-12

- Markdown cleanup on all chapters
- Updated Author page
- Updates to Tips and Tricks
- Updates to Advanced Function Tricks
- Updated Lab Setup
- Updated many references and examples to reflect newer builds of PowerShell and VS Code
- code cleanup and reformatting for clarity

2018-Jul-15

- Refactoring Part 1

2018-Apr-5

- Finishing Pester content
- Added Command Tracing
- Cleaned up some erroneous backreferences

2018-Mar-30

- Continuing Pester content

2018-Mar-12

- Added Tips & Tricks chapter
- Continuing to build Pester content

2018-Mar-08

- Filling a bunch of the new Part on Pester; publishing Plaster chapter.

2018-Jan-21

- We're adding a whole new Part on unit testing to the book, and this release provides our scaffolding for it. We're also adding a new chapter on Plaster.

2017-May-17

- Major changes. You'll notice that all of Part 1 is entirely different. Due to some contractual disagreements over *Learn PowerShell Toolmaking in a Month of Lunches*, we've agreed to revise that book into *PowerShell Scripting in a Month of Lunches*, and to base its core narrative on the same topics that formerly comprised Part 1 of this book. If you have a previous edition of this book with the original Part 1, you're welcome to hang on to it. That positions the new *Month of Lunches* book as a "prerequisite" to this one. Part 1 of this book is now a lightning review of that content's core narrative, along with two opportunities for you to self-assess your comprehension of that content. If you do well in those assessments, then you're good to go on this book (and those assessments appear in this book's free sample, too). This actually works out okay for everyone, we hope - the stuff in Part 1 is really evergreen and fundamental, whereas the rest of this book is going to need updates for PowerShell v6 and later. So we'll continue to make those updates and additions, and leave the "entry level" content in the traditionally-published book. *This* book will continue to focus on professional scripting and toolmaking, with constant updates to accommodate new versions.
- This comprises the "Second Edition" of the book as sold on Amazon.

2017-Feb-24

- "First Edition" final

2017-Feb-23

- All chapters in draft

2017-Feb-18

- Finalizing several chapters
- Part 4 is nearly complete!
- Part 3 is nearly complete!
- Started "Using .NET Framework "Raw""
- Don't forget to run Update-Module against PowerShell-Toolmaking, so that you have the latest sample code

2017-Feb-15

- Part 2 is now complete!
- Started "Working with SQL Server"
- Started "Graphical Controllers"
- Started "Tools for Toolmaking"
- https://gitpitch.com/concentrateddon/ToolmakingSlides/master?grs=github&t=black[94] offers a slide deck and recommended delivery sequence, enabling the book to be used as a classroom text more easily. This release begins the presentation; it'll be finished in a future release.

2017-Feb-10

- "Controlling Your Source"
- Many of the previous chapters are now finalized

2017-Feb-6

- "Publishing Your Tools"
- "Dynamic Parameters"
- "Working with XML Data"
- Starting "Proxy Functions"
- Starting "Unit Testing Your Code"
- Updates to JSON chapter
- Started "Analyzing Your Code"
- Starting "Extending Output Types"
- Starting "Advanced Debugging"
- Starting "Converting a Function to a Class"
- Note that the online version may not provide access to front matter; we urge readers to rely primarily on one of the downloadable formats

2017-Jan-30

[94] https://gitpitch.com/concentrateddon/ToolmakingSlides/master?grs=github&t=black

- “Writing Full Help”
- “Working with JSON”
- Minor fixes throughout
- If you see paths (in non-code font especially) missing backslashes, please let us know. We need to use forward slashes since the backslash is a Markdown escape character.

2017-Jan-16

- Release notes moving to reverse chronology
- You’ll find some partially complete chapters here - they’re noted as such

2017-Jan-13

- Writing Full Help
- Tech review of all but “Error Handling” in Part 1
- We will now indicate draft (pre-tech-reviewed) chapters at the top of the chapter.

2017-Jan-31

- Initial release of Part 1.